

WebSphere™ Application Server Enterprise Edition
Component Broker



Application Development Tools Guide

Version 3.5

WebSphere™ Application Server Enterprise Edition
Component Broker



Application Development Tools Guide

Version 3.5

Note

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 911.

Second Edition (July 2000)

This document replaces SC09-4448-00.

Order publications through your IBM® representative or through the IBM branch office serving your locality.

© Copyright International Business Machines Corporation 1999, 2000. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About This Book	xi	Rose properties and bridging guidelines	70
Who Should Read This Book	xi	Rose Bridge export	79
Conventions used in this book	xi	Exporting a design from Rose	80
Related information	xiii	Tutorial: Exporting from Rose	81
How to send your comments	xiii	Working with an exported design	88
What's new in the Application Development Tools Guide	xiii	Rose Bridge import	89
		Importing a project into Rose	92
		Tutorial: Importing into Rose	94
		Rose to Object Builder mapping rules	97
Chapter 1. Object Builder overview	1	Projects in Rose	98
Object Builder	1	Modules in Rose	99
Restrictions for R3.5	3	Constructs in Rose	101
Composition restrictions	16	Class properties in Rose	109
Projects and models	17	Attribute properties in Rose	113
Developing in Object Builder	19	Package properties in Rose	116
Platform-specific information	20	Method properties in Rose	116
OLT and Debug API support	23	Class relationships in Rose	119
Setting up Object Builder	24	Object Builder to Rose mapping rules	123
Object Builder environment variables	24		
Opening a project	26	Chapter 4. Creating a component.	127
Setting Object Builder preferences	27	Naming objects	128
Color Assignments	28	Internationalization of data	132
Find and Replace (Source page)	29	Creating a component for transient data	135
Indentation (Source page)	30	Creating a component for new DB data	136
Searching the Tasks and Objects pane	30	DDL	137
Checking a model for consistency	31	Package file	138
Requirements for Java development	32	DBCS and Binary Data Support	138
Migrating projects from 3.0	33	Creating a component for existing DB data	139
Migrating old projects	33	Supported CORBA Types	140
Migrating from the command line	35	DB2 data type mappings	142
Migrating a team environment	36	Oracle data type mappings	146
		Informix data type mappings	148
Chapter 2. Tutorials for Object Builder	39	Mapping DBCS data types	149
Tutorial: Creating a component with transient data	39	Data encoding schemes	152
Tutorial: Creating a component for new DB data	50	OOSQL keywords	154
Additional tutorials	62	Keywords for query support	154
		Null value tolerance with sentinel values	155
Chapter 3. Using Rational Rose with Object Builder	63	Creating a component for PA data	157
Rose	64	Enterprise Access Builder (EAB)	158
Setting up Rose 98	65	Procedural adaptor bean (PA bean)	159
Setting up Rose 98i and Rose 2000	66	Java data type mappings	159
Importing Component Broker frameworks	68	Supported platforms for connectors	160
The Rose Bridge	69	Customizing PA bean query methods	160
		Mapping query method parameters to PA bean attributes	161

Handling exceptions thrown by PA bean push-down methods	163	Defining a 1-1 relationship	279
Adding resource methods to a sessional business object	164	Distributed query	280
Tutorial: Unit test for procedural adaptors	166	Defining a 1-n relationship	281
Tutorial: Creating a component for PA data (bottom-up)	167	Defining a circular relationship	283
Tutorial: Creating a component for PA data (meet-in-the-middle)	177	Foreign key patterns	284
Creating a component for an inbound message	187	Defining a foreign key pattern	285
Tutorial: Creating an inbound message application	188	Storing an object reference as a handle	288
Creating a component for an outbound message	201	Using complex relationships in SQL clauses	289
Tutorial: Creating an outbound message application	203	Design patterns and iterators	291
Reusing existing objects	215	Dependencies within an IDL file	292
Creating a local-only object	216	Relationship Implementation	293
Local-only objects	216	Local Persistent Reference	294
Creating a local-only object file	217	OOSQL Implementation	295
Adding a local-only object module	218	Foreign Key Implementation	296
Adding a local-only object	219	Checking for null foreign key values	298
Tutorial: Creating local-only objects	220		
Tracking data types in models	235	Chapter 6. Inheritance.	299
Type Usage	238	Inheritance and overriding	300
File and method adornments	239	Inheritance and overriding in helper objects	300
Adding file adornments	240	Inheritance and overriding in business objects	301
Adding method adornments	242	Inheritance and overriding in data objects	303
Business Object Behavior	244	Abstract base class inheritance	303
Pattern for Handling State Data	245	Choosing an inheritance pattern for persistence	304
Object Reference	246	Creating a child component	306
Data Object Interface	247	Inheritance with attributes duplication	307
Session Service	248	Defining a child with attributes duplication	309
Data Object Behavior	249	Tutorial: Inheritance with attributes duplication	310
Environment	249	Inheritance with key duplication	322
Type of Persistence	251	Defining a child with key duplication	325
Data Access Pattern	254	Tutorial: Inheritance with key duplication	327
Handle for Storing Pointers	255	Inheritance with a single datastore	341
Data Object Implementation Inheritance	257	Defining a child with a single datastore	342
Objects to source files mapping	258	Tutorial: Inheritance with a single datastore	344
Chapter 5. Components working together	261	Inheritance with views	357
Creating a composite component	261	Defining a child with views	360
Composite component	262	Tutorial: Inheritance with views	362
Composition	263	Building a child component	380
Composite business object	264	Packaging a child component	381
Composite key	265		
Tutorial: Composite component creation	267	Chapter 7. Working with external files	383
Defining relationships	278	External files for method bodies	383
		Importing edited source files	385
		Importing C++ or Java classes	386

Exporting XML	387	Tutorial: Team development with Rose	449
Importing XML	389	Setting up a team environment	457
Chapter 8. Working with enterprise beans	391	Project divisions in a team environment	458
Importing enterprise beans into Object Builder	392	Splitting up a project for team development	459
Java to Object Builder type mapping	394	Adding an integration project to a team environment	460
Keys for enterprise beans	395	Cross-project dependencies	462
CMP Entity Bean-Specific Settings	396	Change control.	463
BMP Entity Bean-Specific Settings	399	Setting up a change control process	464
Session Bean-Specific Settings.	400	Setting up an automated build process	465
Importing enterprise beans using Object Builder	401	Setting up a team development environment	467
The EJB Deployment Tool	402	XML-based change control.	469
Importing enterprise beans from the command line	403	Setting up XML-based change control for CMVC	471
Importing enterprise beans from VisualAge for Java	405	Change control sample	473
Deploying enterprise beans	408	Customizing XML-based change control	475
Deploying enterprise beans using Object Builder	410	Template interpreter format	478
Deploying enterprise beans using the EJB Deployment Tool	411	Working in a team environment	480
Deploying enterprise beans into a polymorphic home	412	Creating a project in a team environment	481
Working with deployed EJB JAR files	413	Checking out files.	482
Creating a deployed EJB JAR file	414	Checking in files	483
Editing an EJB JAR file	415	Extracting files.	484
Deleting an EJB JAR file	415	Editing a project in a team environment	485
Working with deployed enterprise beans	416	Deleting a project in a team environment	486
Creating a deployed enterprise bean	416	Building DLLs in a team environment	487
Editing an EJB class	416	Packaging an application in a team environment	489
Deleting an EJB class.	417	Testing cross-project applications with QuickTest	489
Chapter 9. Multi-platform development	419	Maintaining a team environment	490
Multi-platform development	419	Moving a project	491
Setting platform constraints	421	Model interchange with XML	492
Deployment platforms	423	XML interchange files	493
Platform differences	425	Changing project divisions.	496
Cross-platform development	426	The Compare and Merge Tool for XML	497
Tutorial: Developing a multi-platform application	429	Comparing files with the Compare and Merge Tool for XML	497
Chapter 10. Team development	443	Merging files with the Compare and Merge Tool for XML	498
Developing as part of a team	444	Managing cross-project dependencies	499
Working with Rose in a team environment	444	Documenting projects	501
Exporting a Rose design to a team environment	445	Chapter 11. Customizing Object Builder	503
Importing a Rose design from a team environment	447	XML wizards	503
		Creating an XML wizard	504
		XML template design	505
		Starting the SmartGuide Customizer for XML	506

Defining XML wizard macros	507	Build targets	567
Customizing value lists in an XML wizard	510	Build options	568
Deriving values in an XML wizard	511	Remote build configuration	570
Propagating values in an XML wizard	513	Remote build	570
Constraining values in an XML wizard	515	Pass ticket	571
Defining the layout of an XML wizard	516	Profile	571
Testing an XML wizard	517	Launching a remote OS/390 build	572
Running an XML wizard	518	Tutorial: Launching a remote OS/390 build	573
Editing an XML wizard	519	Packaging applications	574
Distributing an XML wizard	519	Creating an application family	575
Attribute identity in XML	520	Adding an application	576
XML ID attributes	521	Container	578
XML references	522	Creating a container instance	578
XML references with customized targets	525	Home	581
XML wizard properties	530	Polymorphic homes	581
Macro setting	530	Managed object configuration behavior	584
Element properties	530	Home Name	584
Attribute properties	531	Home Options	587
Attribute identity options	533	Configuring a managed object	588
Constraints	535	Parent Interface for Polymorphism	592
Filters	536	Generating the DDL files	593
Filtering the Tasks and Objects pane	537	Documenting applications	595
Creating a filter for the Tasks and Objects pane	537	Container behavior	595
XML browsing with XSL	538	Container service	596
Browsing XML files	539	Container policies	597
Setting up for XSL	540	Behavior for Methods Called Outside a Transaction	599
The XSL sample	541	Behavior for Methods Called Outside a Session	600
Viewing a sample XSL-based document	543	Connection	601
Applying the sample XSL style sheet	544	Application DDL files	602
Creating your own XSL style sheet	546	Creating DDL files	603
Applying an XSL style sheet	547	The structure of a DDL file	604
Chapter 12. Configuration	549	Chapter 13. Testing applications with QuickTest	611
Configuring builds	549	QuickTest	611
Specifying a build location	550	Generating QuickTest client applications	614
Generating code	551	Running QuickTest client applications	615
Defining a client DLL	552	QuickScript	616
Defining a server DLL	554	Recording QuickScript	617
Generating a makefile	556	Compiling the QuickScript file	618
Specifying the order of a build	558	Running QuickScript	618
Building the DLLs	558	The QuickTest framework	620
Building the JAR files	561	QuickTest with the Component Broker Programming Model	622
Building for ActiveX clients	562	QuickTest with Java and JFC	634
Building for Java clients	563	QuickTest-generated files	647
Building for QuickTest	564	Running the QuickTest tutorial	647
Rebuilding DLLs	565		
Build configuration behavior	566		
Default Configuration	566		

Chapter 14. Command-line interfaces . . .	659
Using Object Builder from the command line	659
obmigrate	659
Exporting XML from the command line . . .	660
obexport	661
Importing XML from the command line . . .	663
obimport.	664
Importing IDL from the command line. . .	666
importidl	667
Creating a local-only object by importing	
an IDL file	669
Importing enterprise beans from the	
command line	670
cbjeb options	673
Importing edited source files from the	
command line	682
importimpl	683
Generating code from the command line . .	684
obgen.	685
make options	688
obcheck	694
Chapter 15. Object tasks	697
Working with components	697
Working with attributes.	697
Attributes	698
Adding an attribute	699
Editing an attribute	700
Deleting an attribute.	701
Setting sentinel values for null field	
values.	701
Mapping a data object to a persistent object	703
Mapping a data object to a DB persistent	
object	703
Mapping a data object to a PA persistent	
object	708
Mapping a data object to the parent's	
persistent object	711
Mapping a data object to the child's	
persistent object	712
Customizing referential integrity. . . .	714
Attribute mapping properties.	716
Data Object Attributes	716
Key Attributes	717
Mapping helper class	719
Mapping Patterns.	722
Patterned Attribute Mapping Selection	724
Persistent Object Attributes	726
Schema Columns	728
Mapping data object attributes to persistent	
object attributes	730
Mapping attributes using the Primitive	
pattern	731
Mapping attributes using a key	732
Mapping helper	735
Mapping attributes using a mapping	
helper.	738
Mapping business object reference	
attributes	744
Complex attributes and mapping patterns	745
Mapping complex attributes using the	
Primitive pattern	746
Mapping complex attributes using the	
Explode pattern	748
Working with methods	750
User-defined methods	751
Implementing methods	752
Adding an initializer method	753
Editing user-defined methods.	754
Get and set methods	755
Editing get and set methods	756
Framework methods.	757
Editing framework methods	757
Special framework methods	758
Editing special framework methods. . .	758
Push-down methods	759
Working with PA bean push-down	
methods	760
Using push-down methods with DB	
persistent objects	761
Using push-down methods with PA	
persistent objects	762
Relationship methods	764
Customizing business object OOSQL	
implementation methods	765
Customizing persistent object ESQL	
framework methods	766
Deleting a method	767
Method mapping properties	767
Special Framework Methods	768
User-Defined Methods	768
Method Reordering	769
Working with constructs	769
Constructs	770
Defining constructs with file scope . .	770
Defining constructs with module scope	771
Defining constructs with interface scope	772
Editing a construct	773
Deleting a construct	774

Working with business objects	774	Working with DB persistent objects	832
Creating a business object file.	775	Adding a persistent object and schema	833
Adding a business object module	777	Adding a persistent object from a DB	
Adding a business object interface	777	schema	837
Creating a business object by importing		Editing a DB persistent object.	838
an IDL file	780	Deleting a DB persistent object	839
Adding a business object implementation		Working with DB schema groups	840
and data object interface	780	Creating a DB schema group	840
Adding a business object from a data		Editing a DB schema group	841
object	784	Deleting a DB schema group	843
Mapping a business object to a data		Working with DB schemas	843
object	787	Creating a DB schema by importing an	
Editing a business object file	789	SQL file	844
Editing a business object interface	791	Re-importing an SQL file	847
Editing a business object implementation	792	The SQL View Editor	849
Editing a business object implementation		Working with the SQL View Editor	850
file.	793	Creating a view with the SQL View	
Deleting a business object interface	794	Editor.	850
Deleting a business object implementation	794	Editing a view with the SQL View Editor	851
Working with data objects	795	Editing a view	853
Creating a data object file	797	Editing a DB schema.	855
Editing a data object file	798	Editing a generated SQL file	857
Adding a data object module	800	Deleting a DB schema	859
Creating a data object interface	801	Working with PA persistent objects	860
Creating a data object by importing an		Adding a persistent object from a PA	
IDL file	804	schema	860
Adding a data object implementation	807	Editing a PA persistent object	861
Associating a persistent object with a data		Deleting a PA persistent object	861
object	811	Working with PA schemas	862
Adding a data object from a business		Creating a PA schema by importing a PA	
object	813	bean	862
Adding a data object from a DB		WHERE clause syntax	866
persistent object	814	Editing a PA schema	868
Adding a data object from a PA persistent		Deleting a PA schema	869
object	815	Working with managed objects	869
Editing a data object interface.	817	Service to Use	870
Editing a data object implementation	819	Adding a managed object	871
Associating a PA persistent object with an		Editing a managed object	872
existing data object implementation.	822	Editing a managed object file	873
Editing a data object implementation file	823	Editing a managed object configuration	874
Deleting a data object interface	824	Deleting a managed object.	874
Deleting a data object implementation	824	Deleting a managed object configuration	875
Working with keys	825	Working with specialized homes	875
Adding a key	826	Creating a specialized home	876
Editing a key	828	Example: Creating a specialized home	878
Deleting a key	829	Editing a specialized home.	879
Working with copy helpers	829	Deleting a specialized home	879
Adding a copy helper	830	Creating a specialized polymorphic home	880
Editing a copy helper	831	Querying abstract classes	882
Deleting a copy helper	832	Working with container instances	883

Editing a container instance	883	Troubleshooting	905
Deleting a container instance	884	Appendix. IR Browser.	907
Working with compositions	884	Starting the IR Browser	907
Creating a composition file.	885	Viewing objects in the repository.	907
Adding a composition module	886	Viewing the definition of an object	907
Adding a composition	886	Viewing relationships between objects	907
Editing a composition	889	Viewing the operations of an interface	908
Working with composite business objects	891	Searching the repository	908
Adding a composite business object		Finding an object	908
interface	892	Searching with wildcards	909
Adding a composite business object		Finding an interface's referencing	
implementation and data object interface .	894	operations	909
Editing a composite business object		Searching by object type	909
interface	898	Deleting objects from the repository.	910
Editing a composite business object		Notices	911
implementation	899	Trademarks and service marks	913
Working with composite keys.	900	Index	917
Adding a composite key	901		
Editing a composite key	903		
Chapter 16. Troubleshooting	905		

About This Book

The *Application Development Tools Guide* provides detailed information about generating multi-tier applications in the VisualAge[®] Component Development environment.

Who Should Read This Book

The *Application Development Tools Guide* is intended for application programmers who want to use Object Builder and its associated tools to:

- create, execute and manage distributed applications across network computing environments.
- connect multiple backend systems to dynamic, new applications.
- capture information from database systems, transaction processing systems, and applications, as highly manageable components.

Conventions used in this book

The following conventions distinguish different text elements.

plain	Window titles, folder names, icon names, and method names.
monospace	Programming examples, user input at the command line prompt or into an entry field, directory paths, and user output.
bold	Menu choices and menu names, labels for push buttons, check boxes, radio buttons, group-box controls, drop-down list boxes, combo-boxes, notebook tabs, and entry fields.
<i>italics</i>	Programming keywords and variables, titles of documents, and initial use of terms that are in the glossary.

The following conventions are used to abbreviate menu selections and object expansions within a user interface:

> The right arrow when used within a menu shows a series of menu selections. For example, **File > New** is translated to mean:
"On the **File** menu, click **New**."

The right arrow, when used within a tree view, denotes a series of folder (or object) expansions. For example, Expand **Management Zones > Sample Cell and Work Group Zone > Configurations** is translated to mean:

1. Expand Management Zones.
2. Expand Sample Cell and Work Group Zone.
3. Expand Configurations.

+ The plus sign (+) denotes where a tree structure can be expanded to show more objects. To expand, click the plus sign (+) beside any object. If you double-click the object itself, a new tree structure is displayed with that object as the root of the tree.

- The minus sign (-) denotes where a tree structure can be collapsed to remove its objects from view. To collapse, click the minus sign (-) beside any object.

left mouse button The left mouse button is used for all actions in the application except for opening the pop-up menu of an object. Unless otherwise stated, the left mouse button is assumed.

right mouse button The right mouse button opens the pop-up menu of an object. The pop-up menu contains a list of actions that can be performed on that object. The list of options varies depending on the type of object.

The following icons are used to flag new and changed information:



Denotes information that was added for Component Broker Version 3.5



Denotes information that has changed significantly between Versions 3.0 and 3.5

The following icons are used to indicate platform-specific sections:



Denotes a section that applies to the Windows[®] 95 or Windows NT[®] platforms.

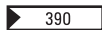
Note: The Windows 95 platform only supports the Component Broker Java[™] client.



Denotes a section that applies to the AIX[®] platform.



Denotes a section that applies to the Solaris platform.

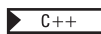


Denotes a section that applies to the OS/390[®] platform.



Denotes a section that applies to the HP-UX platform.

The following icons are used to flag programming language-specific information:



Denotes a section that applies to C++.



Denotes a section that applies to Java.

Related information

A complete list of books in the Component Broker library can be found in the preface to the *Planning, Performance and Installation Guide*. All Component Broker documentation is available in PDF form, accessible from the Component Broker online library at the following URL:

[http://localhost:49213/cgi-bin/vahwebx.exe/\\$lang/cbdoc/Extract/0/hgpdf.htm](http://localhost:49213/cgi-bin/vahwebx.exe/$lang/cbdoc/Extract/0/hgpdf.htm)

Note: For this URL to work, the Component Broker documentation (and supporting code) must be installed on the host on which your Web browser is running. For more information about installing the Component Broker documentation, see the *Planning, Performance, and Installation Guide*.

The PDF files can also be accessed from the Web, at the following URL:

<http://www.ibm.com/software/webservers/appserv/library.html>

Documentation for the IBM Distributed Debugger is available in online or PDF form, and is accessible from the `x:\IBMDebug\help` subdirectory, or through the help menus in the OLT or debugger user interfaces.

How to send your comments

Your feedback is important in helping to provide the most accurate and highest quality information. If you have any comments about this book or any other WebSphere Application Server Enterprise Edition documentation, send your comments by e-mail to waseedoc@us.ibm.com. Be sure to include the name of the book, the document number of the book, the version of WebSphere Application Server Enterprise Edition, and, if applicable, the specific location of the information on which you are commenting (for example, a page number or a table number).

What's new in the Application Development Tools Guide



Documentation updates

The documentation for WebSphere Enterprise Edition Component Broker, Version 3.5 includes:

- New information (page xiv)

- Changes to existing information (page xviii)

Two new symbols have been introduced to show you what information topics are new or have been changed since Version 3.0:

-  Denotes information added for Component Broker Version 3.5
-  Denotes information that has changed significantly between Versions 3.0 and 3.5

New information

The following new Object Builder information has been added in Version 3.5:




Internationalization of data

The documentation now contains the set of characters valid for use in object names and attributes in system management configuration data. For error-free communication between different code pages, this is the set you must use (a restricted set of ASCII characters). Since many of the objects that you create in Object Builder are ultimately loaded into System Management (installed on the server), when you deploy your applications, this is the set that is valid for use for all Object Builder entities as well. See “Internationalization of data” on page 132, and the related topic: “Naming objects” on page 128.

Polymorphic homes

Component Broker currently supports polymorphic homes with the single table inheritance pattern. See “Polymorphic homes” on page 581.

Informix application adaptor support

   This version of Component Broker supports Informix databases. However, the support is limited only to the Windows NT, AIX, and Solaris platforms. For more restrictions on this support, see “Restrictions for R3.5” on page 3.

Specifying the order of a build

You can design your build system in such a way that the projects get built in an order, which is compatible with the logical layering of the system. See “Specifying the order of a build” on page 558.

Propagation of changes to secondary models

When you edit a project in a team environment, some of your changes may affect other projects, and generated code. You can now either make the dependent projects writable, and have Object Builder propagate the changes

correctly, or cancel out of the editing process, undoing any changes you have made to the project. See “Editing a project in a team environment” on page 485.

Distributed query

The choice of having a distributed query is available for all platforms: Windows NT, AIX, Solaris, and HP-UX. For the last release, it was limited to the Windows NT, and AIX platforms.

Editing DB schemas and persistent objects

In previous releases, if you had to make changes to your schema, or the persistent objects that mapped to it, you had to delete it and start over again, which meant you lost all mapped data object implementations, their attributes and framework methods. Now, you can edit DB schemas and the persistent objects that map to them. Changes you make are propagated down to the objects that map to them (using default mappings). See “Editing a DB schema” on page 855 and Editing a DB persistent object.

Method adornments

Now, in addition to being able to add adornments to the top of files, you can also add adornments to methods. This is useful for adding comments (such as Javadoc) to methods. See “File and method adornments” on page 239.

Iconic changes indicate XML read-write or read-only state

Objects in the Tasks and Objects pane are represented by a different icon when the XML for the object is read-only. In this state the pop-up menu for the object has only a subset of the items it has when the object is in the read-write state.

This feature is disabled by default. If you want to enable it, select **File > Preferences > Tasks and Objects > Environment**, and select the **Enable team library check-in and check-out check** box.

Single file speed-up build

In this process, the DLLs are built so that the header files are not processed multiple times. This reduces the time required for the build. You can specify this type of build both through Object Builder’s interface (**File > Preferences > Tasks and Objects > Make File Generation**, and select the option **Minimize C++ compiler invocations**), and through the command line (specify `IVB_COMBINE_SOURCE=1`).

Recognition of IVB_UNOPTIMIZE=0 and IVB_OPTIMIZE=0 configuration options for make and nmake

You can now use `IVB_UNOPTIMIZE=0` instead of `IVB_OPTIMIZE=1`, and `IVB_OPTIMIZE=0` instead of `IVB_UNOPTIMIZE=1` when you run `make` or `nmake`.

Enterprise bean deployment

Enterprise beans can now be deployed on the OS/390 and Solaris platforms in addition to the Windows NT, and AIX platforms.

Support for the EJB Programming Model

To support the EJB Programming Model, the Component Broker Programming Model now permits the keys and copy helpers for a business object to contain attributes that are defined only on the business object implementation, as well as on the business object client interface.

OLT support

In a prior version, the emitted managed object code for Solaris and OS/390 was adjusted to exploit a new set of OLT and Debug APIs. In this version of Object Builder, the same changes in code are applied to the managed objects for the Windows NT and AIX platforms.

Object-specific platform constraints

You can now set object-specific platform constraints on application families besides being able to set them on other objects such as business object interfaces, business object implementations, data object interfaces, data object implementations, managed objects, containers, and DLLs.

Solaris 3.5 support

There is a distinction between R3.5, the CB version; and 3.5, the CB run-time level of function (characterized by the run-time function of R3.0 plus the run-time features for R3.5).

For version 3.0 (R3.0), the following platforms are supported in the CB run-time level of function:

- CB/NT 3.0
- CB/AIX 3.0
- CB/Solaris 2.0
- CB/390 1.2

For version 3.5 (R3.5), the following platforms are supported in the CB run-time level of function:

- CB/NT 3.5
- CB/AIX 3.5
- CB/Solaris 3.5
- CB/390 3.02
- CB/HP-UX 3.0 (beta)

* A minimal subset (from a tooling standpoint) of the 3.5 net function on the workstation.

► WIN ► AIX ► SOLARIS All R3.5 Object Builder features that exploit new run-time function in 3.5 will be enabled for models and artifacts that are deployed to the Windows NT, AIX and Solaris platforms.

► WIN ► AIX ► 390 ► SOLARIS ► HP-UX All R3.5 Object Builder features, which are pure Toolkit features will be enabled for models and artifacts that are deployed to all platforms.

► SOLARIS Several run-time functional enhancements were made in R3.0. Since the Solaris run time is stepping directly from an R2.0 level of function to an R3.5 level of function, Object Builder retroactively enables exploitation of these R3.0 run-time features on Solaris.

► SOLARIS **Types of attributes allowed in a copy helper**
Object Builder now allows you to select business object attributes of complex types (structures, unions, sequences, arrays, anys) for addition to a copy helper.

Ability to select attributes defined on the business object implementation as key and copy helper attributes

To support the EJB Programming model, the CB programming model now allows the key and copy helper of a business object to contain attributes that are either defined on a particular business object implementation, and its interface; or attributes that are defined only on the interface.

Null value tolerance with sentinel values

You can now set sentinel values that will represent null field values when they are mapped to a CORBA object in Object Builder. See “Null value tolerance with sentinel values” on page 155.

SAP connection support

Object builder now lets you use a configure a container that uses PAA session services to use a SAP connector for connecting to a 3-tier SAP system or group of systems.

Creating a specialized home

The documentation now contains a sample of creating a specialized home. See Example: Creating a specialized home.

Associating a PA persistent object with an existing data object implementation

There is now additional information about how to associate a PA persistent

object with a data object implementation, along with information about possible pitfalls. See “Associating a PA persistent object with an existing data object implementation” on page 822.

Documentation of obcheck tool

The obcheck command is a command-line model consistency checker tool that verifies models built with Object Builder. Documentation for this tool is provided. See “obcheck” on page 694.

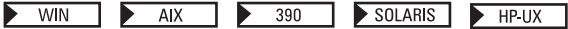
Bridging guidelines between Rational Rose and Object Builder

The documentation now contains precautions and recovery strategies to be adopted when bridging between the two tools. Rational Rose properties are explained, as well as the consequences of changing them. See “Rose properties and bridging guidelines” on page 70

Changes to existing information

Component Broker Version 3.5 also offers the following enhancements to the Object Builder documentation:

DDL file generation

 Object Builder generates the DDL file on each deployed platform. There is no distinction between specific and non-specific DDL files now. (Object Builder is capable of generating the file on all platforms - Windows NT, AIX, OS/390, Solaris, and HP-UX). You will need to load this file into System Management when you deploy your applications.


Your DDL file is generated along with a backup version. For example, if the name of your application family is MyAppFam, and you have specified the name of the DDL file as AppFam, the DDL emitter will generate the following files:

```
Working\platform\output_directory\MyAppFam\AppFam.ddl
Working\platform\output_directory\MyAppFam\backup\AppFam.auto.ddl
```





platform is one or more of NT, AIX, Solaris, 390, or HPUX (the platforms on which you are deploying your application).

output_directory is each of NOOPT, PRODUCTION, TRACE, TRACE_DEBUG (that is, for a given platform, you get four sets of DDL files generated).

IR executables

 Object Builder now generates code which builds the IR executables based on business object files rather than managed object interfaces.

Consequently, you now specify the IR file name to be generated when you are defining the business object files. In addition, migration and importing of older models and XML, respectively, will result in the elimination of old IR file names (based on the managed object interfaces) and the creation of new IR file names (based on the business object files).

    Object Builder generates the IR executable based on the managed object IDL file name. `idlc -eir` is run against the managed object IDL file to produce a C++ source file. This file is then compiled and linked to produce the IR executable. The IR executable's name is the managed object IDL file's root name with the suffix `_IR`. This is the executable that the DDL will run when System Management loads.

 This file gets the `.exe` extension only on Windows NT.

 **C++ run time in every DLL**

Component Broker ships its own copy of DLLs (with the server changing the prefix of these libraries from `'cpp'` to `'som'`). Object Builder's new default behavior has the C++ run time dynamically linked into each DLL that is being built. After linking, `DLLRNAME` is called against the DLLs to rename them to equivalent versions shipped with CB. As a result, the DLLs that are built are dynamically linked against our renamed version of these DLLs. This ensures that the DLLs are always there even though the Toolkit (and the compiler) are not required to exist on the server.

This default behavior improves performance since the DLLs that dynamically link to the C++ run time are noticeably smaller. Besides, run-time scalability is improved with the capacity to run a large number of objects.

1-n relationships with inherited classes

You can now have diagonal 1-n relationships. That is, a 1-n relationship is possible between objects in the following situations:

- There are three objects: Department, Person, and Employee. Person has a foreign key, and an object reference to Department. Employee is the child object of Person, and inherits Person's interface, but has no direct object reference to Department. A 1-n relationship is possible from Department to Employee.
- There are three objects: Person, Department, and Research. Research inherits Department's interface. Person has a foreign key, and an object reference to Department. Person does not have an object reference to the child object Research. You can set up a 1-n relationship from Research to Person.

Overriding the working directory

For each platform, you can specify what the absolute path is for the working

directory. This path is emitted to the generated make files. This way, when you move the generated code to another location (for example, another machine or another platform), you do not have to fix the paths in the generated files. You can set the target directory before you generate, and it will be used in the generated files.

FlowMark[®] support removed

The FlowMark (or, as it is now called, IBM MQSeries[®] Workflow) support previously provided with Object Builder has been removed.

Chapter 1. Object Builder overview

Object Builder

Object Builder is the development environment for Component Broker. You can use it to develop your application from start to finish, or start by designing in Rose and then import the design into Object Builder, where you add the final objects and program logic.

Object Builder supports the CORBA programming model using IDL, C++, and Java. You can generate complete working applications, including full client-server packages complete with server setup scripts.

You can use Object Builder to:

- develop new applications
- wrapper existing applications
- add new function to existing applications
- package an application

In order to build the application DLLs you define in Object Builder, you will need the Component Broker Server SDK installed, as well as any prerequisite application development software.

The model for your application is constructed out of components. Many of the development tasks in Object Builder revolve around defining components.

The Object Builder user interface is divided into panes, which provide access to different views of your application. Most interactions with Object Builder are through the panes and the pop-up menus for the objects in these panes. Object Builder has the following panes:



Tasks and Objects pane


This pane contains multiple folders. These folders organize the component objects as they are created. The objects in these folders represent a rough task flow through your use of Object Builder.

- The *Framework Interfaces* folder shows the framework interfaces provided by Object Builder.
- The *Local-Only Objects* folder is the folder for any local-only objects you define in Object Builder.
- The *User-Defined Business Objects* folder, the *User-Defined Data Objects* folder, the *DBA-Defined Schemas* folder, the *User-Defined PA Schemas* folder, and the

User-Defined Compositions folder are the folders you use to define component objects and show the component objects already defined.

- The *Non-IDL Type Objects* folder is the folder for C++ and Java objects defined outside of Object Builder.
- The *Build Configuration* folder is the folder where you configure component objects into the DLL files.
- The *Application Configuration* folder is the folder where you configure the DLL files into applications.

  **Note:** DLL files are called “shared library” files and are in the format lib*.so.

 **Note:** DLL files are called “shared library” files and are in the format lib*.sl.

- The *Container Definition* folder and the *Default Homes* folder show the container and homes with which your applications can be configured.
- The *Enterprise Beans* folder is the folder where you import and work with enterprise beans.

Inheritance pane


This pane shows the inheritance structure for the selected component object. You can switch between the interface inheritance view and the implementation inheritance view. You can also turn off this pane, giving more room to the Methods pane, by either using **View > Minimize Pane**, or clicking the Minimize button on the upper left corner of the pane. You can also detach the pane from the Object Builder main window using **View > Detach Pane**.

Methods pane

This pane lists the methods, attribute getters and setters, relationship methods and adornments for the object selected in the Tasks and Objects pane.

Source pane

This pane is used to edit the implementations for the selected method from the Methods pane. When a method is selected, its source code is displayed in this pane. It also displays the generated code for a particular object, when you select the **View Source** option from an object’s pop-up menu.

 To start Object Builder from the Windows NT **Start** menu, select **Programs > IBM Component Broker for Windows NT > Object Builder**. Object Builder takes a few moments to start. The Specify Directory wizard is opened. Enter a directory name (for example, x:\CBroker\MyProject) and click the **Finish** button. If this directory does not yet exist, a message is displayed asking if you want to create it. Click the **Yes** button, and provide a name for the new model.

▶ AIX To start Object Builder from the command line, enter `ob`. Object Builder takes a few moments to start. The Open Project wizard is opened. Enter a directory name (for example, `$HOME/MyProject`) and click the **Finish** button. If this directory does not yet exist, a message is displayed asking if you want to create it. Click the **Yes** button, and provide a name for the new model.

RELATED CONCEPTS

Components (*Programming Guide*)
"Projects and models" on page 17
Application architecture (*Programming Guide*)

RELATED TASKS

"Opening a project" on page 26
"Filtering the Tasks and Objects pane" on page 537
"Chapter 3. Using Rational Rose with Object Builder" on page 63
"Creating a component for transient data" on page 135
"Creating a component for new DB data" on page 136
"Creating a component for existing DB data" on page 139
"Creating a component for PA data" on page 157
"Chapter 8. Working with enterprise beans" on page 391

RELATED REFERENCES

"Internationalization of data" on page 132
"Naming objects" on page 128

Restrictions for R3.5

▶ CHANGED

The following restrictions apply to Object Builder version 3.5:

Supported characters for internationalization of data

You must use characters only from the displayable portion of the invariant ASCII character set for Object Builder entities that will ultimately be loaded into System Management (installed on the server) when you deploy your applications. These objects include application families, applications, managed objects, DLL names, application DDL files, and so on), along with some text string properties, (for example, database names). For rules to be followed when naming different Object Builder entities, see "Naming objects" on page 128.

Installation and Startup

CLASSPATH restriction

The CLASSPATH variable may not have contents longer than 1780 characters. The longer the name of your installation directory, the less space left for the

CLASSPATH value (Therefore, it is recommended that you use the default installation directory `x:\Cbroker`). If the CLASSPATH exceeds this limit, you will get a run-time error (“line too long”) when you start Object Builder. This is because commands (such as `ob.bat`), which invoke the Object Builder functions, prefix the Object Builder `.jar` files to the class path, and then invoke the Java code to run Object Builder.

Opening a project with disconnected network drives

If you open a project in Object Builder while there are disconnected network drives on your system, when you click **Browse** the network drives will be accessed and reconnected. This may take some time.

Heap size for Java Virtual Machine

If your project has 30 or more components, you may need to manually edit the `ob.bat` file to increase the maximum heap size for the Java Virtual Machine used by Object Builder. The default heap size is 255m (`-mx255m`): increase the heap size by 5m for each component beyond 30.

Rational Rose

Rose design restriction

You should not restructure your design after exporting. If you restructure your design (for example, move a class from one package to another), the export process will treat the change as a combination add and delete, rather than a move. This would result in two definitions of the class in Object Builder (a new class definition for its new position, and the old class definition for its old position), which is not valid.

Using Object Builder

Linking to non-IDL types on AIX

When you define a shared library (client or server DLL) on AIX that contains references to a non-IDL type, the shared library statically links the code for the non-IDL type. For example, if some of your code uses the `IString` class, then the shared library that contains your code links statically to the library file that contains the `IString` object code.

Static linkage can multiply the size of your shared library file by a factor of 20 or more. The advantage of static linkage is that you do not need to ship the shared library file that defines the non-IDL type.


To link dynamically, and reduce the size of your shared library file, edit the makefile for your shared library file (DLL) and change the referenced file `libbmcl.a` to `libbmcls.a`. You must then ship `libbmcls.a` with your shared

library (include it on the Additional Executables page of the DLL wizard) in accordance with the Licensed Program Specifications for C Set++ for AIX.

Opening in editor puts model in use

If you open a schema or schema group in an editor (the **Open in Editor** pop-up menu choice), the directory (the directory that the editor is invoked from, likely the working directory) will be locked by the editor. When you close the editor, the model remains locked until you log off of Windows NT and log back on again, or the processes EVFXLXPM.EXE and IWFWTV35.EXE terminate automatically after 5 minutes of inactivity.

Makefile generation for QuickTest

 **390** You do not have the option of building for the QuickTest target when you build all targets, if the view platform (**Platform > View**) is OS/390.

QuickTest build does not result in an executable file

When you build for QuickTest (**Build > Out-of-Date Targets > QuickTest**, or use the command `nmake all.mak quicktest` at the command line), the resulting `qt.ksh` file that is created in the `/Working/platform/config/` is not executable. You must explicitly change its mode to executable before you can run it. See [Building for QuickTest](#).

The one-large-object-per-DLL build style

This new compilation style for makefiles is loosely coupled with the old one. But, the dependencies for the combined C++ file are incomplete. This means that some changes which would have caused a recompile of files under the old style will not have any effects with the new style. For example, if a `.ih` file is updated, building with `IVB_COMBINE_SOURCE=1` will not result in any changes. Moreover, if `IVB_COMBINE_SOURCE=0` is then used in an attempt to build with the old style, a complete rebuild according to the old style will occur.

If you have global scope definitions of identical names, which are in two different C++ files, you may get warnings or errors from the compiler when you use this option.

 **390** You can use this option on OS/390 only if you do not require debug information.

Silent exceptions

Not all exceptions are displayed in the user interface. After major actions such as saving a project, check Object Builder's command window for any exceptions. The command window is the window from which you started Object Builder, or the window that appeared in the background if you started Object Builder from the **Start** menu.

Objects

Naming restrictions for interfaces, modules, constructs, attributes, methods, and relationships

The name must conform to CORBA naming conventions. This means it must include only alphanumeric characters (letters and numbers), and must start with a letter: for example, `a1bc23` is acceptable; `a#bc23` and `1abc23` are not acceptable.

You cannot use the following keywords to name the interface:

- Java keywords
- IDL keywords

None of the following names can be used as interface names or module names:

- any method name in `Java.lang.Object`. These include names such as *clone*, *finalize*, *hashCode*, *notifyAll*, *wait*, *equals*, *getClass*, *notify*, and *toString*
- any name that is suffixed with *Package*, *Holder*, *Helper*, *Ref*, *_var*, or *_ptr*.
- *goto*

Also, CORBA does not allow the identifier of an object to be the same as that of its immediate parent in the namespace. For example, a business object, local-only or data object interface cannot have the same name as module that contains it.

Note: For attributes, the following additional restriction applies:

If you use OOSQL keywords like `KEY`, `REF`, `TYPE` and `WORK` as attribute names, Object Builder generates objects that cannot be used with the Query Service, and you will not be able to perform OOSQL queries. See “OOSQL keywords” on page 154, for a complete list.

390 OS/390 naming restrictions

When you create a persistent object and schema from a data object implementation, and OS/390 is one of the deployment platforms for the data object implementation, the persistent object class name must not exceed eight characters. Object Builder validates the length of the persistent object class when you create a persistent object from a data object implementation, but if you change the deployment platform after you have created the persistent object, be sure that you follow the rule. If not, Object Builder will truncate the name to the 8.3 format. This may result in two persistent object file names becoming identical after truncation, since Object Builder assumes the object’s file name to be the same as the persistent object class name.

The same restriction applies when you create a persistent object from a schema, and OS/390 is one of the deployment platforms.

In general, when OS/390 is one of the deployment platforms, the following names must not exceed eight bytes in length, and must start with an alphabet character, and have only alphanumeric characters in the other positions (2 to 8):

- the persistent object class name
- the IR executable name
- the library name of an OS/390-deployed DLL
- the names of additional executables for applications
- the names of non-IDL types

If the SM/DDDL file names break the above file-naming rule, you will get a warning message. It is not an error, though, since DDL files are not necessarily required to be copied into partitioned data sets (PDS).

Note: DB2® v5.2: The table names and user names are restricted to 18 bytes (18 SBCS, or 9 DBCS characters) for all platforms. For a list of complete length restrictions for tables, see “Naming objects” on page 128.

Restrictions when adding comments

When you add comments for the different objects in Object Builder (for example, a business object module), the comments are generated within language comment delimiters. You cannot use these delimiters: /* and */ within your comments.

Attributes

Restrictions for attributes and methods at the interface level

You cannot select a non-IDL type class that you have imported into Object Builder as the type of an attribute, or a method return type if you are defining attributes and methods for an object’s interface; you can use the non-IDL type class only if you are defining the attributes or methods for the object’s implementation.

An attribute that you define for an object’s interface can only have a public implementation. If you want to define attributes that are either private or protected, you must define them at the object’s implementation level.

Naming restriction for attributes of keys and copy helpers

The name of a key attribute cannot be the same as the name of the key it is defined in. The same restriction applies to the names of copy helper attributes.

Even similar names that differ only in capitalization are not permitted. For example, the copy helper named AccountCopy cannot have an attribute named accountCopy.

▶ WIN ▶ AIX ▶ SOLARIS ▶ HP-UX ▶ 390 The Windows NT, AIX, Solaris, OS/390 and HP-UX platforms do not support the CORBA unsigned long long, long double and fixed types.

▶ 390 Besides, OS/390 does not support the CORBA long long, wchar and wstring types.

Type of attributes available for use in copy helpers

▶ WIN ▶ AIX ▶ SOLARIS ▶ HP-UX On the Windows NT, AIX, Solaris and HP-UX platforms, business object attributes of all types are available for use in copy helpers.

▶ 390 On the OS/390 platform, the following attribute types are not available for use in copy helpers:

- CORBA constructed types except enumerations
- CORBA template types except strings
- *any*, and the types that are not supported at all (long long, wchar, wstring, fixed, typedef)

Type of attributes available for use in keys

▶ WIN ▶ AIX ▶ SOLARIS ▶ HP-UX On the Windows NT, AIX, Solaris and HP-UX platforms, business object attributes of the following data types are available for use in keys:

- CORBA base types (except *any*), enumerations, string, wstring
- object references - business object interfaces that are defined in the model (You can define a business object interface and use it as a key attribute's data type for another business object.)

▶ 390 On OS/390, keys can use all the same attributes that are available for use on the other platforms except for long long, wchar and wstring.

Requirement for copy helper attributes

Any attributes you include in the copy helper must also be included in the data object.

Inheritance

Business object interface inheritance restriction




Abstract base classes are supported by Object Builder. There are some restrictions when you use them:

- To use this pattern, define a business object interface for the abstract class, and do not provide an implementation.
- Any business object interface that derives from the abstract interface and includes an implementation, must override all the attributes and methods that were defined on the abstract interface.

Key and copy helper inheritance restriction

If the interface has one or more parents, the parent interface attributes are also available for selection. If the key or copy helper will inherit from the parent's key or copy helper, you should **not** select any of the parent interface attributes.

Polymorphic homes

-   Polymorphic homes support is not available on the OS/390 and HP-UX platforms
-  If a business object interface is deployable to OS/390, you cannot have a polymorphic home class as the parent for either the business object interface, or the implementation. That is, you cannot select the IPolymorphicHome class as a parent class for the business object interface, and you cannot select IPolymorphicHome class as a parent for the business object implementation. The same is true for a managed object that is deployable to OS/390: it cannot inherit from the ISpecializedPolymorphicHomeManagedObject class.
- Abstract classes must have queryable homes
- The same primary key class must be used for all types and subtypes
- All data for the types can be stored only in the same database type
- All polymorphic homes must exist on the same server
- Multiple parent inheritance is not supported by Query for polymorphic homes in this version of Object Builder
- Component Broker does not support polymorphism over configured managed objects using either the inheritance with attributes duplication pattern, or the inheritance with key duplication pattern. Object Builder enforces these restrictions, and supports only the single table pattern of inheritance for polymorphism.
- Object Builder does not support container configuration of atomic transactions for query (Start a new transaction)
- Object Builder does not support typed tables (this support is specific to DB2 Version 6)

Object Mapping

Mapping pattern restrictions (data object attributes to persistent object attributes)

- In general, user-defined mappings are not queryable, but object references are always queryable. **Key Home** mappings for object references, and **Exploded** mappings for structures are not primitive mappings, but can participate in object queries. So can handle mappings for object references. But, with handle-mapped object references, the query may not perform as well because the query engine cannot push down these references to the database.
- When you map a data object to multiple persistent objects, you must map all the primary key attributes of the data object to the corresponding key attributes of each of the different persistent objects.
- If you are mapping key attributes, you must select only those attributes of the persistent object that are defined as PO keys, to map to the key attributes. To check which attributes are PO keys, select the persistent object in either the User-Defined Data Objects folder or the DBA-Defined Schemas folder and use **Properties** from its pop-up menu to view the Persistent Object page.

Mapping using helper class

- Object Builder does not provide a default mapping from the data object implementation to the persistent object for the CORBA *any*, union, sequence or array data types; or for any typedef ultimately to the aforementioned types. In these cases, you must provide your own mapping helper class.
- When you are creating an application that involves wide character set data (for example *wcharor wstring*), and is required to store persistent object data in a column of data type CHARACTER, in a DB2 table, you must write your own mapping helper. For an example, see the reference section “Mapping DBCS data types” on page 149.
- When you map a data object attribute that is also specified as an attribute of the key for the corresponding business object, to multiple persistent object attributes using a mapping helper, all the persistent object attributes that are mapped must be persistent object keys.

Foreign Keys

Foreign key relationship restriction

You cannot have a foreign key relationship between business object interfaces that are contained in different modules. The interfaces can be contained in the same module.

Foreign key not propagated down to the schema

When you map attributes using a foreign key and create a persistent object

and schema from the data object implementation, it will not automatically create a foreign key constraint in the schema. That is, the FOREIGN KEY constraint is not created in the table's .sql description file.

DB Schemas and Persistent Objects

Importing schemas with no primary keys

Object Builder lets you import schemas for which no primary keys have been defined. However, these schemas can result in query exceptions at runtime.

To avoid this happening, you can either select **Properties** from the pop-up menu of the schema, and select any of the schema columns as the database key (Select the **DB Key** check box.), or before you import the SQL file, edit the source file and add a PRIMARY KEY constraint.

Schema group restrictions

A table associated with a schema of one schema group cannot reference a foreign key defined in a table within another schema group.

Multi-user database applications

It is recommended that only one user build the persistent object, and perform the bind operation.

A database server is required for each developer workstation that is expected to build persistent objects. That is, most developers must not have access to the production database; instead they must do "unit test" binds against their own test database.

Restrictions when adding a persistent object from a schema

You must map all schema columns to their corresponding persistent object attributes; otherwise you may get exceptions thrown at run time if you use the Query Service.

Restrictions when naming a persistent object attribute

A persistent object attribute name must not exceed 26 characters in length.

SQL

SQL statements supported for import

Currently, the only SQL statements supported are the following:

- CREATE TABLE
- ALTER TABLE
- DROP

- CREATE VIEW
- COMMENT ON

None of these statements must contain expressions or column functions. The CREATE VIEW statement must contain only a simple query (SELECT statement). Currently there is no support for unnamed columns, expressions, functions, or sub-selects in CREATE VIEW.

SQL files supported for import

- This release supports SQL DDL files for DB2 MVS™ 4.1 databases. If an SQL DDL file contains a mix of supported and unsupported statements, the supported statements will be imported.
- This version of Object Builder supports SQL DDL files that are compliant with the following DBMSs:

- DB2 MVS 4.1, and DB2 V5.2 (UDB)
Object Builder has limited support for UDB V6.1.
- Oracle:

   Oracle 8.0.5 databases are supported only on the Windows NT, AIX and Solaris platforms.

    Oracle 8.1.6 (Oracle 8i Release 2) databases are supported on the Windows NT, AIX, Solaris, and HP-UX platforms.

Oracle SQL files can be imported, but language elements that have no analog in DB2 will not be parsed correctly except for columns of the VARCHAR2 or NUMBER data type. Object Builder does not support any non-ANSI syntax construction such as Oracle comments (/...*/), SQL commands.

- Informix:
A given transaction cannot access more than one Informix database per CB server. To involve two Informix databases in a transaction, you must access each database from a different server.

  Informix Dynamic Server version 7.30 files are supported only on AIX and Solaris.








   Informix Dynamic Server version 7.31 files are supported only on Windows NT, AIX and Solaris.

SQL files with the DOUBLE, TIME and TIMESTAMP types will not load as-is into Informix. You can use a DBA design tool such as ERWin or DataAtlas to make appropriate changes.

- SQL files larger than 2 MB are not recommended.

Oracle

Restrictions when importing Oracle SQL files

-    Oracle 8.0.5 databases are supported only on the Windows NT, AIX and Solaris platforms.
-     Oracle 8.1.6 (Oracle 8i Release 2) databases are supported on the Windows NT, AIX, Solaris, and HP-UX platforms.
- Support for Oracle backend databases is limited to data objects that use the Oracle Cache Service only. That is, data objects that use embedded SQL, or any other type of persistence will not be able to access data stored in Oracle databases.
- Reference collections are not supported in conjunction with Oracle backends for this release of Component Broker.
- If your schema uses the Oracle Cache Service, you can import the schema if the columns are of the Oracle VARCHAR2 and NUMBER data types, or any of the IBM DB2 data types (that is, those Oracle data types that have an equivalent type in DB2). Object Builder accepts all SQL/DS and DB2 types and the Oracle NUMBER, NUMBER(p), NUMBER(p,s) and VARCHAR2 types. It will not accept any other Oracle types such as RAW(n), LONG RAW, NCHAR(n), NVARCHAR2, and ROWID. See “Oracle data type mappings” on page 146 for a complete list.
- Object Builder will not accept the Oracle data type NUMBER with a negative scale.

Procedural Adaptor (PA) Beans

Importing PA beans

- Procedural Adaptor Object (PAO) beans that are created with both VisualAge for Java version 3.0 and version 2.0 are supported with this release of Component Broker.
- You **cannot** import PA beans that inherit from the BeanInfo superclass in VisualAge for Java.
- When you create the PA bean, ensure that you name its key the same name as the PA bean, with the suffix Key. For example, if the name of the PA bean is AccountPAO, then the name of its key must be AccountPAOKey. Only such beans can be imported into Object Builder.
- The PA beans that you import cannot have arrays as either attribute types, or method parameter types, or method return types. The only types that are supported are those listed in “Java data type mappings” on page 159.
- You cannot import into Object Builder PA beans that have two methods with the same name. You can create such PA beans with overloaded method names (methods with the same name that differ in either the type or number of their parameters, or both) using VisualAge for Java.

Supported types of the PA bean

Only the Java types *int*, *float*, *double*, *boolean*, *char*, *short*, *byte* and *java.lang.String* are supported as attribute types, push-down method return types and parameter types, and PAA query method types for the PA bean.

Supported types for PA bean push-down methods

When you are creating a data object from a PA persistent object, only push-down methods of the *char* type can be pushed up to the data object.

Java types supported in the imported PA bean

None of the Java data types except the following types are supported in the PA bean that you import into Object Builder:

- *int*
- *float*
- *double*
- *boolean*
- *short*
- *byte*
- *void*
- *char*
- *java.lang.String*

390 OS/390 and procedural adaptors

When you import a procedural adaptor bean, and have OS/390 as the deployment (target) platform (**Platform** > **Constrain** > **390**), only the EXCI, OTMA, and Generic connector types are available for selection (LU 6.2, HOD, SAP, and ECI are not available).

 390 When you use procedural adaptors, you cannot call the resource methods *endResource()*, *checkpointResource()*, and *resetResource()* on the business object when the target platform is OS/390.

390 OS/390 data type restrictions

When one of the constraint platforms is 390 (you select **Platform** > **Constrain** > **390**), the *wchar*, *wstring*, and *long long* types are not available for selection as either attribute types, method return types, or method parameter types for your objects.

390 OS/390 service restrictions

When you define a data object implementation, the Cache Service option is not available when the target platform is OS/390. Oracle and Informix databases are not supported on this platform.

Enterprise beans support

Composers are not supported for enterprise beans that are deployed using Object Builder and the Component Broker MOFW (CORBA) infrastructure even though VisualAge for Java supports both composers and converters.

Some converters are supported only in concept (in the form of its various attribute mapping patterns, which include the specification of arbitrary, user-defined mappings) . The physical VisualAge for Java or VisualAge Persistence (Persistence Builder) converters will not be used when an enterprise bean is deployed using Object Builder. For a list of the default type conversions (the supported converter mappings) that take place during enterprise bean deployment, see Java to Object Builder type mappings.

▶ **390** Enterprise bean support is available on the Windows NT, AIX, Solaris, and HP-UX deployment platforms. It is also available for OS/390. However, Component Broker and WebSphere EJB clients on platforms other than CB OS/390 will not be able to exchange information with CB OS/390 enterprise beans. Neither will CB and WebSphere EJB clients that are on CB OS/390 be able to exchange information with enterprise beans on other platforms.

▶ **390** If the deployment platform is 390, the only backend storage options available are **DB2 V5.2 Embedded SQL**, **DB2 V6.1 Embedded SQL**, **EXCI - Procedural Adaptors** and **OTMA - Procedural Adaptors**.

Redeployment of CMP entity beans

If you are redeploying a CMP entity bean into the same Component Broker model, you cannot make the following changes:

- Delete a container-managed field (You can add container-managed fields.)
- Specify a field that was earlier designated as a key field, to no longer be a key field. (You can assign more fields to be keys.)
- Change a primitive type (for example, int, char, long) to an Object type (for example, Integer, String, java.sql.Date)
- Change an Object type to a primitive type
- Export the redeployed enterprise bean (from VisualAge for Java) to a file with a different name. That is, if the enterprise bean was exported to a JAR file called foo.jar, the redeployed enterprise bean must be exported to a JAR file with the name foo.jar as well.

MQSeries

MQSeries application adaptor support

▶ AIX ▶ 390 You can select only the Windows NT, Solaris and HP-UX platforms for deployment if you want to use the MQSeries application adaptor support (that is, if you want to create applications that send messages to, or receive messages from queues that are managed by MQSeries application adaptors).

Informix

Informix application adaptor support

- ▶ 390 ▶ HP-UX This support is not available on the OS/390 and HP-UX platforms.
- The current version of Object Builder does not support Informix-unique SQL column types.
- ▶ AIX ▶ SOLARIS Informix Dynamic Server version 7.30 files are supported only on AIX and Solaris.
- ▶ WIN ▶ AIX ▶ SOLARIS Informix Dynamic Server version 7.31 files are supported only on Windows NT, AIX and Solaris.

Informix Programming

A given transaction cannot access more than one Informix database per CB server. To involve two Informix databases in a transaction, you must access each database from a different server.

RELATED CONCEPTS

“Troubleshooting” on page 905

RELATED REFERENCES

“Internationalization of data” on page 132

“Naming objects” on page 128

Composition restrictions

One-to-many relationships

The one-to-many relationships of components you add to a composition will not be available in the composition, or in the interface of any composite components based on the composition. One-to-many relationships, and the methods that support them (for example, `addRel`, `removeRel`, `listRel`), will not be included or republished in the composites you create.

Composition include files

The include files for composited components are included automatically. You do **not** need to add them to the Composition File wizard, Include Files page.

The Composition File wizard, Include Files page shows IManagedClient as an include file, even though compositions inherit from IManagedLocal. This include file is required for code generation, and must not be deleted.

DB2 column name limitations

DB2 limits column names to 18 characters. Because of the mapping of attribute names as they are added to a composite, the attribute names may be longer than 18 characters. The attribute names may have to be edited in the Add Persistent Object and Schema wizard to shorten them to less than 18 characters.

Republishing methods

Attributes in the composition should not delegate to methods that throw user exceptions. The code generated will not compile. CORBA attributes do not support exceptions.

Attributes in a composition must not delegate to methods that have “out” or “inout” parameters. It is not possible to republish “out” or “inout” parameters on attributes.

Adding or renaming a managed object after a composite is built

When you add or rename a composited component in the Composition Editor, Composition page (Objects to Composite list), you must follow these steps:

1. Open any business object interfaces that are based on the group, and select **Refresh from Composition**.
2. Update the key, if necessary.

Projects and models

A project provides the directory structure that organizes your work. It can contain any number of components, organized into applications and application families. Each project contains a single model, which can be used to generate code for multiple platforms. When you create a new project, you need to name the model, and also identify any dependencies your work will have on other, existing, projects. The model name you provide will be used to identify the project for team environment builds.

Within the project directory, your work is stored in several subdirectories:

- *project/Model*
Contains the .uni files that Object Builder uses to store your work between sessions, or when you select **File > Save**. Each model directory includes the following files:
 - obp.uni
The project file. Defines project metadata. Defines the models that the

project accesses in read-write and read-only modes. May itself be accessed by other projects in read-write mode (so that projects can exchange dependency information).

At minimum, the project model has a dependency on obframe.uni and obprim.uni, models that define Component Broker framework interfaces and Component Broker primitive elements.

This file should not be deleted, except as part of the entire project directory, or directly edited.

- project.cfg

If you are using the OBModelPath environment variable to manage your dependencies, then this file is not used. If you are **not** using the environment variable, then this file provides an ASCII version of the **depends** and **usedby** relationships this project's model maintains with other projects' models. Generally you should manage cross-project dependencies through the Project Dependencies page of the Open Project wizard, and through the OBMODELPATH environment variable.

- obm.uni

The model file that contains your work. Accessed in read-write mode by the project. Other obm.uni files in other project/Model directories may be accessed in read-only mode.

This file must not be deleted, except as part of the entire project directory; neither must it be directly edited.

If you are using external files to provide the implementations for some methods, these external files are also stored here.

- *project/Working*
Contains the platform subdirectories for generated source files by platform (for example, *project/Working/NT*, *project/Working/AIX*). Source files are generated for the platforms selected on the **Platform** menu of Object Builder. You can generate source files by selecting **Generate > Selected** or **Generate > All** from the pop-up menus of folders or objects in the Tasks and Objects pane, or by using obgen from the command line.
- *project/Export*
Contains any exported model elements, in XML format. You can import these files into other projects, using the **File > Import** menu choice in Object Builder.
- *project/Import*
Contains any XML files that were used by the Rose Bridge to export a Rose model into Object Builder.
- *project/ImportEJB*
Contains the XML file for your model. For example, for the imported EJBHotel.jar file, it will be EJBHotel.xml. You can later import this XML file into Object Builder (for example, for another project).

- *project/XMI*
Contains the file *xmi.xml*, which holds any model information that is not directly translatable between Rose and Object Builder. When you import or export from Rose, this file maintains the extra information that would otherwise be lost in the transfer.

RELATED CONCEPTS

Components (*Programming Guide*)

“Rose” on page 64

“The Rose Bridge” on page 69

“Chapter 10. Team development” on page 443

RELATED TASKS

“Opening a project” on page 26

Developing in Object Builder

Application development in Object Builder consists of defining, building, and packaging components. Components can be defined in any of the following ways:

- From a new design (starting from the component interface and working down to the component datastore)
- From an existing datastore (starting from the datastore and working up to the component interface)
- From both (combining a new component and data interface with an existing datastore)

Each server application can consist of one or more components, each of which could be developed in a different way.

The main development tasks are as follows:

1. “Chapter 3. Using Rational Rose with Object Builder” on page 63
2. “Chapter 4. Creating a component” on page 127
3. “Defining relationships” on page 278
4. “Creating a child component” on page 306
5. “Configuring builds” on page 549
6. “Packaging applications” on page 574
7. “Chapter 13. Testing applications with QuickTest” on page 611
8. “Developing as part of a team” on page 444
9. “Chapter 11. Customizing Object Builder” on page 503
10. “Using Object Builder from the command line” on page 659
11. Working with external files

RELATED CONCEPTS

“Object Builder” on page 1
 Components (*Programming Guide*)

RELATED TASKS

“Chapter 2. Tutorials for Object Builder” on page 39
 “Working with components” on page 697

RELATED REFERENCES

“Internationalization of data” on page 132
 “Naming objects” on page 128

Platform-specific information

Object Builder organizes its generated files into platform-specific directories, some of them with multiple subdirectories as shown in these tables.

Generating code for Windows NT:

Subdirectory that contains the generated code	<project>\Working\NT
Corresponding nmake command	nmake -f <installation directory>\Working\NT\all.mak
Location of generated DDL files**	Working\NT\<application family name>\<build style>
Client C++ Library Location***	Working\NT\<build style>
CBConnector installation library directory	%SOMCBASE%\data\ntApps3.5

Generating code for AIX:

Subdirectory that contains the generated code	<project>\Working\AIX
Corresponding nmake command	make -f <installation directory>\Working\AIX\all.mak
Location of generated DDL files**	Working\AIX\<application family name>\<build style>
Client C++ Library Location***	Working\AIX\<build style>
CBConnector installation library directory	/var/CBConnector/data/aixApps3.5

Generating code for OS/390:



Subdirectory that contains the generated code	<project>\Working\390
Corresponding nmake command	POSIX make with some extensions*
Location of generated DDL files**	Working/390/<application family name>/<build style>
Client C++ Library Location***	Working\390\<build style>
CBConnector installation library directory	DLLs generated from the build are automatically copied to the user- defined data set in the second level system.****

Generating code for Solaris:

Subdirectory that contains the generated code	<project>\Working\Solaris
Corresponding nmake command	POSIX make with some extensions*
Location of generated DDL files**	Working/Solaris/<application family name>/<build style>
Client C++ Library Location***	Working\Solaris\<build style>
CBConnector installation library directory	/var/CBConnector/data/solarisApps3.5

Generating code for HP-UX


Subdirectory that contains the generated code	<project>\Working\HPUX
Corresponding nmake command	POSIX make with some extensions*
Location of generated DDL files**	Working/HPUX/<application family name>/<build style>
Client C++ Library Location***	Working\HPUX\<build style>
CBConnector installation library directory	/var/ CBConnector/data/hpuxApps3.0

*    Extensions for the POSIX make command on AIX, Solaris and HP-UX

The extensions we use on the UNIX platforms are an include directive that allows us to create a makefile, which includes text from an external file. For example, if the file XXX.mak has the following commands:

A = 1

include \$IVB_DRIVER_PATH/bin/obadll35.mk
when the make tool processes XXX.mk, it will expand
\$IVB_DRIVER_PATH/bin/obadll35.mk with the contents of the actual file.

*  390 The make command on OS/390 (equivalent to the POSIX make command with significant extensions)
The make command on OS/390 is used along with metarules.

For example, given the makefile recipe:

```
a%y.o : %c
```

...

the make utility will use the rule to match targets and dependencies. The '%' character is a wild card.

If we were to use the above metarule to build the following targets:

```
aby.o
```

```
a123y.o
```

we would need the following dependencies:

```
b.c
```

```
123.c
```

The '%' character must match on both sides of the target-dependency relationship. These extensions come standard with the OS/390 make tool.

**** Note the following points:**


- <build style> is each of NOOPT, PRODUCTION, TRACE, TRACE_DEBUG (that is, for a given platform, you get four sets of DDL files generated).
- A subdirectory with the name of the application family is created beneath the <build style> subdirectory.
- The DDL file is generated along with a backup version. For example, if the name of your application family is MyAppFam, and you have specified the name of the DDL file as AppFam, the DDL emitter will generate the following files:

```
Working\platform\<build style>\MyAppFam\AppFam.ddl
```

```
Working\platform\<build style>\MyAppFam\backup\AppFam.auto.ddl
```

***** Note the following points:**

- The C++ client library contains the DLL files that need to be copied into the same directory as the client C++ code, for building the client.
- <build style> is one of the configuration directories: NOOPT, PRODUCTION, TRACE, TRACE_DEBUG. See Build targets.
- Client binding JAR files are found in Working/<platform>/<build style>/JCB/.

***  The OS/390 run time has two types of systems:

- **the first level system:** a common direct access storage device (DASD) that everyone shares. Some of the common information is stored here.
- **the second level system:** a personal account that you can configure according to your needs. For example, user A could be using CB R3.0, and user B could be using CB R3.5.

C++ compilers that can be used:

See the Component Broker documentation in the *Planning, Performance and Installation Guides* for the corresponding platform:

 Table 1. Supported Prerequisites for Windows NT

 Table 1. Supported Prerequisites for IBM AIX

 Table 1. Supported Prerequisites for Solaris

 Table 1. Supported Prerequisites for HP-UX

RELATED CONCEPTS

“Multi-platform development” on page 419

“Cross-platform development” on page 426

“Remote build” on page 570

Application DDL files

RELATED TASKS

“Generating code” on page 551

“Launching a remote OS/390 build” on page 572

“Opening a project” on page 26

“Building the DLLs” on page 558

“Building the JAR files” on page 561

“Building DLLs in a team environment” on page 487

RELATED REFERENCES

“Platform differences” on page 425

“Build targets” on page 567

OLT and Debug API support



The managed object code emitted by Object Builder for all the platforms has been adjusted to exploit a new set of Object-Level Trace (OLT) and Debug application programming interfaces (APIs).

RELATED CONCEPTS

Business object (*Programming Guide*)

Managed object (*Programming Guide*)

RELATED TASKS

“Working with methods” on page 750

“Chapter 8. Working with enterprise beans” on page 391

Setting up Object Builder

There are several tasks you need to perform before you can take advantage of the full range of Object Builder’s functionality. At minimum, you will need to start a project and set Object Builder preferences.

Complete the following tasks to set up tools, enable and customize function, and start working in Object Builder on either a new or an existing project:

1. “Setting up Rose 98” on page 65
2. “Setting up Rose 98i and Rose 2000” on page 66
3. “Opening a project” on page 26
4. “Migrating old projects” on page 33
5. “Setting Object Builder preferences” on page 27
6. “Searching the Tasks and Objects pane” on page 30
7. “Checking a model for consistency” on page 31

RELATED CONCEPTS

“Rose” on page 64

“Object Builder” on page 1

RELATED TASKS

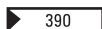
“Chapter 2. Tutorials for Object Builder” on page 39

“Developing in Object Builder” on page 19

Object Builder environment variables



The values of Object Builder environment variables that are specified in make files take precedence over external settings for the same variables.



IVB_BATCH_PROCESS_FACTOR

This variable is found in the file `prjdefs.mk`, which is generated during the build process. It denotes the maximum number of submakes that can occur. It is generated with a default value of 1, indicating that the build will be serialized (not parallel). The value that is assigned to this variable within `prjdefs.mk` overwrites the value that you assign to it outside of the file. To

override the default value of 1, you would have to either edit `prjdefs.mk`, or pass this variable as a parameter with the make command.

IVB_HOME

Represents the directory where tool-state settings are saved (These settings are saved in the files `panes.ini` and `Workbook.ini`).

To look at these files, use the command:

 `cd /D %USERPROFILE%`

    `cd $HOME`

Note the following points:

- The `/D` option is required on Windows NT because it supports drive letters.
- If this directory is not accessible, you can edit the `ob.bat` file, and change the definition for `IVB_HOME` (change the directory path that is passed along with the `-D` option to the Java Virtual Machine).

Warning: Do not change `%USERPROFILE%` instead as this may be used by other programs and Windows NT services.

OBModelPath

Enables you to relink linked models when either drive letters or directories change.

EJBOBTEMP

A temporary directory used for EJB work. It is set to the system `%tmp%` directory, but if no system `%TMP%` is defined, it will be set to the current directory `“.”`. Temporary output files are stored in this directory when you deploy enterprise beans using Object Builder.

  On AIX and Solaris, the temporary directory is `/tmp`.

RELATED CONCEPTS

“Object Builder” on page 1

RELATED TASKS


“Deploying enterprise beans using Object Builder” on page 410

RELATED REFERENCES



“make options” on page 688

Opening a project

Starting Object Builder

 To start Object Builder from the Windows NT desktop, follow these steps:

1. Select **Start > Programs > IBM Component Broker > Object Builder**. The Open Project wizard opens to the Specify Project Directory page.
2. Type a name for the project, or specify an existing one.

  To start Object Builder on AIX (or on Windows NT from a command prompt), type the command

```
ob <project_directory>
```

You cannot specify a directory name that contains spaces.



A project directory has a model subdirectory (called Model), where Object Builder stores an internal model of your work. The project directory also has a working directory (called Working), where Object Builder generates the code (such as .idl and .cpp files) for your work.

Creating a new project

To create a new project, follow these steps:

1. Specify a new project name on the Specify Project Directory page.
2. Object Builder prompts you to provide a model name. It provides a default model name, which it assumes is the same as the directory name for the project.
3. Either accept the default name, or change it. The name you specify will be used to identify the project in a team environment, regardless of any changes in directory structure.
4. Click **OK**.

Note: If you are working with a large project (more than 30 components), you may need to increase the maximum heap size of the Java virtual machine. You can do so by editing the ob.bat file:

1. Make sure Object Builder is closed.
2.  Edit <CBroker>\bin\ob.bat
3.  Edit <CBroker>/bin/ob
4. Change the parameter -mx255m, increasing the number by 5m for each additional component in your project (this number is approximate, and assumes components of average complexity).

For example, if your project contains 100 components, change the parameter to `-mx605m` (70 additional components multiplied by 5m each, plus the original 255m).

5. Start Object Builder. The new parameter is used, and the maximum size of the Java virtual machine is increased.

Warning: Do not alter the contents of the directories: `<CBroker>\obprime` and `<CBroker>\obframe`. These directories contain definitions for IDL primitive types and for the Component Broker frameworks, both of which are used by the project models you create in Object Builder.

RELATED CONCEPTS

- “Object Builder” on page 1
- “Projects and models” on page 17

RELATED TASKS

- “Migrating old projects” on page 33

RELATED REFERENCES

- “Platform-specific information” on page 20

Setting Object Builder preferences

You can customize the appearance and behavior of Object Builder using the Preferences notebook. To access the notebook and set preferences for Object Builder, follow these steps:

1. Click **File > Preferences**. The Preferences notebook opens.
2. Click on a folder or node in the tree view on the left. The General folder organizes general settings for Object Builder’s appearance and behavior. The other folders organize specific settings for the different panes in Object Builder, or for the activities available through that pane (for example, the preferences for the Tasks and Objects pane include a page for Makefile Generation settings).
3. Select a node. The settings for the node appear on the right.
4. Specify the settings you want.
Click **Help** for a description of the settings for the currently selected node.
5. Click **OK** to apply your settings and return to Object Builder.

RELATED CONCEPTS

- “Object Builder” on page 1

RELATED TASKS

- “Setting up Object Builder” on page 24

RELATED REFERENCES

Color Assignments
Indentation (Source page)

Color Assignments

▶ CHANGED

Use this section of the Text Style page to specify the mapping of color to code elements in Object Builder's Source pane. You can select a category, then select the color to assign to it. This section has the following controls:

- Enable
- Categories
- Color Assignment

Enable

Click this option to enable the color assignments shown under **Categories**. When this option is **not** selected, the color assignments are ignored, and all categories are shown in the system-defined foreground color (usually black) text against the system-defined background color. By default, the color assignments are used with all parsable code (method bodies in C++ or Java, and Object Builder-generated source files in Java, C++, or IDL).

Categories

Select the category you want to assign a color to. Each category is displayed in its currently assigned color against the system-defined background. You can select from the following categories:

- Base text
- Comments
- Constants
- Errors
- Keywords

Color Assignment

You can set the colors using the following pages:

- **Swatches**

On the Swatches page, you can select a color from the provided palette. To see the RGB values for a color, hover the mouse pointer over the color. When you select a color, it is added to the Recent palette for easy future retrieval.

- **HSB**

On the HSB page, you can customize color in the following ways:

- Moving the scroll box up and down on the color column. This selects the hue.
- Clicking a point on the square
This selects the color's shade. The shade is measured in terms of

saturation and brightness. The bottom right corner of the square has the least saturation and brightness percentages (0). The upper left corner of the square has the maximum percentage values for both saturation and brightness values. The upper right corner has a saturation percentage of 0, and the maximum brightness. The lower left corner has 100 % saturation, and minimum brightness.

Alternatively, you can enter values for the Hue (**H**), Saturation (**S**), and Brightness (**B**), or Red (**R**), Green (**G**), and Blue (**B**).

- **RGB**

On the RGB page, the values of the red, green, and blue components of the color are set according to your selections on the HSB page. You can further fine-tune the settings.

Preview

You can see the result of your color selections in the Preview box, as you make the selections.

You can use the following buttons after you have made your color selections:

- **Apply**

Click this button to apply your selections to the current category.

- **Restore Default**

Click this button to restore the category to its default color.

RELATED TASKS

“Setting Object Builder preferences” on page 27

“Implementing methods” on page 752

Find and Replace (Source page)

Use this section of the Source page to select the behavior you want when finding and replacing text in the Source pane:

- Beep when no match is found
- Show message when no match is found

Beep when no match is found

When a search fails to find a match, your computer beeps.

Show message when no match is found

When a search fails to find a match, a message appears (silently) that tells you so.

RELATED TASKS

“Setting Object Builder preferences” on page 27

“Implementing methods” on page 752

Indentation (Source page)

Use this section of the Source page to select the indentation behavior you want in the Source pane:

- New lines are not indented
- New lines maintain existing indentation
- Parse and indent automatically

New lines are not indented

When you press Enter while editing, the cursor goes to the beginning (far left) of the next line.

New lines maintain existing indentation

When you press Enter while editing, the cursor goes to the next line, indented by the same amount as any text on the previous line.

Parse and indent automatically

As you enter code, it is parsed, and indented accordingly.

RELATED TASKS

“Setting Object Builder preferences” on page 27

“Implementing methods” on page 752

Searching the Tasks and Objects pane

When you are working with a large or complex application, it can be difficult to locate a particular component object or element of the application in Object Builder. To find a particular item in the Tasks and Objects pane, follow these steps:

1. Select **Edit > Find**. The Find dialog box opens.
2. Type the name of the item in the **Find Next** field.
3. Set any search options you want in effect:
 - **Match case**
Only finds items with names that have the same capitalization as the name you specified.
 - **Whole word**
Only finds items with the exact name you specified. Does not find items whose names merely include the string you specified.
 - **Wrap**
Searches the entire pane. If you do not select **Wrap**, the search only occurs from the currently selected item to the bottom of the pane.
4. Click **Find Next**.

The first item with a matching name is selected in the Tasks and Objects pane. The tree view is expanded as necessary to show the item.

5. When you are finished searching, click **Cancel** in the Find dialog box.

The **Find** function will not find any items that do not appear in the Tasks and Objects pane. If you have applied a filter to the pane, the search will not find items that are excluded from the pane by the filter.

RELATED CONCEPTS

“Object Builder” on page 1

RELATED TASKS

“Filtering the Tasks and Objects pane” on page 537

RELATED REFERENCES

“Naming objects” on page 128

Checking a model for consistency

While Object Builder enforces consistency on your model, there are circumstances in which the model can become internally inconsistent. If you are experiencing consistency problems with your generated code (for example, type mismatches between an attribute and its referenced interface), run the consistency checker to diagnose the problem, and generate a report on the state of your model.

To check a model for consistency, follow these steps:

1. Open the project whose model you want to check (select **File > Open New Project**). The project opens, and the project model is loaded into Object Builder.
2. From Object Builder’s menu bar, click **File > Check Model**.
The consistency checker dialog opens.
3. Select the types of consistency problems you want to check for.
4. Click **Run**. The consistency check runs, and its output is displayed in a report window.
5. Review the report.

You can save the report for later review by clicking **Save**.

Each error, warning, or information message includes the file, module, object type, and object name to which the message applies.

6. Click **OK** to close the report window and return to Object Builder.

Alternatively, you can use the command-line tool `obcheck` to check a model’s consistency.

RELATED CONCEPTS

“Projects and models” on page 17

“Troubleshooting” on page 905

RELATED REFERENCES

“obcheck” on page 694

Consistency checker errors (*Problem Determination Guide*)

Requirements for Java development

► CHANGED

To develop Java business objects, you need the Java 2 SDK, Standard Edition, Version 1.2.

► WIN Set up for NT

The configuration to support Java business objects includes updating the PATH environment variable.

1. Logon to Windows NT as a user with administrator privileges.
2. To use the Java 2 SDK, you must update the PATH for your Component Broker user ID. For example, if the Java 2 SDK is installed in the c:\jdk1.2.2 directory, update the environment variable PATH:
PATH=c:\jdk1.2.2\bin;c:\jdk1.2.2\jre\bin;c:\jdk1.2.2\jre\bin\classic
3. Stop the CB Server service.
4. Restart the CB Server service.
5. Reactivate the configuration using the System Manager User Interface.

► AIX Set up for AIX

If the path does not already exist, you may need to add a path to libjava.a.

For more details on setting up the Java 2 SDK, see the chapter ‘Installing and configuring the Java 2 SDK’ in the *Planning, Performance and Installation Guides* for the relevant platform.

RELATED CONCEPTS

Programming languages and conventions (*Programming Guide*)

Installing and configuring the Java 2 SDK (*Planning, Performance and Installation Guides for Windows NT*)

Installing and configuring the Java 2 SDK (*Planning, Performance and Installation Guides for AIX*)

Installing and configuring the Java 2 SDK (*Planning, Performance and Installation Guides for Solaris*)

Migrating projects from 3.0

You can migrate projects individually as needed, simply by opening the old project with the 3.5 version of Object Builder. This is sufficient for stand-alone projects, but can be time-consuming for team environments. To migrate an entire team environment at once, use the command-line utility “obmigrate” on page 659. The obmigrate utility can be used to import stand-alone projects, or entire sets of interdependent projects in a single step.

Once you have migrated a project (either by opening it and saving it in the 3.5 version of Object Builder, or using the obmigrate utility), it can no longer be worked with in Object Builder 3.0.

There is no direct way to migrate from releases earlier than 3.0 to 3.5. You can only migrate through immediately subsequent versions, until you reach 3.5 (for example: from 1.2 to 1.3 to 2.0 to 3.0 to 3.5).

For information on migrating existing NT or AIX projects to OS/390 targets, consult the *OS/390 Component Broker Planning and Installation Guide*.

The following tasks cover migration of 3.0 projects to the 3.5 format:

1. “Migrating old projects”
2. “Migrating from the command line” on page 35
3. “Migrating a team environment” on page 36

RELATED CONCEPTS

“Projects and models” on page 17

RELATED TASKS

“Setting up Object Builder” on page 24

“Setting Object Builder preferences” on page 27

“Opening a project” on page 26

RELATED REFERENCES

“obmigrate” on page 659

Migrating old projects

You can migrate projects individually as needed, simply by opening the old project with the 3.5 version of Object Builder. This is sufficient for stand-alone projects, but can be time-consuming for team environments. To migrate an entire team environment at once, use the utility “obmigrate” on page 659.

For information on migrating existing Windows NT or AIX projects to OS/390 targets, consult the *OS/390 Component Broker Planning and Installation Guide*.

To migrate a single project from 3.0 to 3.5, follow these steps:

1. Start Object Builder.
2. Specify the location of your old project (for example, Cbroker\my3.0Project) on the Specify Project Directory page of the Open Project wizard.
3. Click **Finish**.
4. Save the project. The Model Conversion dialog opens.
You have several options in how you convert the model:
 - By default, the model is saved in the 3.5 format, replacing the old version.
 - If you select the **Compress 3.5 version** option, Object Builder will save the model in compressed form. This may take some time, but can result in considerable space savings.
 - If you select the **Retain 3.0 version** option, Object Builder will create a backup version of the old model, in the directory \Model30, before saving the model in the new format.

5. Click **Save**.

The saved project is now updated to the 3.5 version. The XML files that are associated with the project are also migrated.

If you open a 3.0 project as a project dependency from a 3.5 project (by selecting it on the Project Dependencies page of the Open Project wizard), then it remains at the 3.0 level, until you open and save it directly. Use the obmigrate utility to migrate a project and all its dependencies in a single step.

IR file name migration

During migration, if Object Builder detects a managed object IR file name whose length is less than or equal to eight characters, it migrates this file name as the business object interface's IR file name property. It can be viewed in the business object's File Properties wizard.

Metadata migration

When you migrate an Object Builder version 3.0 model to the current version (3.5), the following metadata is migrated by default:

- the data object implementation's discriminator predicate with a style of None
- the configured managed object's disambiguated supertype, the default being either none, or the first one in the list
(The default is none if there are no parents in the list. The default is the first one in the list if there are two or more parents, and for new configured managed objects, or for those whose previous choice is no longer valid due to the deletion of the chosen configured managed object from the application family.)

- Models from older versions of Object Builder that have distributed queries will use the path expression form of the query in the list() method, and will have the default factory names and factory finder names.

RELATED CONCEPTS

“Projects and models” on page 17

RELATED TASKS

“Setting Object Builder preferences” on page 27

“Opening a project” on page 26

“Migrating from the command line”

“Migrating a team environment” on page 36

RELATED REFERENCES

“obmigrate” on page 659

Migrating from the command line

►CHANGED

You can migrate your 3.0 projects to the 3.5 version using the command-line utility “obmigrate” on page 659. The command will either migrate the specified projects alone, or their project dependencies as well (only if it is used with the with the -all option).

Generally, if you have project dependencies, you should migrate all the projects at once (using the -all option), to save time. However, if your project has no dependencies, or you want to migrate the projects it depends on selectively, you can omit the -all option.

To migrate only specified projects (without migrating all the projects they depend on), use the obmigrate command:

1. On the command line, type: `obmigrate project1 project2 ...projectn`

If the tool encounters a project dependency in the course of the migration, you are asked how it should be handled. You have the following options:

- **Y**
Migrate the project being depended on. Prompt again if further dependencies are encountered.
- **A**
Migrate the project being depended on, and migrate any further dependencies without prompting.
- **n**
Do *not* migrate the project being depended on. Prompt again if further dependencies are encountered.
- **N**
Do *not* migrate the project being depended on, and do *not* migrate any further dependencies. Do not prompt again.

2. Once the migration is complete, check the migration log (obmigrate.log), in the directory you invoked obmigrate from.

The log consists of the following information:

- The date and time
- A list of read-only projects that could not be migrated
- A list of dependent projects that were not migrated.
- A list of dependent projects that could not be found
- A list of projects that could not be found (generally because they were in a directory not listed by ObModelpath).

Generally, if you want to migrate all project dependencies without prompting (option **A**), you should call the obmigrate command with the -all option. For more information on using the -all option, see the task “Migrating a team environment”.

Once migration is complete, the model for each project has been migrated to the 3.5 version, in compressed format. The XML files that are associated with each project are also migrated. The old model is preserved in a backup directory (*project*\Model30).

If you migrate old projects by opening and saving them in Object Builder, instead of from the command line, you have the option of saving in uncompressed format, and the option *not* to create a backup model.

RELATED CONCEPTS

“Projects and models” on page 17

RELATED TASKS

“Migrating old projects” on page 33

“Migrating a team environment”

RELATED REFERENCES

“obmigrate” on page 659

Migrating a team environment



You can migrate an entire team environment from 3.0 to 3.5 in a single step using the command-line utility “obmigrate” on page 659 with the -all option. The command will migrate specified projects, as well as all projects that are dependencies, in a single step.

To migrate all the projects in a team environment, follow these steps:

1. Identify or create a project that lists all other projects as dependencies. Typically your environment will include an integration project (for building and packaging) that meets this requirement.
2. Make sure all projects are checked out (in read-write mode, and locked to prevent others from making simultaneous changes).
3. Make sure your search path for project dependencies is set correctly (either in the OBMODELPATH environment variable, or in the currently selected Project Search Path in Object Builder's Open Project wizard). The search path should include all the directories that contain projects in your environment.

4. On the command line, type:

```
obmigrate -all myIntegrationProject
```

For example:

```
obmigrate -all e:\projects\Integration
```

migrates the integration project, and migrates project dependencies recursively. For example, \Integration depends directly on \A, which depends in turn on \B, which depends in turn on \C. Even though \Integration only depends directly on \A, it depends indirectly on \B and \C as well, so those projects are also migrated.

If your team environment includes projects that are not listed in your integration project (directly or indirectly), you can include these projects on the command line as well.

For example:

```
obmigrate -all e:\projects\Integration e:\projects\Standalone
```

migrates the integration project with its dependencies, and the standalone project with its dependencies.

5. Once the migration is complete, check the migration log (obmigrate.log), in the directory you invoked obmigrate from.

The log consists of the following information:

- The date and time
 - A list of read-only projects that could not be migrated
 - A list of dependent projects that were not migrated (generally because you did not specify -all).
 - A list of dependent projects that could not be found (generally because they were not checked out)
 - A list of projects that could not be found (generally because they were in a directory not listed by OBMODELPATH).
6. Check the projects back in to your change control system.

The model for each project has been migrated to the 3.5 version, in compressed format. The XML files that are associated with the projects are also migrated. The old model is preserved in a backup directory (*project\Model30*).

If you migrate old projects by opening and saving them in Object Builder, instead of from the command line, you have the option of saving in uncompressed format, and the option *not* to create a backup model.

RELATED CONCEPTS

“Projects and models” on page 17

“Change control” on page 463

RELATED TASKS

“Migrating projects from 3.0” on page 33

“Setting up a team environment” on page 457

“Maintaining a team environment ” on page 490

“Opening a project” on page 26

RELATED REFERENCES

“obmigrate” on page 659

Chapter 2. Tutorials for Object Builder

The following tutorials cover some development scenarios in Object Builder. These act as extended examples and introductions to Object Builder's functionalities.

You can select from the following Object Builder tutorials:

- "Tutorial: Creating a component with transient data"
- "Tutorial: Creating a component for new DB data" on page 50
- "Additional tutorials" on page 62

RELATED CONCEPTS

"Object Builder" on page 1

RELATED TASKS

"Developing in Object Builder" on page 19

Tutorial: Creating a component with transient data

Objectives

To create a simple component with transient data, locatable by primary key.



To generate the code for the component.

To build the DLLs for the component.

To define the application configuration information for the component.

Before you begin

You need the following installed on your system:

- A CB Server
- A CB Client
- CB tools, including Samples
- DB2 Universal Database[®]
- DB2 SDK
-  VisualAge for C++ and (for Java applications) VisualAge for Java
-  IBM C and C++ Compilers for AIX V3.6.4 and (for Java applications) VisualAge for Java

For a complete list of prerequisites that you must install on your system, see the *Getting Started with Component Broker* documentation for the appropriate platform.

You should be familiar with the Component Broker programming model, as described in the *IBM Component Broker Programming Guide*. At minimum, you should have read chapter 1, the introduction to the programming model.

Sample files

There are equivalent samples for this exercise. The samples include:

- A BusinessObjects project that contains the finished Object Builder model
- A zipped Rational Rose model that contains definitions for the business object, key, copy helper, and data object interface
- Documentation for the sample, including instructions on running a Java client for the sample



   The samples are in the following directories (under <CBroker>, the install directory):

For C++:

```
samples\Tutorial\Fundamentals\transient\BusinessObjects
samples\Tutorial\Fundamentals\transient\Rose\TransientObjectRose.zip
samples\Tutorial\Fundamentals\transient\Docs\Transient.html
```

For Java:

```
samples\Tutorial\Fundamentals\jtransient\BusinessObjects
samples\Tutorial\Fundamentals\jtransient\Rose\JTransientRose.zip
samples\Tutorial\Fundamentals\jtransient\Docs\JTransient.html
```

  Samples are shipped with the CB Toolkit; not Component Broker. On Solaris and HP-UX, Component Broker and the Toolkit are installed in different directories.

Description

This exercise describes how to create a simple component in Object Builder, without persistent data. The component has a key and copy helper, that allow client programs to locate and create instances of the component on the server.

The following tasks do not give explicit instructions for every step, but should at least get you into the right wizards. If you are experiencing problems, click the Help button within a wizard, or go to the Help pulldown in Object Builder.

For this exercise, you will complete the following tasks:

1. Creating the project
2. Creating a business object interface

3. Adding a key and copy helper
4. Adding a business object implementation
5. Adding a data object implementation
6. Adding a managed object
7. Generating the code
8. Defining a client DLL and server DLL
9. Defining an application family and application
10. Configuring the component with the application

Creating the project

Create a sample project to hold your work. For example, e:\tutorials\transient

1. Start Object Builder.
2. In the Open Project wizard, type a name and path for the project directory.
3. Click **Finish**.
4. When asked whether you want to create a new project, click **Yes**.

Creating the business object interface

Define a business object file (sample6):

1. From the User-Defined Business Objects folder's pop-up menu, click **Add File** to open the Business Object File wizard.
2. Name the file.
3. Click **Finish**. The file now appears under the folder.

Add a module (sample6):

1. From the pop-up menu of the file, click **Add Module** to open the Business Object Module wizard.
2. Name the module.
3. Click **Finish**. The module now appears under the file.

Add an interface (csAgent). The business object interface defines the interface for the whole component. It will be contained in the sample6 module.

1. From the pop-up menu of the module, click **Add Interface** to open the Business Object Interface wizard.
2. Name the interface csAgent.
3. Click the page title and turn to the Attributes page.
4. Add the following attributes:
 - readonly float commissions
 - float commPercent
 - float pendingPaycheck
 - string agentName
 - readonly long id

5. Set the size of agentName to 100. You should always provide a size for string attributes.
6. Click **Next** and turn to the Methods page.
7. Add the following method:
 - void payCommission (in float amount)
8. Click **Finish**.

The csAgent interface now appears under the sample6 module. The attributes and methods appear in the Methods pane, when the interface is selected.

The attributes are represented as paired get and set methods, except for the id attribute, which was defined as read-only, and therefore only has a get method.

The interface does not have any business logic associated with it. The implementation of the interface is defined separately, in the business object implementation.

Adding a key and copy helper

Add a key (csAgentKey). The key allows client applications to locate or create instances of the component on the server. It consists of an attribute or attributes of the business object interface that uniquely identify an instance of the component. In this case, the csAgent id attribute is the appropriate choice.

1. From the interface's pop-up menu, click **Add Key** to open the Key wizard.
2. Accept the default name; select the id attribute and add it to the **Key Attributes** list.
3. Click **Finish**.

Even though the id attribute of the business object interface is read-only, the id attribute of the key is both readable and writable. The client application can set the value of the id on the key, and use it either to initialize a new instance of the component, or to locate an existing instance of the component, on the server.

Add a copy helper (csAgentCopy). The copy helper allows client applications to create and initialize instances of the component on the server, using one call to set numerous attributes, rather than one call per attribute.

1. From the interface's pop-up menu, click **Add Copy Helper** to open the Copy Helper wizard.
2. Accept the default name; select all listed attributes and add them to the **Copy Helper Attributes** list.
3. Click **Finish**.

csAgentCopy appears under csAgent.

Adding a business object implementation and data object interface

Add a business object implementation (csAgentBO) and data object interface (csAgentDO). The business object implementation contains the actual business logic of the component, including the method implementations. Any state data attributes (those attributes that cannot be deduced or derived from other attributes) become part of the data object interface. The separation of business logic (in the business object) from state data (in the data object) allows issues such as data persistence and integrity to be partitioned from the rest of the business logic.

1. From the interface's pop-up menu, click **Add Implementation** to open the Business Object Implementation wizard.
2. Change the **Pattern for Handling State Data** to **Delegating**. This is easier to debug than the **Caching** pattern.
3. Click the page title and turn to the Implementation Language page.
4. Select the language you want the business object to be implemented in (**C++** or **Java**).
5. Click the page title and turn to the Key and Copy Helper page. The appropriate key and copy helper are already selected.
6. Click the page title and turn to the Data Object Interface page.
7. Select all attributes and add them to the **State Data** list (to be preserved in the data object).
8. Click **Finish**.

csAgentBO appears under csAgent, and csAgentDO appears under csAgentBO.

Adding a method implementation

Type in the implementation for the method payCommission. The appropriate implementation logic for the data get and set methods, and for framework methods required by the programming model, are calculated for you by Object Builder. You only need to provide implementations for methods you explicitly defined.

1. Click on the business object implementation. The Methods pane shows the user-defined method payCommission, and the various user-defined attributes (in the form of paired get and set methods).
2. Click on long id(). The id attribute only has a get method, because it is read-only, as defined in the business object interface. The provided implementation appears in the Source pane.
3. Review the provided implementation for long id(). The get method for the id attribute delegates directly to its equivalent attribute in the data object, as defined by the **Delegating** pattern chosen in the business object implementation.

4. In the Methods pane, click on the `payCommission` method: `void payCommission (in float amount)`. The signature for the method appears in the source pane, based on the definition you provided in the business object interface and the language you selected in the business object implementation. The method does not have an implementation yet. You must provide the implementation for user-defined methods.
5. In the Source pane, provide the following implementation for `payCommission`:

C++

```
float tmp;
tmp = amount * iDataObject->commPercent();
iDataObject->commissions(tmp);
iDataObject->pendingPaycheck(iDataObject->pendingPaycheck() + tmp);
```

Java

```
float tmp;
tmp = amount * iDataObject.commPercent();
iDataObject.commissions(tmp);
iDataObject.pendingPaycheck(iDataObject.pendingPaycheck() + tmp);
```

You can continue to the next step. When you click on other objects, the implementation will disappear from the Source pane, but the code you typed is now part of the project model, and will be generated as part of the source code for the component (in the file `sample6B0_I.cpp` for a C++ business object, or `_csAgentB0Base.java` for a Java business object).

Adding a data object implementation

Add a data object implementation (`csAgentDOImpl`). The data object implementation defines the way in which you want to handle the component's state data.

1. From `csAgentDO`'s pop-up menu, click **Add Implementation** to open the Data Object Implementation wizard.
2. Accept the default name and platform settings, and click **Next** to turn to the Behavior page.
3. Set the following patterns:
 - **Environment - BOIM with any key**
The component will be locatable by its key (instead of being locatable by a UUID).
 - **Type of Persistence - Transient**
The component's data will not have persistence beyond the lifespan of the component instance. As a result, the component does not require a persistent object (which would manage the mapping of the data to a persistent datastore, such as a database).

- **Data Access Pattern - Local copy**

This is the only option available for a transient data object. There is no persistent datastore to delegate to.

4. Click the page title and turn to the Key and Copy Helper page. `csAgentKey` and `csAgentCopy` should already be selected.
5. Click **Finish**.

`csAgentDOImpl` appears under `csAgentDO`.

Adding a managed object

Add a managed object (`csAgentMO`). The managed object mediates the interaction between the client application and the component, or between other components and this component. It exposes the business object interface to the client application and other components, and accesses any relevant services before and after a call.

1. From the business object implementation's pop-up menu, click **Add Managed Object** to open the Managed Object wizard to the Name and Service page.
2. Accept all the defaults and click **Finish**.
3. Click **File > Save** to save your work before continuing to the next step.

Generating the code

You are now ready to generate the code for the objects you have defined. Generated code will appear in your project's `\Working\platform` directory (for example, `e:\tutorials\transient\working\NT`)

1. From the pop-up menu of the User-Defined Business Objects folder, click **Generate > All**.

The code generation can take several minutes.

2. Review the contents of the `Working\platform` directory. All the source files for the component have been generated, and you can now define how to build them.

While you can view the generated source files for a particular object (by selecting **View Source** from its pop-up menu), you cannot edit the source files through Object Builder. If you edit them outside of Object Builder, you should restrict your changes to method implementations, and import your changes back into Object Builder before re-generating.

Defining a client DLL and server DLL

Configure the build process that will create the client and server DLLs.

Define the client DLL configuration and client DLL or library file (for this exercise, name them both `sample6c`). The client DLL provides client applications with access to the component on the server, using the key and copy helper. You must also include the business object interface, which defines the methods and attributes of the component that the client can access.

1. From the pop-up menu of the Build Configuration folder, click **Add Client DLL** to open the Client DLL wizard to the Name and Options page.
2. Name the configuration. This is the name that uniquely identifies the configuration node.
3. Name the library as well. This is the name for the makefile and for the resulting DLL file. You can create multiple configuration nodes that produce different versions of the same DLL. For example, you could set different compile and link options to produce a development version and a deployment version of the same DLL.
4. For this exercise, the default configuration options are sufficient.
5. Click the title and turn to the Client Source Files page.
6. Select all the client files and add them to the **Items Chosen** list.
7. Click **Finish**.

The sample6c DLL configuration appears under the Build Configuration folder.

Define the server DLL configuration and server DLL file (for this exercise, name them both sample6s). The server DLL is installed on the server to deploy the component, making it available for access by client applications and other components.

1. From the pop-up menu of the Build Configuration folder, click **Add Server DLL** to open the Server DLL wizard.
2. Name the configuration and target DLL file.
3. Click **Next** to turn to the Libraries to Link With page.
4. Add the client DLL file. The server DLL needs access to the client DLL because the client DLL defines the component's interface, and the component implementation inherits from it.
5. Click **Next** to turn to the Server Source Files page.
6. Select all the server source files for the component, and add them to the **Items Chosen** list.
7. Click **Finish**.

The sample6s DLL configuration appears under the Build Configuration folder.

Building the DLLs

To generate the makefiles, based on the configuration choices you made in the DLL wizards:

1. From the pop-up menu of the Build Configuration folder, click **Generate > All > All Targets**. This generates makefiles for all the DLL files defined in the folder and generates an all.mak file that calls the individual DLL makefiles. By choosing **All Targets**, you set the default behavior of the

makefile, when it is built. You can still override this default when you build (for example, you can choose to build Java targets only, and override the default of all targets).

To build the DLL and (optionally) JAR files:

1. From the Build Configuration folder's pop-up menu, select **Build > Out-of-Date Targets > C++**.

You always need to generate C++ targets, even if you selected **Java** as your business object implementation language. The other objects of the component (such as the data object implementation and managed object) are still implemented in C++.

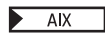
2. If you selected **Java** as your business object implementation language, then you also need to build the Java targets. From the Build Configuration folder's pop-up menu, select **Build > Out-of-Date Targets > Java**.

When you have a mixed-language application (some business objects are implemented in C++, some in Java) you must generate both C++ and Java targets for *all* components, even for those implemented in C++. The .jar files for C++ components allow Java components to interact with the C++ components on the server.

For this exercise, your application contains just a single component, so you only need to build the Java targets if you implemented the business object in Java.



The sample6C.dll and sample6S.dll files are stored in the working\NT\PRODUCTION directory.



The libsample6C.so and libsample6S.so files are stored in the working/AIX/PRODUCTION directory.

If you have a Java business object, the sample6C.jar and sample6S.jar are stored in the working\platform\PRODUCTION directory.

If you had a Java client application, regardless of the language your component is implemented in, you would also select **Build > Out-of-Date Targets > Java Client Bindings** to generate Java client bindings (working\platform\<build style>\JCB\JCBSample6C.jar). <build style> is one of the configuration directories: NOOPT, PRODUCTION, TRACE, TRACE_DEBUG. See Platform-specific information for more information.

For this exercise, you will be using QuickTest to run and test your application, which has its own build process that includes generation of client bindings.

Defining an application family and application

Define the application family (Sample6). An application family groups a set of applications so they can be installed as a unit.

1. From the pop-up menu of the Application Configuration folder, click **Add Application Family** to open the Application Family wizard.
2. Name the application family.
3. Click **Finish**.

The Sample6 application family appears under the Application Configuration folder.

Define the server application (Sample6Objects). An application defines a set of components that will operate together on the server. The application name you provide here is the name that will be used by System Management, when the application and its components are deployed.

1. From the pop-up menu of the application family, click **Add Application** to open the Application wizard.
2. Name the application.
3. Click **Finish**.

The Sample6Objects application appears under the Sample6 application family.

Configuring the component with the application

Configure the component's managed object (sample6MO sample6MO::csAgentMO) with the application (Sample6Objects), including the home (BOIMHomeOfRegHomes) that will be used to find and create it, and the container (TransientObjects) that will provide it with access to services.

1. From the pop-up menu of the application, click **Add Managed Object** to open the Managed Object Configuration wizard.
2. In the **Managed Objectlist**, select the component's managed object. The other lists should be automatically populated with the appropriate selections.
3. Click **Next** and select the data object implementation. Click **Add Another**. The appropriate data object implementation and DLL will be selected. If there were more than one data object implementation for the component, or it was being built into more than one DLL, you would click **Add Another** again to add the extra information.
4. Click **Next** to turn to the Container page.
5. Check the **Create a new container** option. An extra page is inserted into the wizard, to allow you to name the new container and specify its policies.
6. Click **Next** to turn to the Create New Container page.

7. Name the container, and set its behavior for methods called outside a transaction.

Most of the container's behavior is defined for you, based on the managed object you are configuring, and the data object implementation you selected.

8. Click **Next** and review the home. The appropriate home is already selected, and the default options are acceptable. If there were more than one appropriate home, you could choose the home to use.
9. Click **Finish**.

The csAgentMO managed object configuration appears under the Sample6Objects application.

Generate the application installation information:

1. From the pop-up menu of the Application Configuration folder, click **Generate**.

The DDL that defines the applications for System Management is generated into Working\platform\PRODUCTION\Sample6.

Testing the application

You can test the application using QuickTest, with the help of some additional Java client files and a QuickTest script that ship with the samples. For instructions on setting up and running QuickTest with your application, see the samples documentation (under <CBroker>, the install directory):

For C++:

samples\Tutorial\Fundamentals\transient\Docs\Transient.html

For Java:

samples\Tutorial\Fundamentals\jtransient\Docs\JTransient.html

Summary

You have created a component with transient data, that can be deployed on a server and accessed by a client or by other components on the server.

You can find information on installing the component on the server in the System Administration Guide, "Configure a New Application Environment" chapter, or online in the topic Install and Configure a New Application.

You can find information on testing the installed component with a QuickTest client application in "Chapter 13. Testing applications with QuickTest" on page 611.



Tutorial: Creating a component for new DB data

Objectives

- To create a simple component with persistent data, locatable by primary key.
- To generate a DB schema for the component's state data.
- To generate the code for the component.
- To build the DLLs for the component.
- To define the application configuration information for the component.

Before you begin

You need the following installed on your system:

- A CB Server
- A CB Client
- CB tools (VisualAge Component Development Toolkit), including Samples
- DB2 Universal Database
- DB2 SDK
-  VisualAge for C++ and (for Java applications) VisualAge for Java
-  IBM C and C++ Compilers for AIX V3.6.4 and (for Java applications) VisualAge for Java

Note: If you are not compiling on the same machine as you are building, the requirements are different. For example, if you are using Solaris and HP-UX machines to compile and test, you need either a Windows NT, or an AIX machine to build the model. On this machine that you use to build the model, you only require Object Builder installed. On the Solaris and HP-UX machines, you need the Component broker run time, the VisualAge Component Development Toolkit, DB2, and the compiler.

For a complete list of prerequisites that you must install on your system, see the *Getting Started with Component Broker* documentation for the appropriate platform.

You should be familiar with the Component Broker programming model, as described in the *IBM Component Broker Programming Guide*. At minimum, you should have read chapter 1, the introduction to the programming model.

Sample files

There are equivalent samples for this exercise. The samples include:

- A BusinessObjects project that contains the finished Object Builder model
- A zipped Rational Rose model that contains definitions for the business object, key, copy helper, and data object interface

- Documentation for the sample, including instructions on testing it with QuickTest

The samples are in the following directories (under <CBroker>, the install directory):

For C++:

```
samples\Tutorial\Fundamentals\lifeSamples\BusinessObjects
samples\Tutorial\Fundamentals\lifeSamples\Rose\TransientObjectRose.zip
samples\Tutorial\Fundamentals\lifeSamples\Docs\sample1.html
```

For Java:

```
samples\Tutorial\Fundamentals\jsimple\BusinessObjects
samples\Tutorial\Fundamentals\jsimple\Rose\JSimpleRose.zip
samples\Tutorial\Fundamentals\jsimple\Docs\jsimple.html
```

Description

In this exercise you define a simple component with database persistence, starting from the component's business object interface and working down to the component's DB schema.

The following tasks do not give explicit instructions for every step, but should at least get you into the right wizards. If you are experiencing problems, click the Help button within a wizard, or use the Help menu in Object Builder.

For this exercise, you will complete the following tasks:

1. Creating the project
2. Creating a business object interface
3. Adding a key and copy helper
4. Adding a business object implementation
5. Adding a data object implementation
6. Adding a persistent object and schema
7. Adding a managed object
8. Generating the code
9. Configuring the database
10. Defining a client DLL and server DLL
11. Defining an application family and application
12. Configuring the component with the application

Creating the project

Create a sample project to hold your work. For example, e:\tutorials\newdb

1. Start Object Builder.

2. In the Open Project wizard, type a name and path for the project directory.
3. Click **Finish**.
4. When asked whether you want to create a new project, click **Yes**.

Creating the business object interface

Define a business object file (sample1):

1. From the User-Defined Business Objects folder's pop-up menu, click **Add File** to open the Business Object File wizard.
2. Name the file.
3. Click **Finish**. The file now appears under the folder.

Add a module (sample1):

1. From the pop-up menu of the file, click **Add Module** to open the Business Object Module wizard.
2. Name the module.
3. Click **Finish**. The module now appears under the file.

Add an interface (csAgent). The business object interface defines the interface for the whole component. It will be contained in the sample1 module.

1. From the pop-up menu of the module, click **Add Interface** to open the Business Object Interface wizard.
2. Name the interface csAgent.
3. Click the page title and turn to the Attributes page.
4. Add the following attributes:
 - readonly float commissions
 - float commPercent
 - float pendingPaycheck
 - string agentName
 - readonly long id
5. Set the size of agentName to 100. You should always provide a size for string attributes.
6. Click **Next** and turn to the Methods page.
7. Add the following method:
 - void payCommission (in float amount)
8. Click **Finish**.

The csAgent interface now appears under the sample1 module. The attributes and methods appear in the Methods pane, when the interface is selected.

The attributes are represented as paired get and set methods, except for the read-only attributes, which have only get methods.

The interface does not have any business logic associated with it. The implementation of the interface is defined separately, in the business object implementation.

Adding the key and copy helper

Add a key (csAgentKey). The key allows client applications to locate or create instances of the component on the server. It consists of an attribute or attributes of the business object interface that uniquely identify an instance of the component. In this case, the csAgent id attribute is the appropriate choice.

1. From the interface's pop-up menu, click **Add Key** to open the Key wizard.
2. Accept the default name; select the id attribute and add it to the **Key Attributes** list.
3. Click **Finish**.

Even though the id attribute of the business object interface is read-only, the id attribute of the key is both readable and writable. The client application can set the value of the id on the key, and use it either to initialize a new instance of the component, or to locate an existing instance of the component, on the server.

Add a copy helper (csAgentCopy). The copy helper allows client applications to create and initialize instances of the component on the server, using one call to set numerous attributes, rather than one call per attribute.

1. From the interface's pop-up menu, click **Add Copy Helper** to open the Copy Helper wizard.
2. Accept the default name; select all listed attributes and add them to the **Copy Helper Attributes** list.
3. Click **Finish**.

Adding a business object implementation and data object interface

Add a business object implementation (csAgentBO) and data object interface (csAgentDO). The business object implementation contains the actual business logic of the component, including the method implementations. The separation of interface from implementation allows you to create multiple implementations of the same interface, to test different development options without affecting the interface. For this exercise, you will create only one implementation for the business object.

Any state data attributes (those attributes that cannot be deduced or derived from other attributes) become part of the data object interface. The separation of business logic (in the business object) from state data (in the data object) allows issues such as data persistence and integrity to be partitioned from the rest of the business logic.

1. From the interface's pop-up menu, click **Add Implementation** to open the Business Object Implementation wizard.

2. Change the **Pattern for Handling State Data** to **Delegating**. This is easier to debug than the **Caching** pattern.
3. Click the page title and turn to the Implementation Language page.
4. Select the language you want the business object to be implemented in (**C++** or **Java**).
5. Click the page title and turn to the Key and Copy Helper page. The appropriate key and copy helper are already selected.
6. Click the page title and turn to the Data Object Interface page.
7. Select all attributes and add them to the **State Data** list (to be preserved in the data object).
8. Click **Finish**.

csAgentBO appears under csAgent, and csAgentDO appears under csAgentBO.

Adding a method implementation

Type in the implementation for the method `payCommission`. The appropriate implementation logic for the data get and set methods, and for framework methods required by the programming model, are calculated for you by Object Builder. You only need to provide implementations for methods you explicitly defined.

1. Click on the business object implementation. The Methods pane shows the user-defined method `payCommission`, and the various user-defined attributes (in the form of paired get and set methods).
2. Click on `long id()`. The `id` attribute only has a get method, because it is read-only, as defined in the business object interface. The provided implementation appears in the Source pane.
3. Review the provided implementation for `long id()`. The get method for the `id` attribute delegates directly to its equivalent attribute in the data object, as defined by the **Delegating** pattern chosen in the business object implementation.
4. In the Methods pane, click on the `payCommission` method: `void payCommission (in float amount)`. The signature for the method appears in the source pane, based on the definition you provided in the business object interface and the language you selected in the business object implementation. The method does not have an implementation yet. You must provide the implementation for user-defined methods.
5. In the Source pane, provide the following implementation for `payCommission`:

C++

```
float tmp;
tmp = amount * iDataObject->commPercent();
iDataObject->commissions(tmp);
iDataObject->pendingPaycheck(iDataObject->pendingPaycheck() + tmp);
```

Java

```
float tmp;  
tmp = amount * iDataObject.commPercent();  
iDataObject.commissions(tmp);  
iDataObject.pendingPaycheck(iDataObject.pendingPaycheck() + tmp);
```

You can continue to the next step. When you click on other objects, the implementation will disappear from the Source pane, but the code you typed is now part of the project model, and will be generated as part of the source code for the component (in the file `sample1B0_I.cpp` for a C++ business object, or `_csAgentB0Base.java` for a Java business object).

Adding a data object implementation

Add a data object implementation (`csAgentDOImpl`). The data object implementation defines the way in which you want to handle the component's state data. As with the business object implementation, the separation of interface and implementation allow you to create multiple implementations of the same data object, without affecting the interface. For this exercise, you will create only one implementation for the data object.

1. From `csAgentDO`'s pop-up menu, click **Add Implementation** to open the Data Object Implementation wizard to the Name and Platform page.
2. Accept the default name and platform settings, and click **Next** to turn to the Behavior page.
3. Set the following patterns:
 - **Environment - BOIM with any key**
The component will be locatable by its key (instead of being locatable by a UUID).
 - **Type of Persistence - Embedded SQL**
The component's data will be stored in a database, and accessed by the data object's persistent object directly (instead of using a cache of the database's data managed by the Cache Service).
 - **Data Access Pattern - Delegating**
The data object will pass on calls for data directly to its persistent object, which provides a mapping to the database table (instead of using a local copy of the data, and keeping it synchronized with the database).
4. Click the page title and turn to the Key and Copy Helper page.
`csAgentKey` and `csAgentCopy` should already be selected.
5. Click **Finish**.

`csAgentDOImpl` appears under `csAgentDO`.

Because you selected a form of persistence other than transient (**Embedded SQL**), you will need to define a persistent object, which will handle the persistence and retrieval of component data.

Adding a persistent object and schema

Add a persistent object (csAgenPO), schema (CBSAMPDB.csAgent), and schema group (CBSAMPDBGroup). The persistent object provides a mapping from data object attributes (C++ data types) to table columns (SQL data types); the schema defines the table being mapped to; and the group represents the .sql file that contains the schema (and potentially multiple schemas).

1. From the data object implementation's pop-up menu, click **Add Persistent Object and Schema** to open the Add Persistent Object and Schema wizard to the Names page.
2. Name the schema group (CBSAMPDBGroup) and database (for example CBSAMPDB). Accept the default for the other names.
3. Click **Next** and review the attribute mappings.
4. Click **Finish**.

The persistent object and schema appear under the data object implementation.

In the DBA-Defined Schemas folder, they appear under the schema group you named.

Adding a managed object

Add a managed object (csAgentMO). The managed object mediates the interaction between the client application and the component, or between other components and this component. It exposes the business object interface to the client application and other components, and accesses any relevant services before and after a call. You add the managed object to the component's business object implementation.

1. From the business object implementation's pop-up menu, click **Add Managed Object** to open the Managed Object wizard to the Name and Service page.
2. Accept all the defaults and click **Finish**.
3. Click **File > Save** to save your work before continuing to the next step.

Generating the code

You are now ready to generate the code for the objects you have defined. Generated code will appear in your project's `\Working\platform` directory (for example, `e:\tutorials\transient\working\NT`).

1. From the pop-up menu of the User-Defined Business Objects folder, click **Generate > All**.

The code generation can take several minutes.

2. Review the contents of the `Working\platform` directory. All the source files for the component have been generated, and you can now define how to build them.

While you can view the generated source files for a particular object (by selecting **View Source** from its pop-up menu), you cannot edit the source files through Object Builder. If you edit them outside of Object Builder, you should restrict your changes to method implementations, and import your changes back into Object Builder before re-generating.

Configuring the database

You need to define (in DB2) the CBSAMPDB database and csAgent table that your component will access. You should have a database administrator perform this procedure.

To configure the database and table, use the following commands:

 In a DB2 command window (DB2 CLP), type:

 At a command prompt, type:

```
db2 create database CBSAMPDB
db2 connect to CBSAMPDB
db2 -t -f csAgent.sql
```

Note: The syntax for the creation of the table is provided by Object Builder in the generated SQL file for the CBSAMPDB.csAgent schema (csAgent.sql).

Defining a client DLL and server DLL

Configure the build process that will create the client and server DLLs.

While 'DLL' is the generic term used in Object Builder, the configuration for the build actually results in whatever the appropriate targets are for the selected platforms. For example, on AIX the build process creates shared library files (lib*.so). If you chose to create a Java business object, then in addition to DLLs there will also be JAR files. The names for these files are derived from the name you provide for the DLL file, within the DLL configuration node.

Start by defining the client DLL configuration and client DLL or library file (for this exercise, name them both sample1c). The client DLL provides client applications with access to the component on the server, using the key and copy helper. You must also include the business object interface, which defines the methods and attributes of the component that the client can access.

1. From the pop-up menu of the Build Configuration folder, click **Add Client DLL** to open the Client DLL wizard to the Name and Options page.
2. Name the configuration. This is the name that uniquely identifies the configuration node.

3. Name the library as well. This is the name for the makefile and for the resulting DLL file. You can create multiple configuration nodes that produce different versions of the same DLL. For example, you could set different compile and link options to produce a development version and a deployment version of the same DLL.
4. For this exercise, the default configuration options are sufficient.
5. Click the title and turn to the Client Source Files page.
6. Select all the client files and add them to the **Items Chosen** list.
7. Click **Finish**.

The sample1c DLL configuration appears under the Build Configuration folder.

Define the server DLL configuration and server DLL file (for this exercise, name them both sample1s). The server DLL is installed on the server to deploy the component, making it available for access by client applications and other components.

1. From the pop-up menu of the Build Configuration folder, click **Add Server DLL** to open the Server DLL wizard.
2. Name the configuration and target DLL file.
3. Click **Next** to turn to the Libraries to Link With page.
4. Add the client DLL file. The server DLL needs access to the client DLL because the client DLL defines the component's interface, and the component implementation inherits from it.
5. Click **Next** to turn to the Server Source Files page.
6. Select all the server source files for the component, and add them to the **Items Chosen** list.
7. Click **Finish**.

The sample1s DLL configuration appears under the Build Configuration folder.

Building the DLLs

To generate the makefiles, based on the configuration choices you made in the DLL wizards:

1. From the pop-up menu of the Build Configuration folder, click **Generate > All > All Targets**. This generates makefiles for all the DLL files defined in the folder and generates an all.mak file that calls the individual DLL makefiles. By choosing **All Targets**, you set the default behavior of the makefile, when it is built. You can still override this default when you build (for example, you can choose to build Java targets only, and override the default of all targets).

To build the DLL and (optionally) JAR files:

1. From the Build Configuration folder's pop-up menu, select **Build > Out-of-Date Targets > C++**.

You always need to generate C++ targets, even if you selected **Java** as your business object implementation language. The other objects of the component (such as the data object implementation and managed object) are still implemented in C++.

2. If you selected **Java** as your business object implementation language, then you also need to build the Java targets. From the Build Configuration folder's pop-up menu, select **Build > Out-of-Date Targets > Java**.

When you have a mixed-language application (some business objects are implemented in C++, some in Java) you must generate both C++ and Java targets for *all* components, even for those implemented in C++. The .jar files for C++ components allow Java components to interact with the C++ components on the server.

For this exercise, your application contains just a single component, so you only need to build the Java targets if you implemented the business object in Java.

 The sample1C.dll and sample1S.dll files are stored in the working\NT\PRODUCTION directory.

 The libsample1C.so and libsample1S.so files are stored in the working/AIX/PRODUCTION directory.

If you have a Java business object, the sample1C.jar and sample1S.jar are stored in the working*platform*\PRODUCTION directory.

If you had a Java client application, regardless of the language your component is implemented in, you would also select **Build > Out-of-Date Targets > Java Client Bindings** to generate Java client bindings (working*platform*\<build style>\JCB\JCBsample1C.jar). <build style> is one of the configuration directories: NOOPT, PRODUCTION, TRACE, TRACE_DEBUG. See Platform-specific information for more information.

For this exercise, you will be using QuickTest to run and test your application, which has its own build process that includes generation of client bindings.

Defining an application family and application



Define the application family (Sample1). An application family groups a set of applications within System Management.

1. From the pop-up menu of the Application Configuration folder, click **Add Application Family** to open the Application Family wizard.
2. Name the application family.

3. Click **Finish**.

The Sample1 application family appears under the Application Configuration folder.

Define the server application (Sample1Objects). An application defines a set of components that will operate together on the server. The application name you provide here is the name that will be used by System Management, when the application and its components are deployed.

1. From the pop-up menu of the application family, click **Add Application** to open the Application wizard.
2. Name the application.
3. Click **Next** to turn to the Additional Executables page.
4. Select the platform you are configuring the application for (for example, **NT Files**).
5. Add the file csAgent.sql:
 - a. Click **Add Another**.
 - b. Click the **Browse** button to open the Executables to Include dialog.
 - c. Locate your Object Builder working directory.
 - d. Select csAgent.sql
 - e.  Click the **Open** button.
 - f.  Click the **OK** button.
6. Add the file csAgentPO.bnd, in the same manner.
7. Click **Finish**.

The Sample1Objects application appears under the Sample1 application family.

Configuring the component with the application

Configure the component's managed object (sample1MO sample1MO::csAgentMO) with the application (Sample1Objects). Create a new container instance (Sample1Container) that will **Throw an exception** when a method is called outside a transaction.

1. From the pop-up menu of the application, click **Add Managed Object** to open the Managed Object Configuration wizard.
2. In the **Managed Objectlist**, select the component's managed object. The other lists should be automatically populated with the appropriate selections.
3. Click **Next** and select the data object implementation. Click **Add Another**. The appropriate data object implementation and DLL will be selected. If there were more than one data object implementation for the component,

or it was being built into more than one DLL, you would click **Add Another** again to add the extra information.

4. Click **Next** to turn to the Container page.
5. Check the **Create a new container** option. An extra page is inserted into the wizard, to allow you to name the new container and specify its policies.
6. Click **Next** to turn to the Create New Container page.
7. Name the container, and set its behavior for methods called outside a transaction.

Most of the container's behavior is defined for you, based on the managed object you are configuring, and the data object implementation you selected.

8. Click **Next** and review the home. The appropriate home is already selected, and the default options are acceptable. If there were more than one appropriate home, you could choose the home to use.
9. Click **Finish**.

The csAgentMO managed object configuration appears under the Sample1Objects application, and the new container Sample1Container appears in the Container Definition folder. You can review the properties of the container, including the service and data access patterns that have been selected for you, by clicking **Properties** from the container's pop-up menu.

Generate the application installation information:

1. From the pop-up menu of the Application Configuration folder, click **Generate**.

The DDL that defines the applications for System Management is generated into Working\platform\PRODUCTION\Sample1.

Close Object Builder:

1. Click **File > Save** to save your work.
2. Click **File > Exit** to close Object Builder.

Summary

You have created a component with data stored in a database. The component can be deployed on a server and accessed by a client or by other components on the server.

You can find information on installing the component on the server in the *System Administration Guide*, "Configure a New Application Environment" chapter, or online in the topic Install and Configure a New Application.

You can find information on testing the installed component with a QuickTest client application in "Chapter 13. Testing applications with QuickTest" on page 611.

Additional tutorials

The following tutorials provide introductions to specific aspects of Object Builder functionality that go beyond basic tasks. Before doing a tutorial, you should read any associated introductory and conceptual information, as described in the tutorial's pre-requisites.

- "Tutorial: Creating local-only objects" on page 220
- "Tutorial: Inheritance with attributes duplication" on page 310
- "Tutorial: Inheritance with key duplication" on page 327
- "Tutorial: Inheritance with a single datastore" on page 344
- "Tutorial: Inheritance with views" on page 362
- "Tutorial: Launching a remote OS/390 build" on page 573
- "Tutorial: Exporting from Rose" on page 81
- "Tutorial: Importing into Rose" on page 94
- "Tutorial: Team development with Rose" on page 449
- "Running the QuickTest tutorial" on page 647
- "Tutorial: Unit test for procedural adaptors" on page 166
- "Tutorial: Creating a component for PA data (bottom-up)" on page 167
- "Tutorial: Creating an inbound message application" on page 188
- "Tutorial: Creating an outbound message application" on page 203
- "Tutorial: Composite component creation" on page 267
- "Tutorial: Developing a multi-platform application" on page 429

RELATED CONCEPTS

"Object Builder" on page 1

RELATED TASKS

"Chapter 2. Tutorials for Object Builder" on page 39

"Developing in Object Builder" on page 19

Chapter 3. Using Rational Rose with Object Builder

You can design your application in Rose, and then export the design to Object Builder. You can also import an Object Builder project back into Rose. You need to modify Rose to support the import and export process, which uses the Rose Bridge.

If you export an incomplete design to Object Builder and make changes to it in its Object Builder form, make sure you import the Object Builder project back into Rose before doing any more work with the Rose model. If you do not import the changed project and continue work with the new model, your changes will be lost the next time you export.

You can export and import the following design elements, using the Rose Bridge:

- Packaging information (mapped to projects and modules)
- Classes (mapped either to component objects or to IDL constructs)
- One-to-one relationships (mapped to object references)
- One-to-many relationships (mapped either to an object relationship stored in a collection, or to a sequence attribute).
- Class inheritance (mapped to component inheritance)

By default, classes in your design are mapped to components in Object Builder.

The main design tasks are as follows:

1. "Importing Component Broker frameworks" on page 68
2. "Exporting a design from Rose" on page 80
3. "Tutorial: Exporting from Rose" on page 81
4. "Working with an exported design" on page 88
5. "Importing a project into Rose" on page 92
6. "Tutorial: Importing into Rose" on page 94
7. "Exporting a Rose design to a team environment" on page 445
8. "Importing a Rose design from a team environment" on page 447
9. "Tutorial: Team development with Rose" on page 449

RELATED CONCEPTS

"The Rose Bridge" on page 69

Components (*Programming Guide*)

Application Architecture (*Programming Guide*)

RELATED TASKS

- “Developing in Object Builder” on page 19
- “Setting up Rose 98” on page 65
- “Setting up Rose 98i and Rose 2000” on page 66

RELATED REFERENCES

- “Rose to Object Builder mapping rules” on page 97
- “Object Builder to Rose mapping rules” on page 123
- “Rose properties and bridging guidelines” on page 70

Rose

► CHANGED

Rational Rose is an object-oriented analysis and design modeling tool. You can use it to design your application, and then export the design to Object Builder, where you can finish its implementation. You can also import Object Builder projects into Rose, to work with an existing design.

Note: In order to export to and import from Object Builder, you must use the full version of Rose 98 (or later): Enterprise edition, Professional C++ edition, or Professional Java edition, not just the Rose Modeller. Only the full version supports code generation properties, which are required by the export process.

To use Rose with Object Builder, you must first customize it, and then load the Component Broker frameworks. You can then use the Component Broker frameworks in your design.

When you export, the classes and relationships you defined in Rose are mapped to IDL equivalents in Object Builder. You can also define additional properties in Rose, to have the Rose Bridge create additional Component Broker objects for your design during the export process.

When you import, the elements of the Object Builder project are mapped to their equivalents in Rose.

RELATED CONCEPTS

- “The Rose Bridge” on page 69
- “Rose to Object Builder mapping rules” on page 97
- “Object Builder to Rose mapping rules” on page 123

RELATED TASKS

- “Setting up Rose 98” on page 65
- “Setting up Rose 98i and Rose 2000” on page 66
- “Chapter 3. Using Rational Rose with Object Builder” on page 63

Setting up Rose 98

►CHANGED

You can use Rose to create a design for your application, which you can then export to Object Builder. You can also import Object Builder projects into Rose, to work with an existing design.

Note: In order to export to and import from Object Builder, you must use the full version of Rose Enterprise edition, Professional C++ edition, or Professional Java edition, not just the Rose Modeler. Only the full version supports code generation properties, which are required by the export process.

Before you can use Rose with Object Builder, you must configure its import and export facility, the Rose Bridge. Once the Rose Bridge is configured, you can load Component Broker frameworks into Rose, create your design, and export to and import from Object Builder.

To configure the Rose Bridge for Rose 98, follow these steps:

1. Add the export and import options to the Rose **File** menu, as follows:
 - a. Create a backup copy of the file `rose.mnu` (for example, `rose.bak`).
 - b. Add the following lines to the `rose.mnu` file:

```
Menu File
{
    Separator
    option "Export to Object Builder"
    {
        RoseScript $BOSS_PATH\r982c.ebx
    }
    option "Import from Object Builder"
    {
        RoseScript $BOSS_PATH\c2r98.ebx
    }
}
```

2. Create the Rose path map `BOSS_PATH`, which will point to the directory path that contains the Component Broker model files. This allows you to import the model files, and specifies the location of the export script (`r982c.ebx`) and import script (`c2r98.ebx`).
 - a. Click **File > Edit Path Map**.
 - b. Set the `BOSS_PATH` variable. For example, if you installed the product into `<path>\Cbroker`, set `BOSS_PATH` to `<path>\Cbroker\rose`, which is the directory that contains the `*.cat` files for the Component Broker model.

Some additional Component Broker-specific properties have been added to Rose to enable more information exchange between Rose and Object Builder. To enable the use of these additional properties, you must replace the `roseidl.pty` and `roseddl.pty` files that come with Rose 98 with the Component Broker versions. To replace these files, follow these steps:

1. Change to the directory where Rose 98 is installed
2. Create a backup copy of the file roseidl.pty (for example, roseidl.bak)
3. Create a backup copy of the file roseddl.pty (for example, roseddl.bak)
4. Copy the Component Broker versions of these files to the current directory from the rose subdirectory of your Component Broker install:

```
copy <path>\CBroker\rose\*.pty
```

RELATED CONCEPTS

“Rose” on page 64

“The Rose Bridge” on page 69

RELATED TASKS

“Importing Component Broker frameworks” on page 68

“Chapter 3. Using Rational Rose with Object Builder” on page 63

“Setting up Rose 98i and Rose 2000”

Setting up Rose 98i and Rose 2000



You can use Rose to create a design for your application, which you can then export to Object Builder. You can also import Object Builder projects into Rose, to work with an existing design.

Note: In order to export to and import from Object Builder, you must use the full version of Rose Enterprise edition, Professional C++ edition, or Professional Java edition, not just the Rose Modeler. Only the full version supports code generation properties, which are required by the export process.

Before you can use Rose with Object Builder, you must configure its import and export facility, the Rose Bridge. Once the Rose Bridge is configured, you can load Component Broker frameworks into Rose, create your design, and export to and import from Object Builder.

To configure the Rose Bridge for Rose 98i or Rose 2000, follow these steps:

1. Add the export and import options to the Rose **File** menu:
 - a. Create a backup copy of the file rose.mnu (for example, rose.bak).
 - b. Add the following lines to the rose.mnu file:

```
Menu File
{
  Separator
  option "Export to Object Builder"
  {
    RoseScript $BOSS_PATH\r982c.ebx
  }
  option "Import from Object Builder"
```



```

    {
        RoseScript $BOSS_PATH\c2r98.ebx
    }
}

```

2. Create the Rose path map BOSS_PATH, which will point to the directory path that contains the Component Broker model files. This allows you to import the model files, and specifies the location of the export script (r982c.ebx) and import script (c2r98.ebx).
 - a. Select **File > Edit Path Map**.
 - b. Set the BOSS_PATH variable. If you installed the product into the d:\Cbroker directory, set BOSS_PATH to d:\Cbroker\rose, which is the directory that contains the .cat files for the Component Broker model.

Some additional Component Broker-specific properties have been added to Rose to enable more information exchange between Rose and Object Builder. To enable the use of these additional properties, you must replace the rosejava.pty and rosecpp.pty files that come with Rose 98i or Rose 2000 with the Component Broker versions. To replace these files:

1. At a DOS prompt, go to the directory where Rose 98i is installed.
2. If you have a java subdirectory under the Rose 98i or Rose 2000 base directory, change to it. Create a backup copy of the file rosejava.pty (for example, rosejava.bak).
3. Copy the Component Broker version of this file to the current directory from the rose subdirectory of your Component Broker install:

```
copy d:\CBroker\rose\rosejava.pty
```
4. If you have a cpp subdirectory under the Rose 98i base directory, change to it. Create a backup copy of the file rosecpp.pty (for example, rosecpp.bak).
5. Copy the Component Broker version of this file to the current directory from the rose subdirectory of your Component Broker install:

```
copy d:\CBroker\rose\rosecpp.pty
```

Models that have been developed using Rose 98 will require a replacement of their properties to correspond to the new Rose 98i or Rose 2000 property files. For each model that has been developed using Rose 98, do the following steps:

1. Load the model into Rose 98i.
2. Select **Tools > Model Properties > Replace**.
3. In the Replace Model Properties dialog, select the appropriate .pty file to use (rosejava.pty in the java subdirectory if you are developing Java models, rosecpp.pty in the cpp subdirectory if you are developing C++ models).
4. Click **Open**.

If the model that you have loaded has both C++ and Java classes:

1. Select **Tools > Model Properties > Update**.
2. In the Update Model Properties dialog, select the appropriate .pty file to use (whichever .pty file was not selected in step 3 above).
3. Click Open.

Once you have updated the model with the appropriate properties, save the model by selecting **File > Save**.

Once the model has been saved with the new properties you will be able to bridge information between Rational Rose 98i or Rose 2000 and Object Builder.

RELATED CONCEPTS

“Rose” on page 64

“The Rose Bridge” on page 69

RELATED TASKS

“Importing Component Broker frameworks”

“Chapter 3. Using Rational Rose with Object Builder” on page 63

“Setting up Rose 98” on page 65

Importing Component Broker frameworks

This task is only necessary if you plan to use advanced Component Broker concepts in your analysis and design. The Rose Bridge maps your design elements to components with default behavior. If you want customized behavior (for example, you want to map a class to a specialized home), you can import Component Broker frameworks, such as the Managed Object Framework, into Rose. You can then use (for example, inherit from) these frameworks in Rose.

To import Component Broker frameworks into Rose, follow these steps:

1. Start Rose.
2. Create a new model (using the **File > New** menu option) or load an existing model.
3. Import the Component Broker Framework .cat files (managed.cat, services.cat, boim.cat):
 - a. Click **File > Units > Load**. A dialog box opens, in which you can specify the model files you want to import.
 - b. In the **Files of type** field, type *.cat.
 - c. Navigate to the <path>\Cbroker\rose directory.
 - d. Select one of the framework .cat files.
 - e. Click **Open**. The model files are loaded into Rose, and presented in the current view as a category or package that is selected by default.

- f. Click elsewhere in the current view (to avoid importing the next .cat file into the selected category).
- g. Load the other .cat files, using the same procedure.
Note: Imported categories all appear in the same spot in the current view, which means you often only see the last-imported category in the view. You can drag and drop a category to reveal the categories beneath.

The structure of the Component Broker Frameworks will appear in Rose in accordance with the Rose-to-IDL name scoping relationship. For example, in IDL, the Managed Object Framework contains several modules. In Rose, the Managed Object Framework appears as a package with subpackages corresponding to the modules that it contains. When you expand any of these subpackages in Rose, the classes (corresponding to the interfaces) that it contains are shown. For example, if you expand the IManagedClient subpackage, the IManageable and IHome classes are displayed.

You can now use the framework concepts in your design.

4. Perform the analysis and design of your application in Rose and save the design model.

RELATED CONCEPTS

"Rose" on page 64

RELATED TASKS

"Chapter 3. Using Rational Rose with Object Builder" on page 63

"Exporting a design from Rose" on page 80

The Rose Bridge

The Rose Bridge provides the ability to export from Rose to Object Builder, and to import from Object Builder to Rose.

The Rose Bridge process uses subdirectories of the targeted Object Builder projects to store information in, as follows:

- *project*\Model
Contains the target Object Builder project model.
- *project*\Import
Contains the udbo.*.xml files created by the export from Rose. These files define all the elements that are importable into Object Builder, and are used to create the project model.

- *project\XMI*
Contains the XML file created by an export or import from or to Rose. This file stores all the elements that cannot be mapped between a Rose model and an Object Builder model.

The Rose Bridge export and import behavior are described in:

- “Rose Bridge export” on page 79
- “Rose Bridge import” on page 89

For guidelines while bridging, refer to: “Rose properties and bridging guidelines”.

RELATED CONCEPTS

“Rose” on page 64

RELATED TASKS

“Importing Component Broker frameworks” on page 68

“Setting up Rose 98” on page 65

“Setting up Rose 98i and Rose 2000” on page 66

“Exporting a design from Rose” on page 80

“Exporting a Rose design to a team environment” on page 445

“Working with an exported design” on page 88

“Importing a project into Rose” on page 92

“Importing a Rose design from a team environment” on page 447

RELATED REFERENCES

“Rose to Object Builder mapping rules” on page 97

“Object Builder to Rose mapping rules” on page 123

Rose properties and bridging guidelines



Use only Rational Rose to modify objects originally created in Rose

When you rebridge to move information from Rational Rose to Object Builder, make sure that all items that are created in Rose are modified in Rose. For example, if you bridge from Rose to Object Builder, change a class name in Object Builder, and then rebridge, the new name that was added in Object Builder will be lost. If the name change is made in Rose, however, both Rose and Object Builder will have the new name for the class that is represented.

Ensure that the UUIDs for an artifact in Rose and Object Builder are the same

A UUID is generated for each artifact created by the Rose Bridge for import into Object Builder. This UUID is the key to synchronizing information between the two tools. If, for whatever reason, the UUID maintained in the UML XMI for an artifact, and the UUID which exists in Object Builder for that artifact become different, the potential for problems greatly increases.

Use Rose whenever possible to create artifacts that will exist in the Object Builder model

If an artifact is created in Object Builder that can be created in Rose as well, either the Rose model does not reflect the latest implementation in Object Builder or a parallel artifact is created in Rose to mirror the one created in Object Builder. Although you have created what appear to be the same artifact in both places, they are not the same because they have different UUIDs. If the attribute name is the same in both places and a rebridge is done, obimport will not create a second attribute with the same name (the attribute from Rose with the same name will not be imported). If you rename the attribute in Rose, however, rebridging will result in two attributes in the Object Builder model (one with the original attribute name from Object Builder, and one with the new name from Rose).

Deleting objects in Object Builder

If, for some reason, an artifact needs to be deleted in Object Builder and that artifact has been created in Rose, it should be modified or recreated in Rose and rebridged to recreate it in Object Builder. Depending upon the artifact deleted, the UML XMI file may need to be modified to remove any invalidated references to UUIDs which no longer exist in the Object Builder model due to the deletion.

Recovering from a bridge that results in XMI files that fail during import

Identify the error that results in the import failure by examining the file `export_messages.txt` that captures the messages during bridging. The error will report the UUID for the construct that fails to import into Object Builder. The import errors are almost invariably because there is a UUID synchronization problem between the contents of the UML XMI files and the contents of the Object Builder model.

To correct this problem, follow these steps:

1. Delete the construct (attribute, method, class) that has the problem in Object Builder
2. In Rose, recreate the structure, and rebridge. This will create the same construct with a new UUID that is synchronized within Rose and Object Builder.
3. It is important to ensure the class that you are rebridging has the appropriate information set up so the desired Object Builder artifacts are created.

Note: If you had added Object Builder-specific information (such as method body implementations, and had created managed objects, data object implementations, and so on), you may need to rework the Object Builder model once the rebridge is done.

Properties that can be set in Rose, and the consequences on rebridging, of changing these properties

Package properties

Name

Changing the name of a package, and rebridging results in the associated Object Builder file, or module being renamed.

Construct properties

Name

Changing the name of a construct, and rebridging results in the associated Object Builder construct being renamed.

IDLSpecificationType

The IDLSpecificationType of a construct must not be changed once it has been bridged to Object Builder. Changing this property is the equivalent of deleting the Object Builder construct, and adding a new one of the new type. The existing Object Builder construct must be deleted before rebridging. Any references to the construct by other artifacts will be removed as well. The class that represents the construct must be deleted in Rose. References in the Rose model to this type (for example, attribute types) will not be removed. Adding a new class, giving it the same name as the one deleted, and setting the IDLSpecificationType appropriately results in all references to the old type being converted to the new type after a rebridge.

Member names

Changing a member name, and rebridging results in the name change being propagated to the associated Object Builder member.

Member types

Changing a member type, and rebridging results in the name change being propagated to the associated Object Builder member.

ImplementationType

Changing the ImplementationType, and rebridging results in the change being propagated to the associated Object Builder artifact.

ConstValue

Changing the ConstValue, and rebridging results in the change being propagated to the associated Object Builder artifact.

Class properties

IDLSpecificationType (as described under **Construct properties**).

ObjectType

Changing ObjectType from a user-defined business object to a local-only object, and rebridging results in the interface and file structure (file and module) being moved to the local-only section. However, inheritance is not handled correctly (the interface inherits from both IManageable and INonManageable after the rebridge). This situation can be resolved by deleting the extra inheritance in the interface Properties notebook after rebridging.

Name

Changing the class name, and rebridging results in the change being propagated to the associated Object Builder interface.

Comments

Changing the Documentation field in Rose, and rebridging updates the Comment field for the interface in Object Builder.

IsQueryable

Changing the IsQueryable property in Rose, and rebridging updates the property for the interface in Object Builder.

CreateImplementation

Changing this property from False to True, and rebridging creates a business object implementation for your interface in Object Builder. If an implementation has already been created in Object Builder before the rebridge, import errors result. If an implementation is required in Object Builder, it must always be created by setting this property to true in the Rose model, and rebridging.

Changing this property from True to False, and rebridging does not affect the Object Builder model because deletions are not propagated to Object Builder. If the property is then set back to True and rebridged, the bridge will work as it should because the UUID from the first bridge (when CreateImplementation was set to True) was preserved. If the implementation is deleted in Object Builder the entire file-module-interface structure must be deleted as well. In Rose, the package structure and class must be deleted and re-created to avoid UUID conflicts.

CreateKey

Changing this property from False to True, and rebridging creates a key implementation for your interface in Object Builder. If a key has already been created in Object Builder before the rebridge, import errors result. If a key is required in Object Builder, it must always be created by setting this property to true in the Rose model, and rebridging.

Changing this property from True to False, and rebridging does not affect the Object Builder model because deletions are not propagated to Object Builder. If the property is then set back to True and rebridged, the bridge will work as it should because the UUID from the first bridge (when CreateKey was set to True) was preserved. If the key is deleted in Object Builder, the entire file-module-interface structure must be deleted as well. In Rose, the package structure and class must be deleted and re-created to avoid UUID conflicts.

CreateCopyHelper

Changing this property from False to True, and rebridging creates a copy helper for your interface in Object Builder. If a copy helper has already been created in Object Builder before the rebridge, import errors result. If a copy helper is required in Object Builder, it must always be created by setting this property to true in the Rose model, and rebridging.

Changing this property from True to False, and rebridging does not affect the Object Builder model because deletions are not propagated to Object Builder. If the property is then set back to True and rebridged, the bridge will work as it should because the UUID from the first bridge (when CreateCopyHelper was set to True) was preserved. If the implementation is deleted in Object Builder, the entire file-module-interface structure must be deleted as well. In Rose, the package structure and class must be deleted and re-created to avoid UUID conflicts.

Attribute Properties

Name

Changing the name of an attribute, and rebridging results in the associated Object Builder attribute being renamed.

Type

Changing the type of an attribute, and rebridging results in the associated Object Builder attribute type being changed.

Initializer

Changing the initial value of an attribute, and rebridging results in the associated Object Builder attribute initializer being updated.

Export Control

This value must not be changed once bridged to Object Builder. Changing this property implicitly results in a delete and re-add in Object Builder, which is not currently supported by the Rose Bridge. If the Export Control value needs to be changed, delete the attribute in Object Builder and in Rose, then re-add the attribute in Rose with the appropriate values. Rebridging after this process results in the proper information being reflected in the Object Builder model.

Length

Changing the value of this property, and rebridging results in the update being reflected in the Object Builder Size field for the attribute (if the attribute type is 'string').

PrimaryKey

Changing this property from False to True, and rebridging results in the attribute being included in the key for your interface in Object Builder. If the attribute has to be included in the key, it must always be added by setting this property to True in the Rose model, and rebridging.

Changing this property from True to False, and rebridging does not affect the Object Builder model because deletions are not propagated to Object Builder. If the property is then set back to True and rebridged, the bridge will work as it should because the UUID from the first bridge (when PrimaryKey was set to True) was preserved. If an attribute must no longer be included in the key, the attribute must be deleted in Object Builder, then deleted and recreated in Rose with the PrimaryKey set to False. Rebridging after this has been done results in the Object Builder project having the appropriate information.

IsIncludedInCopyHelper

Changing this property from False to True, and rebridging results in the attribute being included in the copy helper for your interface in Object Builder. If the attribute must be included in the copy helper, it must always be added by setting this property to True in the Rose model, and rebridging.

Changing this property from True to False, and rebridging does not affect the Object Builder model because deletions are not propagated to Object Builder. If the property is then set back to True and rebridged, the bridge will work as it should because the UUID from the first bridge (when IsPrimaryKey was set to True) was preserved. If an attribute must no longer be included in the copy helper, the attribute must be deleted in Object Builder, then deleted and recreated in Rose with the IsIncludedInCopyHelper set to False. Rebridging after this has been done results in the Object Builder project having the appropriate information.

IsIncludedInDataObject

Changing this property from False to True, and rebridging results in the attribute being included in the data object for your interface in Object Builder. If the attribute must be included in the data object, it must always be added by setting this property to True in the Rose model, and rebridging.

Changing this property from True to False, and rebridging does not affect the Object Builder model because deletions are not propagated to Object Builder. If the property is then set back to True and rebridged, the bridge will work as it should because the UUID from the first bridge (when

IsIncludedInDataObject was set to True) was preserved. If an attribute must no longer be included in the data object, the attribute must be deleted in Object Builder, then deleted and recreated in Rose with the IsIncludedInDataObject set to False. Rebridging after this has been done results in the Object Builder project having the appropriate information.

IsReadOnly

Changing this property from False to True, and rebridging results in the attribute being represented as read-only in the Object Builder model.

Changing the property from True to False, and rebridging will result in the attribute being switched to read-write in the Object Builder model.

Override in subclass

An attribute with the same name and type in a subclass in Rose will be represented in Object Builder as an attribute to be overridden in the Object Builder model (if CreateImplementation is set to True for the subclass).

Changing the type of the subclass attribute, and rebridging results in a bridge error, and the attribute will not be bridged. This results in the implementation for the subclass in Object Builder not being affected by the rebridge. Changing the name of the subclass attribute will result in UUID problems in the Object Builder model. If the desired result is to remove the attribute from the Overrides list for the subclass, move the attribute to the left-hand side of the Attributes to Override list in the subclass implementation, and delete the attribute from the subclass in the Rose model. Rebridging after this has been done results in the appropriate information reflected in the Object Builder model.

Method Properties

Name

Changing the name of a method, and rebridging results in the associated Object Builder method being renamed.

Return Type

Changing the return type of a method, and rebridging results in the associated Object Builder method return type being renamed.

Export Control

This value must not be changed once it is bridged to Object Builder. Changing this property implicitly results in a delete and re-add in Object Builder, which is not currently supported by the Rose Bridge. If the Export Control value needs to be changed, delete the method in Object Builder and in Rose, then re-add the method in Rose with the appropriate values. Rebridging after this process results in the proper information being reflected in the Object Builder model.

Parameter Name

Changing the return type of a method, and rebridging results in the associated Object Builder method return type being renamed.

Parameter Type

Changing the type of a method parameter, and rebridging results in the associated Object Builder method parameter type being renamed.

Parameter Argument default

Changing the method parameter default, and rebridging results in the associated Object Builder method parameter default being updated.

Exceptions

Changing the list of Exceptions for a method, and rebridging will result in the associated Object Builder method exceptions being updated.

OperationIsOneWay

Changing the OperationIsOneWay property of a method, and rebridging results in the associated Object Builder method being updated.

Override in subclass

A method with the same name and type in a subclass in Rose will be represented in Object Builder as a method to be overridden in the Object Builder model (if CreateImplementation is set to True for the subclass). Changing the signature of the subclass method, and rebridging will result in a bridge error and the method will not be bridged. This will result in the implementation for the subclass in Object Builder not being affected by the rebridge. Changing the name of the subclass method will result in UUID problems in the Object Builder model. If the desired result is to remove the method from the Overrides list for the subclass, move the method to the LHS of the Methods to Override list in the subclass implementation and delete the method from the subclass in the Rose model. Rebridging after this has been done will result in the appropriate information reflected in the Object Builder model. **Class Relationships**

Role name

Changing the role name for an association in Rose, and rebridging results in the name change being reflected for the attribute or object relationship in Object Builder (depending upon the cardinality of the role).

Cardinality

Changing the cardinality of a role is the equivalent of deleting the associated Object Builder attribute, or object relationship, and re-adding a new artifact to represent the new cardinality, which is not supported by the Rose Bridge at this time. To accomplish this, you must first delete the attributes or object relationships that were created in Object Builder, then delete the association in

Rose, and re-add it with the correct cardinalities. This will result in the appropriate information being reflected in the Object Builder model.

MapAsObjectRelationship

Changing this property from True to False, and rebridging is the equivalent of deleting the object relationship for the role, and creating a sequence attribute to represent the collection, which is not supported by the Rose Bridge at this time. To implement this change, you must first delete the attributes or object relationships that were created in Object Builder for the relationship, delete the association in the Rose model, then re-create the association with the desired property settings. Rebridging after these changes have been made will result in an Object Builder model with the appropriate information. The same actions must be taken when changing this property from False to True.

RelationshipImplementation

Changing this property, and rebridging will change the object relationship implementation in the Object Builder model.

IsReadOnly

Changing this property from False to True, and rebridging will change the associated attribute or object relationship to read-only in Object Builder. Changing the property from True to False, and rebridging will make the Object Builder attribute or object relationship read-write.

Export Control

This value must not be changed once it is bridged to Object Builder. Changing this property implicitly results in a delete and re-add in Object Builder, which is not currently supported by the Rose Bridge. If the Export Control value needs to be changed, delete the attributes or object relationships in Object Builder, which correspond to the association, delete the association in Rose, then re-add the association in Rose with the appropriate values. Rebridging after this process will result in the proper information being reflected in the Object Builder model.

RELATED CONCEPTS

- “The Rose Bridge” on page 69
- “Rose Bridge export” on page 79
- “Rose Bridge import” on page 89

RELATED TASKS

- “Chapter 3. Using Rational Rose with Object Builder” on page 63
- “Exporting a design from Rose” on page 80
- “Importing a project into Rose” on page 92

RELATED REFERENCES

- “Naming objects” on page 128
- “Internationalization of data” on page 132
- “Rose to Object Builder mapping rules” on page 97

Rose Bridge export

Once you have completed a design in Rose, you can export it to an Object Builder project. You can export a design for use in a single Object Builder project, or break up the design into separate projects, according to the structure of .cat files in your design. The export process updates the \Model subdirectory of the selected project or projects, and also creates the XML files for the model in the project’s \Import directory. Once the export is complete, you can use Object Builder to further refine the model and to generate code

When you export from Rose, the export process generates an XML file in the target *project*\XMI subdirectory. This file allows the export process to track changes to design elements, so that if you change the name of a method in Rose and re-export, the change will be applied to the appropriate method in Object Builder. It also keeps track of any elements that do not have equivalents in both models, so that these elements are not simply lost in the bridging process. The .xmi file in the *project*\XMI subdirectory does *not* keep track of Rose elements outside of the Logical View.

You should not restructure your design after exporting. If you restructure your design (for example, move a class from one package to another), the export process will treat the change as a combination add and delete, rather than a move. This would result in two definitions of the class in Object Builder (a new class definition for its new position, and the old class definition for its old position), which is not valid.

Re-export process

When you re-export a model, the export process will add new elements to Object Builder, or update existing elements, but will not delete elements that already exist in Object Builder. To delete existing elements, you must work directly in Object Builder.

RELATED CONCEPTS

- “Rose” on page 64
- “The Rose Bridge” on page 69
- “Rose Bridge import” on page 89

RELATED TASKS

- “Chapter 3. Using Rational Rose with Object Builder” on page 63
- “Exporting a design from Rose” on page 80
- “Exporting a Rose design to a team environment” on page 445

RELATED REFERENCES

“Rose to Object Builder mapping rules” on page 97

“Rose properties and bridging guidelines” on page 70

Exporting a design from Rose

► CHANGED

Once you have completed your design work in Rose, you can export your design to an Object Builder project or projects. Rose must first be set up to work with Object Builder. When the export is complete, each class in your design will be mapped to a component in Object Builder. A component consists of a number of related objects, and, at minimum, a business object interface.

You can exclude portions of your design from the export (packages or classes) by setting the property `BridgeToOB=FALSE` in the package or class specification notebook. By default, your entire design is exported. If you set `BridgeToOB=FALSE` in a package, all packages and classes contained within this package will be excluded from the export.

Your design will be exported as either a single project (if your design is contained in a single .mdl file), or to multiple projects (if your design contains some packages that are stored in separate .cat files). The following task does not cover the multiple projects case: see the team development documentation for more information on exporting to a team environment.

To export, follow these steps:

1. Load your design in Rose.
2. Select **File > Export to Object Builder**. The Rose Bridge wizard opens to the Export from Rose 98 to Object Builder page.
In the **Source Model** field, the current Rose model file is selected by default.
3. Specify the Rose model you want to export, and add any necessary virtual symbols and associated actual path mappings to the Virtual Path Mapping listbox.
If you imported the Component Broker frameworks to use in your design, you will need to provide the mapping for `BOSS_PATH`. You can check the value for `BOSS_PATH` by clicking **File > Edit Path Map** in Rose.
4. In the **Target Project** field, specify the destination directory you want to store your project in.
5. You can check the **Log the error(s) to a file instead of message box** option to store errors in a file named `export_results.txt`, which will be generated into the target project directory.

6. You can check the **Log detailed trace information to a file** option to generate more detailed processing information into the `export_results.txt` file.
7. Click **Next**.
8. (Optional) Review the export structure. You can review which packages and classes are being exported (as defined by the `BridgeToOB` property for specific packages and classes), and review the target project directory.
9. Click **Finish**.

Your model is exported to the specified project directory. Your model is saved in Rose as part of the export process.

The classes and relationships you defined in Rose have been mapped to their Object Builder equivalents, and any component objects you specified have been defined in skeleton form.

You are now ready to work in Object Builder.

RELATED CONCEPTS

“The Rose Bridge” on page 69
Component (*Programming Guide*)

RELATED TASKS

“Chapter 3. Using Rational Rose with Object Builder” on page 63
“Importing Component Broker frameworks” on page 68
“Exporting a Rose design to a team environment” on page 445
“Working with an exported design” on page 88

RELATED REFERENCES

“Rose to Object Builder mapping rules” on page 97

Tutorial: Exporting from Rose

Objectives

To create a class in Rose.



To specify class and attribute properties that will affect how the class is exported.

To export a sample application from Rose into an Object Builder project.

Before you begin

You need the following installed on your system:

- A CB Server
- A CB Client
- CB tools, including Samples

- DB2 Universal Database
- DB2 SDK
-  VisualAge for C++ and (for Java applications) VisualAge for Java
-  IBM C and C++ Compilers for AIX V3.6.4 and (for Java applications) VisualAge for Java
- Rational Rose 98, Rose 98i or Rose 2000

You need Rational Rose installed and set up to work with Object Builder, as described in the task “Setting up Rose 98” on page 65 or “Setting up Rose 98i and Rose 2000” on page 66.

For a complete list of prerequisites that you must install on your system, see the *Getting Started with Component Broker* documentation for the appropriate platform.

You should be familiar with the Component Broker programming model, as described in the *IBM Component Broker Programming Guide*. At minimum, you should have read chapter 1, the introduction to the programming model.

You should be familiar with Rational Rose. If you are not familiar with the tool, take the Rose tutorial included with the software.

You should be familiar with Object Builder. If you are not familiar with the tool, run through one of the introductory tutorials, for example, “Tutorial: Creating a component with transient data” on page 39.

Sample files

Most of the samples shipped with Component Broker include Rose models. This exercise results in a model that corresponds to the Rose model for the transient data sample. The sample includes:

- A zipped Rational Rose model that contains definitions for a business object, key, copy helper, and data object interface
- Documentation for the sample

The samples are in the following directories (under <CBroker>, the install directory):

```
samples\Tutorial\Fundamentals\transient\Rose\TransientObjectRose.zip
samples\Tutorial\Fundamentals\transient\Docs\Transient.html
```

Description

This exercise describes how to create a class in Rose, prepare it for export to Object Builder, complete the export, and open the exported project. A follow-on exercise, “Tutorial: Importing into Rose” on page 94, describes how

to reverse the process, importing the project into Rose and updating the model with any changes that have been made to the project. A team development version, “Tutorial: Team development with Rose” on page 449, describes how to use a single Rose model with multiple Object Builder projects.

For this exercise, you will complete the following tasks:

1. Create a class in Rose.
2. Add Component Broker properties to the class and its attributes.
3. Export the model to an Object Builder project.
4. Open the project in Object Builder.

Creating the class

Start Rose, and add a simple class with five attributes and one method (Agent).

Start by adding packages that will contain the class. The packages will translate into module scoping for the class in Object Builder. The top-level package (LifeInsurance) will become a file, and the second-level package (sample6) will become a module in the file, in Object Builder.

Add packages and a class, using pop-up menus in the tree view (left-hand pane in Rose):

1. Start Rose.
2. In the tree view in the left-hand pane, locate the entry for the Logical view.
3. From its pop-up menu, click **New > Package**. A new package entry appears under the Logical view, with a default name selected for you to type over.
4. Name the new package LifeInsurance.
5. From the pop-up menu of the LifeInsurance package, click **New > Class Diagram**, and name the new diagram Main.
6. From the pop-up menu of the LifeInsurance package, click **New > Package**, and name the new package sample6.
7. From the pop-up menu of the sample6 package, click **New > Class Diagram**, and name the new diagram Main.
8. From the pop-up menu of the sample6 package, click **New > Class**, and name the new class Agent.
9. Double-click on the Logical View - Main class diagram. It opens as a Logical View window.
10. Click **Query > Add Classes** to open the Add Classes window.
11. Add Agent to the diagram, and click **OK**.

Add the attributes. Later you will add specific properties to the attributes, which effect how they are exported to Object Builder. for now, simply define their signatures:

1. From the pop-up menu of Agent in the diagram, click **New Attribute**. A placeholder attribute is added (named name, type of type, initial value of initval).
2. Type over each of the values for the new attribute, naming it commission, with type float.
3. Click elsewhere in the diagram to apply the changes.
4. Add the following attributes in the same way:
 - float commPercent
 - float pendingPaycheck
 - string agentName
 - long id

Add the operation payCommission:

1. From the pop-up menu of Agent, click **New Operation**. A placeholder operation is added (named opname, argument argname, return type return).
2. Type over each of the values for the new operation, naming it payCommission, with argument amount, and return value void.
3. Click elsewhere in the diagram to apply the changes.
4. From the pop-up menu of the specification in the tree view (under the Logical View folder), click **Open Specification**
5. Turn to the Details page of the Operation Specification notebook.
6. Change the argument type to float.

You could also set the argument default to the parameter-passing mode you prefer (in, out, inout). If you do not set a value, or set a value other than those listed, it defaults to in.

7. Click **OK**.

You now have a class named Agent, with attributes commissions, commPercent, pendingPaycheck, agentName, and id, and the method payCommission(float amount).

Adding Component Broker properties to the class

You can specify properties of the class and its attributes that will affect the way it is exported to Object Builder. Some of these properties are standard Rose properties that have meaning for the export process, others are specific to Component Broker, and were made available in Rose when you copied over customized .pty files during the Rose setup.

Customize the way the class will be exported, to create the following component objects in Object Builder: business object interface, business object implementation, data object interface, key, copy helper:

1. From the pop-up menu of the class in the class diagram, click **Open Specification** to open the Class Specification notebook.
2. Turn to the IDL page. The following properties map to component objects:
 - IDLSpecificationType
By default, it is set to Interface. A business object interface is created for the class. Other values you can set for this property would make the class export as a construct (for example, a struct or enum).
 - CreateImplementation
By default, it is set to False. A business object implementation and its accompanying data object interface are not created for the class.
 - CreateKey
By default, it is set to False. A key is not created for the class.
 - CreateCopyHelper
By default, it is set to False. A copy helper is not created for the class.
3. Click on CreateImplementation, then click the value False, and change it to True.
A business object implementation and its accompanying data object interface will be created for the class.
4. Set the values for CreateKey and CreateCopyHelper to True as well.
A key and copy helper will be created for the class.
5. Click **OK**.

Adding Component Broker properties to the attributes

Customize the way the attributes will be exported, to make all the attributes public, set the size of agentName to 100, make commissions and id read-only, and make id part of the key.

Customize id:

1. In the tree view, under the Logical View, select the attribute id.
2. From its pop-up menu, click **Open Specification** to open its Class Attribute Specification notebook.
3. On the General page, set its Export Control to public.
4. Turn to the DDL page, and set the PrimaryKey property to True.
5. Turn to the IDL page, and set the isReadOnly property to True.
6. Click **OK**.

The attribute id will now be generated as a public read-only attribute that is part of the business object, key, and copy helper.

Customize commission:

1. In the tree view, under the Logical View, select the attribute commission.
2. From its pop-up menu, click **Open Specification** to open its Class Attribute Specification notebook.
3. On the General page, set its Export Control to public.
4. Turn to the IDL page, and set the isReadOnly property to True.
5. Click **OK**.

The attribute commission will now be generated as a public read-only attribute that is part of the business object and copy helper.

Customize agentName:

1. In the tree view, under the Logical View, select the attribute agentName.
2. From its pop-up menu, click **Open Specification** to open its Class Attribute Specification notebook.
3. On the General page, set its Export Control to public.
4. Turn to the DDL page, and set the Length value to 100.
5. Click **OK**.

The attribute agentName will now be generated as a public attribute that has length=100, and is part of the business object and copy helper.

Customize commPercent and pendingPaycheck:

1. In the tree view, under the Logical View, select the attribute commPercent.
2. From its pop-up menu, click **Open Specification** to open its Class Attribute Specification notebook.
3. On the General page, set its Export Control to public.
4. Click **OK**.
5. Do the same for pendingPaycheck.

You have added properties that specify how the class maps to component objects and attributes in Object Builder. You are ready to export to an Object Builder project.

Exporting to Object Builder

To export to Object Builder, follow these steps:

1. Save and name your model (for example, e:\tutorials\rosemodels\agent.mdl). You cannot export an unnamed model.
2. Click **File > Export to Object Builder**. The Rose Bridge wizard opens.
3. In **Target Project**, specify a directory to export to (for example, e:\tutorials\roseagent\). The Rose Bridge will create the directory if necessary, and turn it into an Object Builder project directory.

You do not need to provide virtual path mappings. These are generally necessary only when your design is divided into multiple .cat files for

team development purposes, or when you have imported the Component Broker framework classes to do advanced design work.

4. Click **Finish**.

The Rose Bridge starts by saving your current Rose model. The Rose Bridge then exports an XML version of the model, consisting of two files: *project\Import\udbo.LifeInsurance.xml*, to become the component in an Object Builder project, and *project\XMI\roseagent.xml*, to hold any Logical View information that would otherwise be lost in the transfer. Finally, the Rose Bridge imports *udbo.LifeInsurance.xml* into Object Builder to create the new Object Builder model files.

You can now open the project in Object Builder and review the results of the export.

Opening the Object Builder project

Open the Object Builder project and review the exported component:

1. Start Object Builder.
2. In the Open Project wizard, type the name of the directory you exported to (for example, *e:\tutorials\roseagent*).
3. Click **Finish**. The project opens.
4. In the Tasks and Objects pane, expand the User-Defined Business Objects folder. You can see the business object file *LifeInsurance*.
5. Expand the file to show the *sample6* module, expand the module to show the Agent interface, expand the interface to show *AgentKey*, *AgentCopy*, and *AgentBO*, and expand *AgentBO* to show *AgentDO*. These objects were created according to the property settings of the Claim class in Rose, as follows:
 - *LifeInsurance* file and *sample6* module
Created because the class was in a nested package, and the class's *IDLSpecificationType* was set to *Interface*.
 - *AgentKey*
Created because the class's *CreateKey* property was set to *True*.
 - *AgentCopy*
Created because the class's *CreateCopyHelper* property was set to *True*.
 - *AgentBO* and *AgentDO*
Created because the class's *CreateImplementation* property was set to *True*.
6. Click the Agent interface. You can see its attributes and methods in the Methods pane.
7. Click the AgentBO implementation. You can see the get and set methods for the attributes, and the method signatures, in the Methods pane.

8. Click the AgentKey key. You can see the get and set methods for id, which you set to be part of the key with the PrimaryKey DDL property.
9. Click the AgentCopy copy helper. You can see the get and set methods for all the attributes, which are part of the copy helper by default.

You can review the skeleton signatures for the methods, the default implementations for get and set methods, and the framework methods added by Object Builder, by clicking on the attribute or method in the Methods pane.

Summary

You have created a class in Rose named Agent, defined its attributes and operations, and exported the result as a set of component objects to Object Builder. You can now continue to “Tutorial: Importing into Rose” on page 94, in which you customize the class and then import the changes into Rose. If you want, you can skip that scenario and continue on to “Tutorial: Team development with Rose” on page 449, in which you add a second class with an object relationship, and export the two classes into separate interdependent projects.

Working with an exported design

Once the export process is complete, you can start working with your design in Object Builder. To work with an exported design, complete the following steps in Object Builder:

1. Review the exported business object interfaces and their equivalent files.
2. Complete the skeleton objects created by the export.
3. Create data object implementations.
4. If you want data persistence, create or map to persistent objects and schemas.
5. Create managed objects (for packaging and instance management).

Review and edit exported objects as necessary:

1. Select the object in the Tasks and Objects pane.
2. From its pop-up menu click **Properties**. A wizard opens.
3. Click **Next** to go through all the pages of the wizard, and review its properties. Complete or change the contents of the wizard as you require.
4. Click **Finish**. Any changes you made are applied to the current model.

RELATED CONCEPTS

“The Rose Bridge” on page 69

RELATED TASKS

“Chapter 3. Using Rational Rose with Object Builder” on page 63

“Adding a data object implementation” on page 807

“Adding a persistent object and schema” on page 833

“Mapping a data object to a persistent object” on page 703

“Adding a managed object” on page 871

Rose Bridge import



You can make changes to the design in Object Builder, and then apply the changes to the original Rose model. If you are doing work in both Object Builder and Rose, make sure you keep the two versions synchronized. For example, if you change the Object Builder model, import the changes into Rose before doing any more work on the Rose model.

If the project was created entirely in Object Builder, then your project elements will be mapped to Rose model elements using default mapping rules.

If the project was created by exporting a design from Rose, then your project elements will be mapped back to their originals in the Rose model, using the mapping rules you defined in the Specification notebooks of the various elements. The .xmi file created by the export in the project\XMI subdirectory preserves these mappings, as defined in the Logical View. If all your model information is in the Logical View, then the import process creates a duplicate of your original project, and you can discard the original. If your original model has information in other views as well, then you will need to merge the two projects (the original, with its information in the other views, and the newly imported one, with its updated Logical View information). You can use the update feature provided with Rose to resolve these differences.

The import process works as follows:

1. The import process calls the `obexport` command to generate XML files for the project (`\Export\udbo.*.xml`).
2. The import process checks to see if there is an XML file in the project's `\XMI` subdirectory. This file is created by the Rose Bridge to preserve any information that would otherwise be lost during transfer between Rose and Object Builder.
3. The import process generates a Rose model file, based on the XML files in `\Export` and `\XMI`.
4. The import process updates the XML file in `project\XMI` to contain any Object Builder information that cannot be imported. For example, details of the implementation, key, and copy helper for a component, that cannot be stored as elements in Rose.

The import process maps Object Builder elements as follows:

- Business object files, modules, and interfaces that already have a mapping (because they were created by export from Rose) maintain that mapping.

- New business object files, modules, and interfaces (added directly to Object Builder, not by export from Rose) are mapped to packages, subpackages, and classes.
- Local-only files, modules, and interfaces that already have a mapping (because they were created by export from Rose) maintain that mapping.
- New local-only files, modules, and interfaces (added directly to Object Builder, not by export from Rose) are mapped to packages, subpackages, and classes with ObjectType property set to **Local-Only Objects**.
- Non-IDL type objects that already have a mapping (because they were created by export from Rose) maintain that mapping.
- New non-IDL type objects (added directly to Object Builder, not by export from Rose) are mapped to classes with ObjectType property set to **Non-IDL Type Objects**.
- IDL constructs with file or module scope become classes in Rose.
- IDL constructs with interface scope in Object Builder become nested classes inside the Rose class which represents the interface.
- Attributes of an interface become attributes of a class in Rose.
- Methods of an interface become operations of a class in Rose.
- Parent interfaces become class relations in Rose.
- Object relationships that were created by export from Rose are imported as the role of an association.
- Object relationships that were created in Object Builder (not by export from Rose) are imported as the role of a unidirectional association.
- Sequence attributes of the interface that were created by export from Rose are imported as the role of an association.

The import process keeps the following elements in the \XMI .xml file:

- Component objects other than the business object interface (business object implementation, data object interface, copy helper, key)
- The method bodies

The import process updates the following properties in the Rose specification notebooks:

- Class Specification, IDL page, CreateImplementation property is set if the business object interface has a business object implementation
- Class Specification, IDL page, CreateKey property is set if the business object interface has a key
- Class Specification, IDL page, CreateCopyHelper property is set if the business object interface has a copy helper

- Class Specification, IDL page, IsQueryable property is set if the business object interface has the option **The interface is queryable** checked, in its properties notebook
- Class Specification, IDL page, IDLSpecificationType property is set according to type of construct mapped from Object Builder (Interface, Typedef, Enumeration, Const, Exception, Struct, Union)
- Class Specification, IDL page, ObjectType property is set according to type of artifact mapped from Object Builder (user-defined business objects, local-only objects, non-IDL type objects).
- Attribute Specification, IDL page, length property is set if the attribute is of type string, and has associated size information.
- Attribute Specification, DDL page, IsIncludedInCopyHelper property is set if the attribute is part of the component's copy helper
- Attribute Specification, DDL page, PrimaryKey property is set if the attribute is part of the component's key.
- Attribute Specification, DDL page, IsIncludedInDataObject property is set if the attribute is part of the component's data object interface.
- Association Specification, IDL A/B pages, MapAsObjectRelationship property is set if an object relationship or sequence attribute in the business object interface was created by exporting the role of an association from Rose
- Association Specification, IDL A/B pages, RelationshipImplementation property is set if the object relationship has a selected implementation type in the business object implementation

In order to track changes between Component Broker objects and Rose elements, the import process uses the UUID of an element as an identifier. The UUID is stored as the uuid property of IDL page in Rose for each package, class, attribute, operation, and role of association.

RELATED CONCEPTS

"Rose" on page 64

"The Rose Bridge" on page 69

"Rose Bridge export" on page 79

RELATED TASKS

"Chapter 3. Using Rational Rose with Object Builder" on page 63

"Importing a project into Rose" on page 92

"Importing a Rose design from a team environment" on page 447

RELATED REFERENCES

“Object Builder to Rose mapping rules” on page 123

“Rose properties and bridging guidelines” on page 70

Importing a project into Rose

► CHANGED

You can import an Object Builder project into Rose. If the imported project was originally created by a Rose export, then the new Rose model created by the import will mirror the information in the original, exported Rose model’s Logical view. If your original model has additional information in other views, you can consolidate the two models (the original one, and the newly imported one) using the Rose Update feature.

If your Object Builder project was created directly in Object Builder (not by export from Rose), then the Rose model is based on default mappings of Object Builder elements to Rose elements.

Once the import is complete, you can work with the design in Rose, and export the changes back to Object Builder.

To import an Object Builder project into Rose, follow these steps:

1. Select **File > Import from Object Builder**. The Rose Bridge wizard opens.
Note: If you intend to replace the Rose model that you have currently loaded, you must exit from this model before running the bridge to enable the bridge to write the .mdl file successfully.
2. In the **Source Project** field, type the directory of the Object Builder project you are importing.
3. Click **Set** to add the directory to the import list.
4. Add any additional projects to the list in the same manner. You can import multiple projects to the same model.
Note: You can only import multiple projects if the source model and its associated .cat files have already been exported to multiple projects. You cannot import multiple projects to a new model file, or to an existing model file that has not been set up for multiple projects.
5. Click **Next**.
6. Enter the name of the new Rose model file you are importing to. If you know your project will be imported into a category file (.cat) on Rose, then you need to specify the virtual path mapping information by entering the symbol and actual path data. The project will be mapped to a .cat file with an equivalent top-level package in the Rose model file you specified.
7. Click **Finish**.

The projects you selected are imported into the Rose model file and .cat files you specified.

You have now imported an Object Builder project into a Rose model. If the imported project was created by export from Rose, and the original Rose model contains information in other views besides the Logical view, then you should consolidate the new model with the original model before doing any more design work.

To merge the new model with the original model, follow these steps:

1. Click **File > Open** to open the original Rose model.
2. Specify the original .mdl file and click **Open**.
3. Click **File > Update** to apply updates from the changed model.
4. Specify the updated .mdl file (created by the import in the previous task) and click **Open**.

Your model now contains the entire updated design, and you can continue your design work. Because the Rose Bridge preserves diagram information in the Logical View, you may have multiple diagrams with the same information after the update. This will cause no problems, but you can delete the duplicate diagrams if desired. No structural information will be lost by the deletion.

Note: If you have Rose 98i or Rose 2000, you can use the Model Integrator provided with Rose to merge the models. Please read the documentation provided with Rose for an explanation of this tool.

When you are ready to switch back to Object Builder, you can export the design back to Object Builder by selecting **File > Export to Object Builder**.

RELATED CONCEPTS

- “Object Builder” on page 1
- “Projects and models” on page 17
- “Rose” on page 64
- “The Rose Bridge” on page 69
- “Rose Bridge import” on page 89

RELATED TASKS

- “Exporting a design from Rose” on page 80
- “Importing a Rose design from a team environment” on page 447
- “Tutorial: Importing into Rose” on page 94

RELATED REFERENCES

- “Object Builder to Rose mapping rules” on page 123

Tutorial: Importing into Rose



Objectives

- To add attributes to a component in Object Builder.
- To edit an existing attribute in Object Builder.
- To apply the change to a Rose model.

Before you begin

This scenario is a continuation of the “Tutorial: Exporting from Rose” on page 81. You must complete the previous scenario before attempting this one.

You need the following installed on your system:

- A CB Server
- A CB Client
- CB tools, including Samples
- DB2 Universal Database
- DB2 SDK
-  VisualAge for C++ and (for Java applications) VisualAge for Java
-  IBM C and C++ Compilers for AIX V3.6.4 and (for Java applications) VisualAge for Java
- Rational Rose 98, Rose 98i, or Rose 2000

You need Rational Rose installed and set up to work with Object Builder, as described in the task “Setting up Rose 98” on page 65 or “Setting up Rose 98i and Rose 2000” on page 66.

For a complete list of prerequisites that you must install on your system, see the *Getting Started with Component Broker* documentation for the appropriate platform.

You should be familiar with the Component Broker programming model, as described in the *IBM Component Broker Programming Guide*. At minimum, you should have read chapter 1, the introduction to the programming model.

You should be familiar with Rational Rose. If you are not familiar with the tool, take the Rose tutorial included with the software.

You should be familiar with Object Builder. If you are not familiar with the tool, run through one of the introductory tutorials, for example, “Tutorial: Creating a component with transient data” on page 39.

Sample files

There are no equivalent samples for this exercise. The exercise assumes that you have created a project by exporting a design from Rose, as described in the previous tutorial.

Description

In this exercise, you will extend the Agent component created in the previous exercise, by adding an attribute `contactInfo`, and changing the name of the `id` attribute to `agentID`. You will then apply the changes to the original Rose model, by importing your Object Builder project into Rose. Once you are done, you can continue on to “Tutorial: Team development with Rose” on page 449, in which you add a second class with an object relationship, and export the two classes into separate interdependent projects.

For this exercise, you will complete the following tasks:

1. Open the project.
2. Edit the Agent component attributes.
3. Import the changes into Rose.

Opening the project

Open the project you created in the previous exercise:

1. Start Object Builder.
2. In the Open Project wizard, specify the project to open (for example, `e:\tutorials\roseagent\`)
3. Click **Finish**.

Adding and editing attributes

Add the two new attributes to the business object interface, and change the name of the existing key attribute:

1. Locate the Agent interface in the User-Defined Business Objects folder (defined under the LifeInsurance File and `sample6` module).
2. From the Agent interface’s pop-up menu, click **Properties** to open the Business Object Interface wizard.
3. Click the title and turn to the Attributes page.
4. Click **Add Another**.
5. Define an attribute named `contactInfo`, of type string, with size 100.
6. Click on the `id` attribute.
7. Type over its name, changing it to `agentID`.
8. Click **Refresh**.
9. Click **Finish**.

The new attribute is automatically added to the business object implementation, and the name change from `id` to `agentID` is applied automatically to the key, copy helper, and implementation.

Add the new attribute to the copy helper:

1. Locate the AgentCopy copy helper, under the Agent interface.
2. From AgentCopy's pop-up menu, click **Properties** to open the Copy Helper wizard.
3. Move contactInfo from the Business Object Attributes list to the Copy Helper Attributes list.
4. Click **Finish**.

Add the new attribute to the data object:

1. Locate the AgentBO business object implementation, under the Agent interface.
2. From AgentBO's pop-up menu, click **Properties** to open the Business Object Implementation wizard.
3. Click the title and turn to the Data Object Interface page.
4. Move contactInfo from the Business Object Attributes list to the State Data list.
5. Click **Finish**.

Save your changes and close Object Builder:

1. Click **File > Save**.
2. Click **File > Exit**.

You have made your changes to the project, saved them, and closed Object Builder. You are ready to import the project into Rose.

Importing the project into Rose

Import the changed Object Builder project to a new Rose model:

1. Start Rose.
2. Click **File > Import from Object Builder**. The Rose Bridge wizard opens.
3. In the **Source Project** field, type the Object Builder project directory path (for example e:\tutorials\roseagent\).
4. Click **Set** to add the project to the import list.
5. Click **Next**.
6. In the **Target Model** field, provide the path and name for the model you want to create (for example e:\tutorials\rosemodels\importagent.mdl).
7. Click **Finish**.

The Rose Bridge generates the udbo.LifeInsurance.xml file in the project's \Export directory, updates the agent.xml file in the project's \XMI directory with any project information it cannot preserve in the transfer, and then imports the two files to create a new model file.

Once the import is complete, load the newly generated file into Rose, and review the changes.

Reviewing the changes

Under the Logical View, you can see that Agent has a new attribute, `contactInfo`, and that the `id` attribute has become `agentID`.

Open the Attribute Specification notebook for `contactInfo`. On the DDL page, you can see that the Length property has the value 100.

Save and close the model.

Because the model in the previous scenario contained information only in the Logical View, with no additional diagrams, the new model can simply replace the previous model. However, if your original model had contained information in other views, or additional diagrams within the Logical view, you could consolidate that information with the newly imported model by using the the Rose Update feature, as described in the topic “Importing a project into Rose” on page 92.

Summary

You have changed your component in Object Builder, and then applied the changes to the component design in Rose. You can now continue working in Rose as part of the “Tutorial: Team development with Rose” on page 449, in which you add a second class with an object relationship, and export the two classes into separate interdependent projects.

Rose to Object Builder mapping rules



The following topics describe the way in which elements of your Rose design map to elements of Object Builder projects:

- “Projects in Rose” on page 98
- “Modules in Rose” on page 99
- “Constructs in Rose” on page 101
- “Class properties in Rose” on page 109
- “Package properties in Rose” on page 116
- “Attribute properties in Rose” on page 113
- “Method properties in Rose” on page 116
- “Class relationships in Rose” on page 119

RELATED CONCEPTS

“Rose” on page 64

“The Rose Bridge” on page 69

“Rose Bridge export” on page 79

RELATED TASKS

“Chapter 3. Using Rational Rose with Object Builder” on page 63

“Exporting a design from Rose” on page 80

“Exporting a Rose design to a team environment” on page 445

RELATED REFERENCES

“Object Builder to Rose mapping rules” on page 123

Projects in Rose

By default, your design in Rose maps to a single project in Object Builder, stored in the base directory you specify at export time. You can create more complex project divisions by dividing your design into separate .cat files.

You can assign a package and its content to a separate .cat file by selecting the package in a Class Diagram in the Logical View, and then clicking **File > Units > Control** *package*(where *package* is the name of the package you selected). You can then set a file name for the package’s associated .cat file.

Once you have assigned the package a .cat file, you can map it to a project directory using the IDL property `OBProjectDirectory`, in the package’s Specification notebook. Set the property to the project path you want to export to. You cannot enter a directory name that contains spaces.

When you export, you will get a separate project for every .cat file in your design (as specified with the `OBProjectDirectory` setting), plus a base project that contains the design elements in your main .mdl file.

For example:

A model consists of three base-level packages. The first contains two subpackages, the second contains a single subpackage, and the third contains only classes. In addition, there is a single class declared at the base level (outside of any packages).

- Package_A
 - Package_A1
 - Package_A2 (stored in PackageA2.cat, `OBProjectDirectory=e:\projectA2`)
 - Class_A3
- Package_B (stored in PackageB.cat, `OBProjectDirectory=e:\projectB`)

- Package_B1
- Package_C
- Class_D

The model is exported to the project directory e:\myprojects and results in the following project structure:

- e:\myprojects (contains Package_A::Package_A1, Package_C, and Class_D)
- e:\projectA2 (contains Package_A::Package_A2::Class_A3)
- e:\projectB (contains Package_B::Package_B1)

Package_A2 and Package_B get their own projects, because they are stored in .cat files that are separate from the rest of the model, and have been targeted for specific projects. The rest of the model is stored in the project specified at export time, e:\myprojects .

RELATED CONCEPTS

- “Projects and models” on page 17
- “Rose” on page 64

RELATED TASKS

- “Chapter 3. Using Rational Rose with Object Builder” on page 63

RELATED REFERENCES

- “Rose to Object Builder mapping rules” on page 97

Modules in Rose

The name scoping used by Component Broker is based on CORBA IDL, where a containment relationship exists among IDL files, modules, and interfaces. In the Rose model, a containment relationship exists among packages (categories), subpackages (subcategories), and classes (interfaces or data types). The export process from Rose supports the same containment relationship as Object Builder. For this to work, some restrictions on what gets mapped into Object Builder are necessary.

Packages that are associated with .cat files map to projects. This overrides any of the mappings shown below. A package is considered to be top-level if it is contained directly by the Logical View, or contained directly by a project package.

Non-project packages map as follows. Rules are given in order of precedence:

1. Top-level packages that contain interfaces map to files
2. Packages that contain both interfaces and packages map to files
3. Packages that contain interfaces but not packages map to modules

4. Packages that contain module packages map to files
5. Other packages are ignored
6. Interfaces not in a package map to a file and interface with the same name.

Example without project packages

Rose packages	Object Builder files and modules
<ul style="list-style-type: none"> • PackageA <ul style="list-style-type: none"> – ClassA1 • PackageB <ul style="list-style-type: none"> – ClassB1 – PackageB1 <ul style="list-style-type: none"> - ClassB2 • PackageC <ul style="list-style-type: none"> – PackageC1 <ul style="list-style-type: none"> - PackageC2 <ul style="list-style-type: none"> • ClassC3 • ClassD 	<ul style="list-style-type: none"> • File PackageA (rule 1, overriding rule 3) <ul style="list-style-type: none"> – Interface ClassA1 • File PackageB (rule 2) <ul style="list-style-type: none"> – Interface ClassB1 – Module PackageB1 (rule 3) <ul style="list-style-type: none"> - Interface ClassB2 • File PackageC1 (rule 4) <ul style="list-style-type: none"> – Module PackageC2 (rule 3) <ul style="list-style-type: none"> - Interface ClassC3 • File ClassD (rule 6) <ul style="list-style-type: none"> – Interface ClassD <p>Not mapped: PackageC (rule 5)</p>

Example with project packages

Rose packages	Object Builder files and modules
<ul style="list-style-type: none"> • PackageA <ul style="list-style-type: none"> – PackageA1 <ul style="list-style-type: none"> - PackageA2 <ul style="list-style-type: none"> • ClassA3 • PackageB <ul style="list-style-type: none"> – PackageB1 (with .cat file) <ul style="list-style-type: none"> - ClassB2 - PackageB2 <ul style="list-style-type: none"> • ClassB3 	<p>Main project:</p> <ul style="list-style-type: none"> • File PackageA1 (rule 4) <ul style="list-style-type: none"> – Module PackageA2 (rule3) <ul style="list-style-type: none"> - Interface ClassA3 <p>Project PackageB1:</p> <ul style="list-style-type: none"> • File ClassB2 (rule 6) <ul style="list-style-type: none"> – Interface ClassB2 • File PackageB2 (rule 1) <ul style="list-style-type: none"> – Interface ClassB3 <p>Not mapped: PackageA, PackageB (rule 5)</p>

RELATED CONCEPTS

“Rose” on page 64

RELATED TASKS

“Chapter 3. Using Rational Rose with Object Builder” on page 63

RELATED REFERENCES

“Rose to Object Builder mapping rules” on page 97

Constructs in Rose



You can specify constructs in Rose by defining classes with the IDLSpecificationType appropriate to the construct.

The IDLSpecificationType is set on the IDL page of the Class Specification notebook. By default, it is set to Interface (so the class will become a business object interface in Object Builder). You can change the default to one of the following types of constructs:

- Struct
- Enumeration
- Typedef
- Union
- Const
- Exception

The following table summarizes the properties of each construct type, the equivalent specifications for those properties in Rose, and the result when the design is exported to Object Builder.

Element (Object Builder)	Specification (Rose)	Description
Struct	Class Specification notebook IDL page IDLSpecificationType=Struct	Becomes a structure with either file scope (if contained in a top-level package) or module scope (if contained in a nested package). Attributes of the class map to members of the struct.
Name	Class Specification notebook General page Name field	Becomes the name of the structure. Should be a valid C++ name. Leading and trailing blank spaces are removed, and embedded spaces are converted to underscores.

Element (Object Builder)	Specification (Rose)	Description
Object Builder type	Class Specification notebook IDL page ObjectType property	Determines whether the construct will be created in the User-Defined Business Objects or Local-Only Objects section of the Object Builder project.
Hide from export	Class Specification notebook IDL page BridgeToOB property	Set to False to hide the construct from the export process. The construct will not be exported to Object Builder. By default, the BridgeToOB property is set to True, and the construct is exported.
Member names	Attribute Specification notebook General page Name field	Should be a valid C++ name. Leading and trailing blank spaces are removed, and embedded spaces are converted to underscores.
Member types	Attribute Specification notebook General page Type field	Should be a valid type: either a predefined IDL type (for example, char, short, float), a type currently defined in Rose, or a type already defined in the Object Builder model you are exporting to. When you specify the type, any leading and trailing blank spaces are removed, and embedded spaces are converted to underscores.

Element (Object Builder)	Specification (Rose)	Description
Enumeration	Class Specification notebook IDL page IDLSpecificationType=Enumeration	Becomes an enumeration with either file scope (if contained in a top-level package) or module scope (if contained in a nested package). Attributes of the class map to members of the enumeration.
Name	Class Specification notebook General page Name field	Becomes the name of the struct. Should be a valid C++ name. Leading and trailing blank spaces are removed, and embedded spaces are converted to underscores.
Object Builder type	Class Specification notebook IDL page ObjectType property	Determines whether the construct will be created in the User-Defined Business Objects or Local-Only Objects section of the Object Builder project.
Hide from export	Class Specification notebook IDL page BridgeToOB property	Set to False to hide the construct from the export process. The construct will not be exported to Object Builder. By default, the BridgeToOB property is set to True, and the construct is exported.
Member names	Attribute Specification notebook General page Name field	Should be a valid C++ name. Leading and trailing blank spaces are removed, and embedded spaces are converted to underscores.

Element (Object Builder)	Specification (Rose)	Description
Member types	Attribute Specification notebook General page Type field	Should be a valid type: either a predefined IDL type (for example, char, short, float), a type currently defined in Rose, or a type already defined in the Object Builder model you are exporting to. When you specify the type, any leading and trailing blank spaces are removed, and embedded spaces are converted to underscores.
Typedef	Class Specification notebook IDL page IDLSpecificationType= Typedef ImplementationType= <i>type</i>	Becomes a typedef with either file scope (if contained in a top-level package) or module scope (if contained in a nested package).
Name	Class Specification notebook General page Name field	Becomes the name of the typedef. Should be a valid C++ name. Leading and trailing blank spaces are removed, and embedded spaces are converted to underscores.
Object Builder type	Class Specification notebook IDL page ObjectType property	Determines whether the construct will be created in the User-Defined Business Objects or Local-Only Objects section of the Object Builder project.

Element (Object Builder)	Specification (Rose)	Description
Type	Class Specification notebook IDL page ImplementationType= <i>type</i>	Determines what type the typedef is for. Should be a valid type: either a predefined IDL type (for example, char, short, float), a type currently defined in Rose, or a type already defined in the Object Builder model you are exporting to. When you specify the type, any leading and trailing blank spaces are removed, and embedded spaces are converted to underscores.
Hide from export	Class Specification notebook IDL page BridgeToOB property	Set to False to hide the construct from the export process. The construct will not be exported to Object Builder. By default, the BridgeToOB property is set to True, and the construct is exported.
Union	Class Specification notebook IDL page IDLSpecificationType=Union	Becomes a union with either file scope (if contained in a top-level package) or module scope (if contained in a nested package).
Name	Class Specification notebook General page Name field	Becomes the name of the union. Should be a valid C++ name. Leading and trailing blank spaces are removed, and embedded spaces are converted to underscores.

Element (Object Builder)	Specification (Rose)	Description
Object Builder type	Class Specification notebook IDL page ObjectType property	Determines whether the construct will be created in the User-Defined Business Objects or Local-Only Objects section of the Object Builder project.
Type	Class Specification notebook IDL page ImplementationType= <i>type</i>	Determines the type of the union switch. Should be a valid type: either a predefined IDL type (for example, char, short, float), a type currently defined in Rose, or a type already defined in the Object Builder model you are exporting to. When you specify the type, any leading and trailing blank spaces are removed, and embedded spaces are converted to underscores.
Hide from export	Class Specification notebook IDL page BridgeToOB property	Set to False to hide the construct from the export process. The construct will not be exported to Object Builder. By default, the BridgeToOB property is set to True, and the construct is exported.
Const	Class Specification notebook IDL page IDLSpecificationType=Const	Becomes a const with either file scope (if contained in a top-level package) or module scope (if contained in a nested package).
Name	Class Specification notebook General page Name field	Becomes the name of the const. Should be a valid C++ name. Leading and trailing blank spaces are removed, and embedded spaces are converted to underscores.

Element (Object Builder)	Specification (Rose)	Description
Object Builder type	Class Specification notebook IDL page ObjectType property	Determines whether the construct will be created in the User-Defined Business Objects or Local-Only Objects section of the Object Builder project.
Type	Class Specification notebook IDL page ImplementationType= <i>type</i>	Determines the type of the const. Should be a valid type: either a predefined IDL type (for example, char, short, float), a type currently defined in Rose, or a type already defined in the Object Builder model you are exporting to. When you specify the type, any leading and trailing blank spaces are removed, and embedded spaces are converted to underscores.
Value	Class Specification notebook IDL page ConstValue= <i>value</i>	Determines the value of the const.
Hide from export	Class Specification notebook IDL page BridgeToOB property	Set to False to hide the construct from the export process. The construct will not be exported to Object Builder. By default, the BridgeToOB property is set to True, and the construct is exported.
Exception	Class Specification notebook IDL page IDLSpecificationType=Exception	Becomes an exception with either file scope (if contained in a top-level package) or module scope (if contained in a nested package). Attributes of the class map to members of the exception.

Element (Object Builder)	Specification (Rose)	Description
Name	Class Specification notebook General page Name field	Becomes the name of the exception. Should be a valid C++ name. Leading and trailing blank spaces are removed, and embedded spaces are converted to underscores.
Object Builder type	Class Specification notebook IDL page ObjectType property	Determines whether the construct will be created in the User-Defined Business Objects or Local-Only Objects section of the Object Builder project.
Hide from export	Class Specification notebook IDL page BridgeToOB property	Set to False to hide the construct from the export process. The construct will not be exported to Object Builder. By default, the BridgeToOB property is set to True, and the construct is exported.
Member names	Attribute Specification notebook General page Name field	Should be a valid C++ name. Leading and trailing blank spaces are removed, and embedded spaces are converted to underscores.
Member types	Attribute Specification notebook General page Type field	Should be a valid type: either a predefined IDL type (for example, char, short, float), a type currently defined in Rose, or a type already defined in the Object Builder model you are exporting to. When you specify the type, any leading and trailing blank spaces are removed, and embedded spaces are converted to underscores.

RELATED CONCEPTS

“Projects and models” on page 17

“Rose” on page 64

RELATED TASKS

“Chapter 3. Using Rational Rose with Object Builder” on page 63

RELATED REFERENCES

“Rose to Object Builder mapping rules” on page 97

Class properties in Rose



When you define a class in Rose, it can be mapped either to a business object interface, a local-only object, a non-IDL type object, or to a type of construct in Object Builder.

For constructs, the name, documentation, and attributes (when appropriate) are preserved by the export process. For business object interfaces, both class properties and class relationships are preserved. For local-only objects, appropriate class properties are preserved.

A class is exported as a business object interface by default, based on the setting of the `IDLSpecificationType` (which is set to `Interface` by default). The `IDLSpecificationType` is set on the IDL page of the Class Specification notebook. You can set additional properties of the class and of its attributes to create additional component objects for the class (such as the business object implementation) when you export.

The properties of constructs, class attributes, class methods or operations, and class relationships are described in their own topics.

The following table describes the properties that are in effect for a class (excluding its attributes, methods, and relationships) whose `IDLSpecificationType` is set to `Interface`.

Component element (Object Builder)	Specification (Rose)	Description
Business object interface	Class Specification notebook IDL page IDLSpecificationType=Interface ObjectType=User-Defined Business Objects	Becomes a business object interface with either file scope (if contained in a top-level package) or module scope (if contained in a nested package). Attributes and operations of the class map to attributes and methods of the component.
Local-only object	Class Specification notebook IDL page IDLSpecificationType=Interface ObjectType=Local-Only Objects	Becomes a local-only object with either file scope (if contained in a top-level package) or module scope (if contained in a nested package). Attributes and operations of the class map to attributes and methods of the component.
Non-IDL object	Class Specification notebook IDL page IDLSpecificationType=Interface ObjectType=Non-IDL Type Objects	Becomes a non-IDL object. No other class properties apply to this object.
Name	Class Specification notebook General page Name field	Becomes the name of the interface. Should be a valid C++ name. Leading and trailing blank spaces are removed, and embedded spaces are converted to underscores. For example, My Class Name would become My_Class_Name. Also becomes the basis for the names of any additional component objects that are created during the export.

Component element (Object Builder)	Specification (Rose)	Description
Comments	Class Specification notebook General page Documentation field	Any documentation you enter for the class becomes comments in Object Builder, where they can be accessed from the last page of the Business Object Interface wizard. Do not include <code>/*</code> or <code>*/</code> in the documentation text: the generated code from Object Builder will provide C++ comment tags around your entries by default.
Queryable (True False)	Class Specification notebook IDL page IsQueryable property	Defines whether the specified interface is queryable. Does not apply to local-only non-IDL objects.
Business object implementation and data object interface	Class Specification notebook IDL page CreateImplementation property	Defines whether a business object implementation, and its accompanying data object interface, will be created for the current interface. The data object interface will include any attributes that have the DDL property <code>IsIncludedInDataObject</code> set to true. By default, all public attributes of the class are included in the data object interface. Does not apply to local-only non-IDL objects.

Component element (Object Builder)	Specification (Rose)	Description
Key	Class Specification notebook IDL page CreateKey property	<p>Defines whether a key will be created for the current interface. The key will include any attributes that have the PrimaryKey property set to True on the DDL page of their Attribute Specification notebooks. By default, no attributes are included in the key.</p> <p>When you set the CreateKey property to true, you should also set the PrimaryKey property to true for at least one of the interface's public attributes.</p> <p>Does not apply to local-only non-IDL objects.</p>
Copy helper	Class Specification notebook IDL page CreateCopyHelper property	<p>Defines whether a copy helper will be created for the current interface. The copy helper will include any attributes that have the IsIncludedInCopyHelper property set to true on the DDL page of their Attribute Specification notebooks. By default, all public attributes of the class are included in the copy helper.</p> <p>Does not apply to local-only non-IDL objects.</p>

Component element (Object Builder)	Specification (Rose)	Description
Hide from export	Class Specification notebook IDL page BridgeToOB property	Set to False to hide the class from the export process. The interface, and its attendant component objects, will not be exported to Object Builder. By default, the BridgeToOB property is set to True, and the class is exported.

RELATED CONCEPTS

“Projects and models” on page 17

“Rose” on page 64

RELATED TASKS

“Chapter 3. Using Rational Rose with Object Builder” on page 63

RELATED REFERENCES

“Rose to Object Builder mapping rules” on page 97

Attribute properties in Rose

The following attribute properties apply when the class in Rose has its IDLSpecificationType set to Interface (the default, on the Class Specification notebook’s IDL page).

Attribute property (Object Builder)	Specification (Rose)	Description
Name	Attribute Specification notebook General page Name field	Should be a valid C++ name. Leading and trailing blank spaces are removed, and embedded spaces are converted to underscores. For example, my Data Name would become my_Data_Name.

Attribute property (Object Builder)	Specification (Rose)	Description
Type	Attribute Specification notebook General page Type field	Should be a valid type: either a predefined IDL type (for example, char, short, float), a type currently defined in Rose, or a type already defined in the Object Builder model you are exporting to. When you specify the type, any leading and trailing blank spaces are removed, and embedded spaces are converted to underscores.
Initializer	Attribute Specification notebook General page Initial Value field	Value placed in this field is transferred to the Initializer field for the attribute in Object Builder. This value should be consistent with the type defined for the attribute.
Access control	Attribute Specification notebook General page Export Control options	Can be one of public, protected, or private, and maps as follows: <ul style="list-style-type: none"> • Public attributes map to attributes of the business object interface. The business object implementation will have get and set methods defined for these attributes. • Protected attributes map to protected attributes of the business object implementation. They do not appear in the business object interface, and will not be exported at all if the class's CreateImplementation property is set to false. • Private attributes map to private attributes of the business object implementation. They do not appear in the business object interface, and will not be exported at all if the class's CreateImplementation property is set to false.
Length	Attribute Specification notebook DDL page Length property	Defines the string length if the attribute is of type string.

Attribute property (Object Builder)	Specification (Rose)	Description
Key attribute	Attribute Specification notebook DDL page PrimaryKey property	<p>Defines whether the attribute is included in the key object for the component (if you set the CreateKey property for the class). If you set this property, you should also set the IsReadOnly property on the IDL page (key attributes must be read-only).</p> <p>Key attributes for parents will automatically be included in the child's key.</p> <p>Note: Do not specify complex types (such as structures or unions) as keys.</p>
Copy helper attribute	Attribute Specification notebook DDL page IsIncludedInCopyHelper property	<p>Defines whether the attribute is included in the copy helper for the component (if you set the CreateCopy property for the class). By default, all public attributes are included in the copy helper.</p>
Data object attribute	Attribute Specification notebook DDL page IsIncludedInDataObject property	<p>Defines whether the attribute is included in the data object interface for the component (if you set the CreateImplementation property for the class). By default, all attributes are included in the data object.</p>
Read-only	Attribute Specification notebook IDL page IsReadOnly property	<p>Defines whether the attribute is read-only. By default the attribute is not read-only.</p>
Override		<p>If you define an attribute in a child class that has the same name and type as an attribute in its parent class, the attribute will be defined as an override in Object Builder, in the Business Object Implementation wizard, Attributes to Override page.</p>

RELATED CONCEPTS

"Projects and models" on page 17

"Rose" on page 64

RELATED TASKS

"Chapter 3. Using Rational Rose with Object Builder" on page 63

RELATED REFERENCES

“Rose to Object Builder mapping rules” on page 97

Package properties in Rose

The following package properties apply for packages in your Rose model.

Package property (Object Builder)	Specification (Rose)	Description
Name	Package Specification Notebook Name field	Name which will be used for the association Object Builder file or module.
Name of associated Object Builder project	Package Specification Notebook OBProjectDirectory field	If the package is identified as a controlled unit in Rose, the value of this field will be the name of the Object Builder project created to hold the information being mapped from Rose.
Hide from export	Package Specification Notebook BridgeToOB property	If this value is set to true for a package in the Rose model, this package and all elements (packages and classes) contained in it will not be mapped from Rose to Object Builder.

RELATED CONCEPTS

“Projects and models” on page 17

“Rose” on page 64

RELATED TASKS

“Chapter 3. Using Rational Rose with Object Builder” on page 63

RELATED REFERENCES

“Rose to Object Builder mapping rules” on page 97

Method properties in Rose

When a class in Rose has its IDLSpecificationType set to Interface (the default, on the Class Specification notebook’s IDL page), then the class’s operations map to methods of the component in Object Builder. The properties of a method can be set in Rose as follows.

Method property (Object Builder)	Specification (Rose)	Description
Name	Operation Specification notebook General page Name field	Should be a valid C++ name. Leading and trailing blank spaces are removed, and embedded spaces are converted to underscores. For example, my Method Name would become my_Method_Name.
Return type	Operation Specification notebook General page Return Class field	Maps to the return type for the method. Should be a valid type: either a predefined IDL type (for example, char, short, float), a type currently defined in Rose, or a type already defined in the Object Builder model you are exporting to. When you specify the type, any leading and trailing blank spaces are removed, and embedded spaces are converted to underscores.
Access control	Operation Specification notebook General page Export Control options	Can be one of public, protected, or private, and maps as follows: <ul style="list-style-type: none"> • Public operations map to methods of the business object interface. • Protected operations map to protected methods of the business object implementation. They do not appear in the business object interface, and will not be exported at all if the class's CreateImplementation property is set to false. • Private operations map to private methods of the business object implementation. They do not appear in the business object interface, and will not be exported at all if the class's CreateImplementation property is set to false.
Parameter name	Operation Specification notebook Details page Arguments area	The argument name maps to the parameter name. Should be a valid C++ name. Leading and trailing blank spaces are removed, and embedded spaces are converted to underscores. For example, first Parameter would become first_Parameter.

Method property (Object Builder)	Specification (Rose)	Description
Parameter type	Operation Specification notebook Details page Arguments area	The argument type maps to the parameter type. Should be a valid type: either a predefined IDL type (for example, char, short, float), a type currently defined in Rose, or a type already defined in the Object Builder model you are exporting to. When you specify the type, any leading and trailing blank spaces are removed, and embedded spaces are converted to underscores.
Argument default	Operation Specification notebook Details page Arguments area	Should be one of: <ul style="list-style-type: none"> • in • out • inout If you specify a different value, it will be ignored, and in will be used.
Exceptions	Operation Specification notebook Details page Exceptions field	Map to exceptions raised by the method. You can specify these exceptions in Rose on the Details page of the Operation Specification notebook. The exceptions should be of a valid type (either one defined in the current Rose model, or one previously defined in the target Object Builder model).
One-way	Operation Specification notebook IDL page OperationIsOneWay property	Maps to the one-way property in Object Builder. The property defaults to False.
Override		If you define an operation in a child class that has the same name and signature as an operation in its parent class, the operation will be defined as an override in Object Builder, in the Business Object Implementation wizard, Methods to Override page.

RELATED CONCEPTS

“Projects and models” on page 17

“Rose” on page 64

RELATED TASKS

“Chapter 3. Using Rational Rose with Object Builder” on page 63

RELATED REFERENCES

“Rose to Object Builder mapping rules” on page 97

Class relationships in Rose

▶ CHANGED

When you export your object model from Rose into Object Builder, the class relationships you have defined are mapped as follows:

Inheritance

Inheritance relationships you define in Rose are preserved by the export process, and applied to the business object interfaces that the exported classes are mapped to. If you are generating additional component objects for a class (an option of the export process), then the inheritance for the additional components parallels the inheritance for the business object interface.

For example, if ChildClass inherits from ParentClass, then after the export the ChildClass business object interface inherits from the ParentClass business object interface. If you added business object implementations during the export, then in addition the ChildClassBO business object implementation inherits from the ParentClassBO business object implementation.

Associations and aggregations

Associations and aggregations map to attributes, object relationships, or sequences, as explained below. Associations and aggregations are only mapped if they are navigable.

The export process preserves or maps the following information about the relationship:

Relationship property (Object Builder)	Specification (Rose)	Description
Name	Association Specification notebook Role A General page, Role B General page Name field	<p>Role A and Role B are the terms in Rose that define the two ends of an association. In the Association Specification notebook, the names you specify for the roles (on the Role A General and Role B General pages) determine the names of the attributes or relationships that each class has to represent its association with the other.</p> <p>The names you specify should be valid C++ names. Leading and trailing blank spaces are removed, and embedded spaces are converted to underscores. If you do not specify names for the roles, then default names based on the names of the referenced interfaces are used.</p> <p>For example, if a class named Agent is in Role A and a class named Customer is in Role B, the relationship is 1..1 and no names are specified, then Agent gets an attribute named the_Customer of type Customer, and Customer gets an attribute named the_Agent of type Agent.</p>
Cardinality	Association Specification notebook Role A Detail page, Role B Detail page Cardinality field	<p>If the cardinality is set to one of 0..n, 1..n, or n, then it is considered to be a cardinality of 'many', and the relationship will be mapped to either an object relationship (stored in a reference collection) or an attribute of type sequence <i>ClassName</i> (where <i>ClassName</i> is the name of the class in the n role). With all other cardinalities, the relationship will be mapped to attributes of type <i>ClassName</i>.</p> <p>When the cardinality is 'many', you can choose whether to map as an object relationship or a sequence with the MapAsObjectRelationship property.</p>

Relationship property (Object Builder)	Specification (Rose)	Description
Relationship mapping	Association Specification notebook IDL A page, IDL B page MapAsObjectRelationship property	For class relationships with role cardinality set to 'many', the MapAsObjectRelationship property defines whether the class relationship is exported as an object relationship or a sequence. By default, relationships are exported as object relationships. To export a relationship as a sequence, set the MapAsObjectRelationship property to false on the IDL A or IDL B page of the Association Specification notebook.
Relationship implementation	Association Specification notebook IDL A page, IDL B page Relationship-Implementation property	<p>If the class relationship has been set to export as an object relationship (MapAsObjectRelationship set to true in the appropriate IDL A or IDL B page of the association notebook), you can specify the implementation type for this object relationship. The RelationshipImplementation property on the IDL A or IDL B page of the Association Specification notebook can be set to one of the three following values:</p> <ul style="list-style-type: none"> • Local Persistent Reference • Transient Reference Collection • Persistent Reference Collection • User-defined OO_SQL Query • Reference Resolved by Foreign Key <p>This property applies only when the class property CreateImplementation is set to True (the default).</p>

Relationship property (Object Builder)	Specification (Rose)	Description
Read-only or read-write	Association Specification notebook IDL A page, IDL B page IsReadOnly property	<p>You can specify that a role in an association be read-only. When the association is exported, then any corresponding attribute is marked accordingly. You can specify whether an attribute is read-only on the appropriate page for the role (IDL A and IDL B). You can set the IsReadOnly property on these pages to true or false. By default, the property is set to false (attributes have read/write access).</p> <p>For example, if Role A (Agent) is read-only, then Role B's attribute (Customer's attribute the_Agent of type Agent) is read-only.</p>
Access control	Association Specification notebook Role A General page, Role B General page Export Control options	<p>If the relationship is being exported as an attribute, then the access control you set is applied. For example, if Agent's role is marked protected, then Customer's attribute of type Agent will be protected. The access control can be one of public, protected, or private, and maps as follows:</p> <ul style="list-style-type: none"> • Public attributes map to attributes of the business object interface. The business object implementation will have get and set methods defined for these attributes. • Protected attributes map to protected attributes of the business object implementation. They do not appear in the business object interface. • Private attributes map to private attributes of the business object implementation. They do not appear in the business object interface.

RELATED CONCEPTS

"Projects and models" on page 17

"Rose" on page 64

RELATED TASKS

“Chapter 3. Using Rational Rose with Object Builder” on page 63

RELATED REFERENCES

“Rose to Object Builder mapping rules” on page 97

Object Builder to Rose mapping rules

▶ CHANGED

When you import an Object Builder model into Rose, elements in the Object Builder model map to elements in a Rose model as follows:

- Business object files, modules, and interfaces that already have a mapping (because they were created by export from Rose) maintain that mapping.
- New business object files, modules, and interfaces (added directly to Object Builder, not by export from Rose) are mapped to packages, subpackages, and classes.
- Local-only objects that already have a mapping (because they were created by export from Rose) maintain that mapping.
- New local-only objects (added directly to Object Builder, not by export from Rose) are mapped to classes with the `ObjectType` property set to **Local-Only Objects**.
- Non-IDL objects that already have a mapping (because they were created by export from Rose) maintain that mapping.
- New non-IDL objects (added directly to Object Builder, not by export from Rose) are mapped to classes with the `ObjectType` property set to **Non-IDL Type Objects**.

Business object file

Business object files in Object Builder that already have a mapping (because they were created by export from Rose) maintain that mapping. New business object files are mapped to packages. Properties of the module map as follows:

- **Name**
Stored as the subpackage name in the Package Specification notebook.
- **Constructs**
Constructs defined with file scope map to top-level classes (for an existing mapping) or classes in the package (for new mappings). The `IDLSpecification` property of the class on the IDL page of its Class Specification notebook is set to one of:
 - Struct
 - Enumeration
 - TypeDef
 - Union
 - Const

- Exception
- **Comments**
Stored as documentation for the subpackage in the Package Specification notebook.

Business object module

Business object modules in Object Builder that already have a mapping (because they were created by export from Rose) maintain that mapping. New business object modules are mapped to subpackages of the file package. Properties of the module map as follows:

- **Name**
Stored as the subpackage name in the Package Specification notebook.
- **Constructs**
Constructs with module scope map to classes contained in the subpackage. The IDLSpecification property of the class on the IDL page of its Class Specification notebook is set to one of:
 - Struct
 - Enumeration
 - TypeDef
 - Union
 - Const
 - Exception
- **Comments**
Stored as documentation for the subpackage in the Package Specification notebook.

Business object interface

Business object interfaces in Object Builder that already have a mapping (because they were created by export from Rose) maintain that mapping. New business object interfaces are mapped to classes in a file package or module subpackage. The IDLSpecification property of the class on the IDL page of its Class Specification notebook is set to Interface. Properties of the interface map to properties of the class as follows:

- **Name**
Stored as the class name in the Class Specification notebook.
- **Constructs**
Constructs with interface scope map to nested classes inside the Rose class corresponding to the Object Builder interface. The Class Specification Notebook for the nested class captures the information for the construct type as described above for constructs at file or module scope.
- **Interface inheritance**
Parent interfaces map to generalize elements in Rose, on the Relations page of the Class Specification notebook.

- **Attributes**

Attributes map to attribute elements of the class in Rose.

- **Sequence attributes**

If an attribute of type sequence was created in Object Builder by the Rose Bridge (by the export of an association with the MapAsObjectRelationship property set to false), then the import preserves this original mapping. If the attribute was created directly in Object Builder (by defining a typedef for the sequence and using the typedef as the attribute type), it maps to an attribute of the class.

Note: If you defined the association in Rose, exported to Object Builder to create the sequence attribute, and then deleted the attribute in Object Builder, the import will **not** delete the association in Rose, but will set the `is_navigable` property of the Role to FALSE.

- **Methods**

Methods map to operation elements of the class in Rose.

- **Object relationships**

If an object relationship was created by exporting an association in Rose, the import preserves the original mapping, to a Role in a many-to-many association or one-to-many association in Rose. If the object relationship was created directly in Object Builder and does not have an equivalent in Rose, it is not imported, but is stored in the .xml file in the project's \XMI directory.

Note: If the object relationship was created directly in Object Builder and does not have an equivalent in Rose, a unidirectional association will be created in Rose to represent the relationship.

- **Comments**

Stored as class documentation in class specification notebook.

RELATED CONCEPTS

“Object Builder” on page 1

“Rose” on page 64

“The Rose Bridge” on page 69

RELATED TASKS

“Importing a project into Rose” on page 92

“Importing a Rose design from a team environment” on page 447

Chapter 4. Creating a component

Components can be defined either in Rational Rose as part of a design that you can export to Object Builder, or directly in Object Builder.

In Object Builder, components can be defined in any of the following ways:

- From a new design (starting from the component interface and working down to the component datastore)
- From an existing datastore (starting from the datastore and working up to the component interface)
- From both (combining a new component and data interface with an existing datastore)

The choices for component creation can be summarized as:

- “Creating a component for transient data” on page 135
- “Creating a component for new DB data” on page 136
- “Creating a component for existing DB data” on page 139
- “Creating a component for PA data” on page 157
- “Creating a component for an inbound message” on page 187
- “Creating a component for an outbound message” on page 201
- “Reusing existing objects” on page 215
- “Mapping a data object to a persistent object” on page 703
- “Creating a composite component” on page 261
- “Creating a local-only object” on page 216
- “Chapter 8. Working with enterprise beans” on page 391

RELATED CONCEPTS

“Object Builder” on page 1

“Rose” on page 64

Components (*Programming Guide*)

Component assembly (*Programming Guide*)

RELATED TASKS

“Developing in Object Builder” on page 19

“Configuring builds” on page 549

RELATED REFERENCES

“Naming objects” on page 128

“Internationalization of data” on page 132

Naming objects




The following tables show the rules for naming objects in Object Builder. You can also refer to the supported characters for the “Internationalization of data” on page 132.

Note: CORBA does not allow the identifier of an object to be the same as that of its immediate parent in the namespace. For example, a business object, local-only or data object interface cannot have the same name as the module that contains it.

Objects for DB persistence

Entity	Naming rules	Maximum length of name
database (SM object)	Characters permitted: alphanumeric characters (a-z, A-Z, 0-9), #, @, \$. Characters not permitted: from European, Asian character sets (for example, umlauts) The first character of the name must be an alphabetic character, or one of #, @, or \$. The names are not case-sensitive.	eight characters
schema group	Characters permitted: alphanumeric characters (letters and numbers) the underscore character the blank character	
DB schema file	Characters permitted: alphanumeric characters, the underscore character Name must start with an alphabet.	
DB persistent object file	File names must be unique (ignoring capitalization) to prevent files overwriting each other when generated on case-insensitive platforms. Do not specify an extension. The extension is supplied by the code generation process.	
DB persistent object class		390 * eight characters
DB persistent object attribute name		26 characters.

*  390 On OS/390, if the length of the DB persistent object class exceeds eight characters, Object Builder truncates it to the 8.3 format.

Length restrictions for the various backend stores

Entity	DB2 v5.2 and v6.1	Oracle	Informix	Units
database name	8	8	8	characters
schema name	8	8	8	characters
table name	18 (128 on v6.1 for workstation platforms)	30	18	bytes
column name	18 (30 on v6.1 for workstation platforms)	30	18	bytes

Note the following points:

- Workstation platforms include Windows NT, AIX, Solaris, and HP-UX (not OS/390).
- When the unit of length is x bytes, it implies x SBCS characters, or half the number (x/2) of DBCS characters.

Component objects

Entity	Characters permitted in name	Naming rules
business object: interface file, implementation file	alphanumeric characters the underscore character	Name must start with an alphabet
local-only object: file, interface		File names must be unique (ignoring capitalization) to prevent files overwriting each other when generated on case-insensitive platforms.
*data object file		Do not specify an extension. The extension is supplied by the code generation process.
*data object interface		
*data object implementation file		
*data object implementation interface		
key class file, interface, and module		
**copy helper class file, interface, and module		
managed object file (SM object)		
specialized home: implementation file, module, interface		
composition interface file		
composition module	Name must start with an alphabet, and cannot be an IDL reserved word.	

*Entities that exist after you create a data object from a persistent object.




**A copy helper attribute name is not case-sensitive, and cannot be the same as the name of the copy helper it is defined in.

Note:For interfaces and modules of business objects, data objects, and local-only objects:

- You cannot use the following keywords to name the interface:
 - Java keywords
 - IDL keywords
- None of the following names can be used as interface names:
 - any method name in `Java.lang.Object`. These include names such as *clone*, *finalize*, *hashCode*, *notifyAll*, *wait*, *equals*, *getClass*, *notify*, and *toString*
 - any name that is suffixed with *Package*, *Holder*, *Helper*, *Ref*, *_var*, or *_ptr*

- goto

Configuration objects

Entity	SM object	Characters permitted in name	Characters not permitted	Naming rules	Maximum length of name
container	Yes	alphanumeric characters (0-9 and A-Z)	special characters	The name must start with an alphabet.	
application family	Yes	alphanumeric characters		The name must start with an alphabet	
application					
DDL	Yes			 390 First character must be alphabetic; other characters (positions 2 to 8) must be alphanumeric.	 390 Cannot exceed eight characters.
IR file	Yes				 390 Cannot exceed eight characters
DLL description		alphanumeric characters (letters and numbers), the blank character		The name must start with an alphabet	
factory finder		The name must exist in the namespace at the time the application is run.			
client DLL	Yes		whitespace and Unix glob characters (special characters)	Operating system-compliant filenames	
server DLL					

Note the following points for an IR file:

► 390 Default name = business object file name with suffix `_IR`.

► WIN ► AIX ► SOLARIS ► HP-UX Default name = managed object IDL file name, with suffix `_IR`.

► WIN This file gets the `.exe` extension only on Windows NT.

RELATED CONCEPTS

Object Builder

Naming conventions (*Programming Guide*)

RELATED TASKS

“Searching the Tasks and Objects pane” on page 30

RELATED REFERENCES

“Internationalization of data”

Internationalization of data

► NEW

Objects that you create in Object Builder (for example, the application family, the application, managed objects, DLL names, application DDL files, and so on), along with some text string properties, (for example, database names), are ultimately loaded into System Management (installed on the server), when you deploy your applications. This data, along with data that is input through the System Manager user interface constitute system management configuration data.

Component Broker provides internationalized support for client-server and server-server communication between different code pages.

For error-free communication between different code pages, you must use a restricted set of ASCII characters for object names and attributes in system management configuration data.

Supported characters for input through the System Manager user interface are shown in the table. This constitutes the set of characters that are supported for Object Builder entities as well. (You must use characters only from the displayable portion of the invariant ASCII character set.)

For other rules to be followed while naming different entities in Object Builder, see “Naming objects” on page 128.

Table 1. System Manager user interface, allowed characters from ISO 8859-1 (Latin 1)

Char	Value : description
!	0x20 : space
“	0x21 : exclamation mark
#	0x22 : quotation mark
\$	0x23 : number sign (hash)
%	0x24 : dollar sign
%	0x25 : percent sign
&	0x26 : ampersand
'	0x27 : apostrophe
(0x28 : left parenthesis
)	0x29 : right parenthesis
*	0x2A : asterisk
+	0x2B : plus sign
,	0x2C : comma
-	0x2D : hyphen (minus)
.	0x2E : full stop (period)
/	0x2F : solidus (forward slash)
0	0x30 : digit zero
1	0x31 : digit one
2	0x32 : digit two
3	0x33 : digit three
4	0x34 : digit four
5	0x35 : digit five
6	0x36 : digit six
7	0x37 : digit seven
8	0x38 : digit eight
9	0x39 : digit nine
:	0x3A : colon
;	0x3B : semicolon
<	0x3C : less-than sign
=	0x3D : equals sign
>	0x3E : greater-than sign
?	0x3F : question mark
@	0x40 : commercial at
A	0x41 : latin capital letter A
B	0x42 : latin capital letter B
C	0x43 : latin capital letter C
D	0x44 : latin capital letter D
E	0x45 : latin capital letter E
F	0x46 : latin capital letter F
G	0x47 : latin capital letter G

Table 1. System Manager user interface, allowed characters from ISO 8859-1 (Latin 1) (continued)

Char	Value : description
H	0x48 : latin capital letter H
I	0x49 : latin capital letter I
J	0x4A : latin capital letter J
K	0x4B : latin capital letter K
L	0x4C : latin capital letter L
M	0x4D : latin capital letter M
N	0x4E : latin capital letter N
O	0x4F : latin capital letter O
P	0x50 : latin capital letter P
Q	0x51 : latin capital letter Q
R	0x52 : latin capital letter R
S	0x53 : latin capital letter S
T	0x54 : latin capital letter T
U	0x55 : latin capital letter U
V	0x56 : latin capital letter V
W	0x57 : latin capital letter W
X	0x58 : latin capital letter X
Y	0x59 : latin capital letter Y
Z	0x5A : latin capital letter Z
[0x5B : left square bracket
\	0x5C : reverse solidus (reverse slash)
]	0x5D : right square bracket
^	0x5E : circumflex accent
_	0x5F : low line (underscore)
`	0x60 : grave accent
a	0x61 : latin small letter a
b	0x62 : latin small letter b
c	0x63 : latin small letter c
d	0x64 : latin small letter d
e	0x65 : latin small letter e
f	0x66 : latin small letter f
g	0x67 : latin small letter g
h	0x68 : latin small letter h
i	0x69 : latin small letter i
j	0x6A : latin small letter j
k	0x6B : latin small letter k
l	0x6C : latin small letter l
m	0x6D : latin small letter m
n	0x6E : latin small letter n
o	0x6F : latin small letter o

Table 1. System Manager user interface, allowed characters from ISO 8859-1 (Latin 1) (continued)

Char	Value : description
p	0x70 : latin small letter p
q	0x71 : latin small letter q
r	0x72 : latin small letter r
s	0x73 : latin small letter s
t	0x74 : latin small letter t
u	0x75 : latin small letter u
v	0x76 : latin small letter v
w	0x77 : latin small letter w
x	0x78 : latin small letter x
y	0x79 : latin small letter y
z	0x7A : latin small letter z
{	0x7B : left curly bracket
	0x7C : vertical line
}	0x7D : right curly bracket
~	0x7E : tilde

RELATED CONCEPTS

“Object Builder” on page 1

RELATED TASKS

“Developing in Object Builder” on page 19

Using the System Manager user interface (*System Administration Guide*)

RELATED REFERENCES

“Naming objects” on page 128

Creating a component for transient data

If your component has data that does not need to be stored, or you are providing customized persistence rather than using Component Broker services, you can create a component for transient data.

You can create a component for transient data in much the same way you create a component for new DB data, starting from the business object file and working down to the data object implementation. Because the data is transient, you do not need a persistent object or schema.

A component is identified as containing transient data by the setting on its data object implementation. When you create the data object implementation, set its Persistent Behavior and Implementation to **Transient**.

If you set the data object implementation's "Environment" on page 249 to **BOIM with UUID key**, you do not require a key for the component.

To create a component for transient data, complete these tasks:

1. "Creating a business object file" on page 775
2. "Adding a business object module" on page 777
3. "Adding a business object interface" on page 777
4. "Adding a key" on page 826
5. "Adding a copy helper" on page 830
6. "Adding a business object implementation and data object interface" on page 780
7. "Implementing methods" on page 752
8. "Adding a data object implementation" on page 807
9. "Adding a managed object" on page 871

For rules on naming the objects of the component, see "Naming objects" on page 128

RELATED CONCEPTS

Components (*Programming Guide*)

RELATED TASKS

"Chapter 4. Creating a component" on page 127

"Tutorial: Creating a component with transient data" on page 39

RELATED REFERENCES

"Naming objects" on page 128

"Internationalization of data" on page 132

Creating a component for new DB data

If you are creating a new component, which connects to a database that does not yet exist, you can create the entire component in Object Builder, starting with the business object interface and working your way down to a DB schema derived from the component's state data.

To create a new component directly in Object Builder, follow these steps:

1. "Creating a business object file" on page 775
2. "Adding a business object module" on page 777
3. "Adding a business object interface" on page 777
4. "Adding a key" on page 826
5. "Adding a copy helper" on page 830

6. “Adding a business object implementation and data object interface” on page 780
7. “Implementing methods” on page 752
8. “Adding a data object implementation” on page 807
9. “Adding a persistent object and schema” on page 833
10. “Adding a managed object” on page 871

For a scenario showing how to create a component for new DB data, see “Tutorial: Creating a component for new DB data” on page 50.

RELATED CONCEPTS

Components (*Programming Guide*)

RELATED TASKS

“Chapter 4. Creating a component” on page 127

RELATED REFERENCES

“Naming objects” on page 128

“Internationalization of data” on page 132

DDL

There are two types of DDLs (Data Definition Languages): System Management DDL and SQL DDL.

System Management DDL: a scripting language that defines the structure of an application on both client and server. Object Builder can generate a DDL script for your application family that defines the structure of the applications in the family. This generated DDL file is found in your working directory, under a subdirectory that has the same name as the application family. It is this file that provides the System Manager with information about the applications during the installation process.

SQL DDL: a language that describes data and their relationships in a database. It is composed of data definition statements that create, alter, or destroy database objects such as tables, aliases, views, and indexes.

A data definition is a program statement that describes the features of, specifies relationships of, and establishes the context of data. It has information that describes the contents and characteristics of a field, a record, or a file. A data definition can include field names, lengths, locations, and data types.

In Object Builder, you can import an SQL DDL file to create schemas within a schema group.

RELATED TASKS

“Generating the DDL files” on page 593

“Creating a DB schema by importing an SQL file” on page 844

Package file

When you create an Embedded SQL persistent object and generate code for it, a package file is created in the associated database, with the name you supply.

Follow these rules when you name the package file:

- It must not exceed eight characters
- It must be unique for each of the persistent objects that you create, if they are to operate under the same server at run time. Package filenames must be unique across models as well. That is, you cannot have package files with the same name that exist in different models.

The name you supply is incorporated in the generated code of the make file where the persistent object is built, and the package filename in the database is set to this value.

This package file is used to resolve any unresolved names such as column names and table names that are specified in the .sqx file, which is generated from the persistent object.

The DB2 precompiler takes the .sqx file (a C++ file with Embedded SQL) as input, and the make file creates a bind file with the same name as the generated .sqx file. This file is converted into the package file, which is bound to the database being accessed.

The purpose of the bind steps is to communicate your SQL requests to DB2, enabling DB2 to determine an optimal database access strategy.

RELATED CONCEPTS

Schema (*Programming Guide*)

Persistent object (*Programming Guide*)

RELATED TASKS

“Adding a persistent object and schema” on page 833

“Adding a persistent object from a DB schema” on page 837

DBCS and Binary Data Support

Double-byte character set (DBCS) is an encoding scheme for Asian characters such as Japanese. DB2 allows you to store both database metadata (for example: table names and column names) and database data in DBCS format. It also supports binary data storage.

The current release of Object Builder enables the following storage patterns:

- Database metadata names are in DBCS format
- Database data is stored in either DBCS or Binary format

Database meta-data names are in DBCS format

When you create a persistent object from a schema that was imported, if the given column name in the schema is an ASCII name (a legal C++ identifier), Object Builder will use the same name as the attribute name for the persistent object; otherwise, Object Builder generates names such as POAttribute1, POAttribute2. You can change these tool-generated names.

Database data is stored in either DBCS or Binary format

Object Builder uses the following “Data encoding schemes” on page 152 for data of *string* type that is stored in database meta-data:

- DBCS
- Single-byte character set (SBCS) or multi-byte character set (MBCS)
- Binary data

Note the following points when you do database queries:

- You can do database queries using DBCS or binary data just as you do queries with any other data type.
- You cannot do queries over large object types such as LONG VARCHAR and LONG VARGRAPHIC if they are used as either primary or foreign keys.

RELATED CONCEPTS

Schema (*Programming Guide*)

Persistent object (*Programming Guide*)

RELATED TASKS

“Adding a persistent object and schema” on page 833

RELATED REFERENCES

“Data encoding schemes” on page 152

“DB2 data type mappings” on page 142

“Oracle data type mappings” on page 146

“Mapping DBCS data types” on page 149

Creating a component for existing DB data

You can create a component for accessing existing or legacy database information by importing the database schema into Object Builder, and deriving a component from it, as follows:

1. “Creating a DB schema by importing an SQL file” on page 844
2. “Editing a DB schema group” on page 841

3. "Editing a generated SQL file" on page 857
4. "Adding a persistent object from a DB schema" on page 837
5. "Adding a data object from a DB persistent object" on page 814
6. "Adding a business object from a data object" on page 784
7. "Implementing methods" on page 752
8. "Adding a key" on page 826
9. "Adding a copy helper" on page 830
10. "Adding a managed object" on page 871

RELATED CONCEPTS

Component (*Programming Guide*)

Schema (*Programming Guide*)

"DDL" on page 137

RELATED TASKS

"Chapter 4. Creating a component" on page 127

RELATED REFERENCES

"Naming objects" on page 128

"Internationalization of data" on page 132

Supported CORBA Types

CORBA supports the following types:

Base Types

- float
- double
- long double
- short
- long
- long long
- unsigned short
- unsigned long
- unsigned long long
- char
- wchar
- boolean
- octet
- any

Constructed Types

- struct
- union
- enum


Template Types

- sequence
- string
- wstring
- fixed

Complex Declarators (arrays)

Native Types (language specific)

Note the following points:

- Object Builder currently does not support *long double*, *unsigned long long*, or *fixed*.
-  **390** Object Builder does not support *long long*, *wstring* and *wchar* on OS/390.
- Object Builder is packaged with the Enterprise Edition of Component Broker, which supports both the EJB component model and a CORBA-based component model called Managed Object Framework (MOFW). It includes implementations of almost all the current CORBA object services.
- **CORBA deployment:** The Enterprise Edition contains a CORBA ORB and the object services, but it is not designed to be used the way you would use a stand-alone CORBA ORB; that is, you do not write directly to low-level ORB interfaces like the Basic Object Adaptor (BOA), or the Portable Object Adaptor (POA). You write business objects to the MOFW component model, define the interfaces in IDL, and call CORBA object services from within your business object methods when you need to. If you have CORBA objects from another vendor's ORB, those can often be run inside Enterprise Edition, sometimes with a few modifications.

RELATED CONCEPTS

Persistent object (*Programming Guide*)

Application adaptor (*Programming Guide*)

RELATED TASKS

“Adding a persistent object and schema” on page 833

“Adding a persistent object from a DB schema” on page 837

“Adding a data object from a DB persistent object” on page 814

DB2 data type mappings

The tables on this page show the mappings among IDL, PO and SQL data types in different situations, assuming a DB2 backend database.

The following mappings are used when you create a schema and persistent object from a data object implementation:

IDL Type	PO Type	SQL Type	Encoding scheme
boolean	short	SMALLINT	
char	char[]	CHARACTER	
string[n] {string length fixed, 0 < n < 255}	DB2VARCHAR	VARCHAR[n]	SBCS or MBCS
string[n] {varying length, n > 255}	DB2VARCHAR[2000]	LONG VARCHAR	SBCS or MBCS
string (if it represents a decimal number)	char[]	DECIMAL	
double	double	DOUBLE	
	double	DECIMAL	
float	double	DOUBLE	
long	long	INTEGER	
unsigned long	long	INTEGER	
octet	short	INTEGER	
short	short	SMALLINT	
unsigned short	long	INTEGER	
any	DB2VARCHAR[2000]	LONG VARCHAR	SBCS or MBCS
void	DB2VARCHAR[2000]	LONG VARCHAR	SBCS or MBCS
Object	DB2VARCHAR[2000]	LONG VARCHAR	SBCS or MBCS
string	DB2VARCHAR[2000]	LONG VARCHAR	SBCS or MBCS
wstring (no size specified)	DB2VARGRAPHIC[2000]	LONG VARGRAPHIC	DBCS

IDL Type	PO Type	SQL Type	Encoding scheme
wstring[n] (fixed string length, 0<n<128)	DB2VARCHAR[2000]	VARGRAPHIC[n] (0<n<128)	DBCS
wstring[n] {varying length, n>=128}	DB2VARCHAR[2000]	LONG VARGRAPHIC	DBCS
wchar	wchar_t	GRAPHIC(1)	DBCS
struct	DB2VARCHAR[2000]	LONG VARCHAR	SBCS or MBCS
typedef	DB2VARCHAR[2000]	LONG VARCHAR	SBCS or MBCS
union	DB2VARCHAR[2000]	LONG VARCHAR	SBCS or MBCS
<i>interface</i>	DB2VARCHAR[2000]	VARCHAR[2000]	SBCS or MBCS
enum	long	INTEGER	
wstring	DB2VARGRAPHIC [2000]	GRAPHIC[n]	DBCS
string[n+1]	char[n+1]	CHAR[n]	SBCS or MBCS
IManagedClient ByteString	::ByteString	VARCHAR for bit data	Binary
IManagedClient ByteString	::ByteString	LONG VARCHAR for bit data	Binary
IManagedClient ByteString	::ByteString	VARGRAPHIC for bit data	Binary
IManagedClient ByteString	::ByteString	LONG VARGRAPHIC for bit data	Binary
All other types	DB2VARCHAR[2000]	LONG VARCHAR	SBCS or MBCS



You can map each of the IDL types in the table below with each of the PO types listed, without using a mapping helper:

IDL Type	PO Type
char	char
enum	
boolean	long
double	
float	short
long	
unsigned long	float
short	
unsigned short	double
octet	

Note: You can also map an IDL type string to a PO type char without using a mapping helper.

Object Builder provides the mapping helper (DB2MappingHelper) for the following IDL to PO type mappings:

IDL Type	PO Type	DO to PO Mapping Method	PO to DO Mapping Method
string	_DB2VARCHAR[]	stringToVarChar	varCharToString
<i>interface</i>	_DB2VARCHAR[]	byteStringToVarChar	varCharToByteString
<i>interface</i>	char[]	byteStringToString	stringToByteString
wchar	wchar_t[]	wStringToVarCharGraphic()	varGraphicToWString()
string	char[11]	stringToJavaDateString	javaDateToString
string	char[9]	stringToJavaTimeString	javaTimeToString
string	char[27]	stringToJavaTimestampString	javaTimestampToString
string	char[7]	stringToJavaBigIntegerString	javaBigIntegerToString
string	char[7]	stringToJavaBigDecimalString	javaBigDecimalToString

  For top-down development, when you map a data object to a persistent object, the IDL data type *long long* maps to the C++ long type on the persistent object. This is a mapping of a data type of higher precision (64 bits) to one of lower precision (32 bits), and Object Builder warns you that

there may be loss of precision as a result (due to integer truncation). This mapping is not supported on the OS/390 and Solaris platforms due to C++ compiler restrictions.

The following mappings are used when you create a persistent object from a schema:

SQL Type	Length	PO Type	Size	IDL Type	Size
CHARACTER	n	char[]	n+1	string	n
CHARACTER[1]	1	char	1	char	1
INTEGER		long		integer	
SMALLINT		short		short	
DOUBLE		double		double	
DECIMAL NUMERIC		double		double	
	n	char[]	n+2	string	(n+2)*Scale
BLOB	n	char[]	n	string	n*Scale
CLOB	n	char[]	n	string	n*Scale
DBCLOB	n	char[]	n	string	n*Scale
GRAPHIC	1	wchar_t[]	1	wchar	1
GRAPHIC	n	wchar_t[]	n+1	wstring	n
DATE		char[]	11	string[10]	
TIME		char[]	9	string[8]	
TIMESTAMP		char[]	27	string[26]	
VARCHAR	n	DB2VARCHAR[]	n	string	n
VARGRAPHIC	n	DB2VARGRAPHIC[]	n	wstring	n
LONG VARCHAR		DB2VARCHAR[]	2000	string	2000
LONG VARGRAPHIC		DB2VARGRAPHIC[]	2000	wstring	2000

The following mappings are used when you create a data object from a persistent object:

PO Type	IDL Type	Size
char	char	
char[n]	string	n-1
wchar_t	wchar	1
wchar_t[n]	wstring	n-1

PO Type	IDL Type	Size
short	short	
long	long	
double	double	
float	float	
DB2VARCHAR[n]	string	n
DB2VARGRAPHIC[n]	wstring	n
All other types	string	256

RELATED CONCEPTS

Persistent object (*Programming Guide*)

RELATED TASKS

“Adding a persistent object and schema” on page 833

“Adding a persistent object from a DB schema” on page 837

“Adding a data object from a DB persistent object” on page 814

Oracle data type mappings

►CHANGED

Object Builder uses the Oracle Application Adaptor (OAA) to access data in Oracle databases on the Windows NT, AIX, Solaris and HP-UX platforms.

Restrictions:

- WIN ► AIX ► SOLARIS Oracle 8.0.5 databases are supported only on the Windows NT, AIX and Solaris platforms.
- WIN ► AIX ► SOLARIS ► HP-UX Oracle 8.1.6 (Oracle 8i Release 2) databases are supported on the Windows NT, AIX, Solaris, and HP-UX platforms.
- Support for Oracle backend databases is limited to data objects that use the cache service only. That is, data objects that use embedded SQL, or any other type of persistence will not be able to access data stored in Oracle databases.
- Reference collections are not supported in conjunction with Oracle backends for the current release of Component Broker.
- In the current release of Component Broker, only the Oracle VARCHAR2 and NUMBER data types are supported, along with those Oracle data types that have an equivalent type in DB2. That is, Object Builder accepts all SQL/DS and DB2 types and the Oracle NUMBER, NUMBER(p),

NUMBER(p,s) and VARCHAR2 types. It will not accept any other Oracle types such as RAW(n), LONG RAW, NCHAR(n), NVARCHAR2, and ROWID.

- Object Builder will not accept the Oracle data type NUMBER with a negative scale.

The following table shows the DB2-equivalent Oracle types and their mappings to the persistent object type, Interface Definition Language type, and SQL type.

Oracle SQL type	precision (p)	scale (s)	PO type	IDL type	SQL type
NUMBER(p,s)	0	0	double	double	double
NUMBER(p,s)	1..4	0	short	short	smallint
NUMBER(p,s)	5..9	0	long	long	integer
NUMBER(p,s)	>=10	0	double / string	string	decimal(p,0)
NUMBER(p,s)***	p	<0	double / string	string	decimal(p,0)
NUMBER(p,s)	p	>38	double	double	double
NUMBER(p,s)	p	>p	double		double
NUMBER(p,s)*	p	s	double / string	string	decimal(p,s)
VARCHAR2(n)			string	string	varchar(n)
DATE**			string**	string	timestamp
RAW(n)***			::ByteString	::ByteString	varchar for bit data****
LONG RAW***			::ByteString	::ByteString	varchar for bit data****

* Consider NUMBER(p) = NUMBER(p,0) and NUMBER = NUMBER(38,0).

** Length 27, not 11 as in DB2.

*** Not supported by the **Import SQL** action in the current release of Object Builder.

**** Both varchar for bit data and varchar(n) for bit data are valid. If it has a maximum length (n), you must provide it. If you do not specify n, Object Builder allocates a buffer of 32 K.

RELATED CONCEPTS

Persistent object (*Programming Guide*)

Application adaptor (*Programming Guide*)

RELATED TASKS

“Adding a persistent object and schema” on page 833

“Adding a persistent object from a DB schema” on page 837

“Adding a data object from a DB persistent object” on page 814

Informix data type mappings

► NEW

► WIN ► AIX ► SOLARIS Object Builder uses the Informix Application Adaptor (IAA) to access data in Informix databases on Windows NT, AIX and Solaris platforms.






Informix type	DB2 equivalent type	CORBA (DAO) type	Comments
integer	integer	long	
smallint	smallint	short	
smallfloat	l; float	float	Cannot be mapped to CORBA::String.
float	double	double	Cannot be mapped to CORBA::String.
decimal(p,s)	decimal(p,s)	string	
decimal(p)	double	double	This is not a fixed point decimal, and is not the same as decimal(p,0). decimal(p) is handled in query as a double; not as an ICBCDecimal.
date	date	string	
datetime hour to second	time	string	
datetime year to fraction(5)	timestamp	string	yyyy-mm-dd hh:mm:ss.fffff
char(n)	char(n)	string	
varchar(n)	varchar(n)	string	

Note the following points:

- Object Builder does not support the types NCHAR(n), NVARCHAR(n), Byte, and Text, on SQL Import, in the generated SQL/DDDL or in SM/DDDL.

- In fact, Object Builder will neither import nor generate any Informix-specific SQL datatypes. The SQL/DDDL must consist of DB2's equivalents to the Informix types. For example, the SMALLFLOAT, DATETIME HOUR TO SECOND and DATETIME YEAR TO FRACTION(5) have to be expressed to Object Builder as FLOAT, TIME and TIMESTAMP respectively.
- DATETIME YEAR TO FRACTION(5) has five digits of fraction (for the seconds) whereas DB2's timestamp has six. Cache however pads the sixth digit with '0', returning datetime in ISO formatted string form:
yyyy-mm-dd hh:mm:ss.fffff0.

Restrictions:

- A given transaction cannot access more than one Informix database per CB server. To involve two Informix databases in a transaction, you must access each database from a different server.
-   Informix Dynamic Server version 7.30 files are supported only on AIX and Solaris.
-    Informix Dynamic Server version 7.31 files are supported only on Windows NT, AIX and Solaris.
- SQL files with the DOUBLE, TIME, and TIMESTAMP types will not load as-is into Informix. You can use a DBA design tool such as ERWin or DataAtlas to make appropriate changes.
- As for Oracle databases, support for Informix backend databases is limited to data objects that use the cache service only. That is, data objects that use embedded SQL, or any other type of persistence will not be able to access data stored in Informix databases. Cache will be responsible for all data transfers (insert, retrieve, update, delete) for Informix backend datastores. The Cache Service will use Informix ESQL and C interfaces for this purpose.

RELATED CONCEPTS

Persistent object (*Programming Guide*)

Application adaptor (*Programming Guide*)

RELATED TASKS

"Adding a persistent object and schema" on page 833

"Adding a persistent object from a DB schema" on page 837

"Adding a data object from a DB persistent object" on page 814

Mapping DBCS data types

Object Builder maps the *wstring* or *wchar* IDL types in the data object to the LONG VARGRAPHIC column type in the persistent object by default.

When you are creating an application that involves *wstring* data, and needs to store persistent object data in a CHARACTER column in a DB2 table, you must write your own mapping helper (a sample is given below), and follow the procedure described in the task “Mapping a data object to a persistent object” on page 703.

```

/*****
Name: MyMappingHelper.hpp
Description: A mapping helper class for mapping a WString_var DO data
to char* PO data, and vice versa.
*****/

#include <stdlib.h>
#include <string.h>
#include <wchar.h>

class MyMappingHelper {
public:
    // Conversion from WString_var to char*
    static void wstringToString
    (CORBA::WString_var& wszData, char* szData);

    // Conversion from char* to Wstring_var
    static void stringToWstring(const char* szData,
    CORBA::WString_var& wszData);
};

inline void MyMappingHelper::wstringToString
(CORBA::WString_var& wszData, char* szData)
{
    int instr_len;

    if (wszData == NULL){
        strcpy (szData, "");
        // This behavior, when the passed pointer is a
        // NULL pointer, is user-dependent.
    }else{

        // Get the number of wide characters to copy.

        instr_len = wcslen(wszData);

        // Copy the bytes into the char* variable.

```

```

        memcpy(szData, (wchar_t*)wszData,
instr_len*sizeof(wchar_t));
        szData[instr_len*sizeof(wchar_t)] = '\0';

    } // end of if (wszData == NULL)

}; // end of wstringToString()

inline void MyMappingHelper::stringToWstring(const char* szData,
CORBA::WString_var& wszData)
{
    int instr_len;
    char* szPtr1;

    if (szData == NULL){
        wscpy(wszData, L"\0");
    }else{
        /* Removing trailing blanks that DB2 inserted. This procedure
is optional, but if you choose not to remove them, please keep the
string length from exceeding the buffer length. */
        szPtr1 = (char*)szData;
        for (; *szPtr1; ++szPtr1);
        szPtr1--;
        for (; szData <= szPtr1 && *szPtr1 == 0x20; --szPtr1);
        memset(szPtr1+1, '\0', 1);
        /* End of removing trailing blanks */

// Get the number of bytes to copy.

        instr_len = strlen(szData);

// Copy the bytes into the wide character variable.

        memcpy((wchar_t*)wszData, szData, instr_len);
        wszData[instr_len/2] = L'\0';

    } // end of if (wszData == NULL)

}; // end of stringToWstring

```

RELATED CONCEPTS

Data object (*Programming Guide*)
Persistent object (*Programming Guide*)
“Mapping helper” on page 735

RELATED TASKS

“Adding a persistent object and schema” on page 833

RELATED REFERENCES

“Data encoding schemes”

Data encoding schemes

Object Builder uses the following data encoding schemes for database data:

DBCS encoding scheme

Attribute Type (IDL Type)	Attribute is a Key	SQL Type	PO Type	Size	ESQL or Cache Service
wchar	Yes, No	GRAPHIC[1]	wchar_t[]		ESQL
wstring	Yes, No	VARGRAPHIC[n]	_DB2VARGRAPHIC		ESQL
wstring	No	LONG VARGRAPHIC	_DB2VARGRAPHIC		ESQL
string		VARCHAR	char[]		Caching
string	No	LONG VARCHAR	char[]	2000	Caching
wchar		GRAPHIC	wchar_t[]		Caching
wstring		VARGRAPHIC	wchar_t[]		Caching
wstring	No	LONG VARGRAPHIC	wchar_t[]	2000	Caching

Binary Data encoding scheme**Note the following:**

- The ::ByteString option is available only with Cache Service, not ESQL.
- You should avoid using CHAR FOR BIT DATA with ESQL, because null values are lost.

Attribute Type (IDL Type)	Attribute is a Key	SQL Type	PO Type
ByteString	Yes, No	VARCHAR FOR BIT DATA	::ByteString or DB2VARCHAR
ByteString	Yes, No	VARCHAR	DB2VARCHAR
ByteString	No	LONG VARCHAR FOR BIT DATA	::ByteString or DB2VARCHAR

Attribute Type (IDL Type)	Attribute is a Key	SQL Type	PO Type
ByteString	No	LONG VARCHAR	DB2VARCHAR
ByteString	Yes, No	CHAR FOR BIT DATA	::ByteString

SBCS/MBCS encoding scheme

Attribute Type (IDL Type)	Attribute is a Key	SQL Type	PO Type	ESQL or Cache Service
any void Object string struct typedef union	No	LONG VARCHAR	DB2VARCHAR[2000]	Embedded SQL
		LONG VARCHAR	char[]	Cache Service

RELATED CONCEPTS

Data object (*Programming Guide*)

Persistent object (*Programming Guide*)

“DBCS and Binary Data Support” on page 138

RELATED TASKS

“Adding a persistent object from a DB schema” on page 837

“Adding a data object from a DB persistent object” on page 814

RELATED REFERENCES

“DB2 data type mappings” on page 142

“Oracle data type mappings” on page 146

OOSQL keywords

OOSQL Keywords					
adt	cursor	false	lcase	primary	true
alias	dao	fetch	like	procedure	type
all	data	flatten	lower	public	ucase
and	date	float	max	quit	union
any	day	for	microsecond	real	unique
as	days	foreign	min	ref	update
asc	decimal	from	minute	references	upper
avg	declare	goto	month	remote	usa
between	default	grant	nest	rollback	user
bit	define	group	not	schema	values
bo	delete	having	null	second	varchar
by	desc	hour	numeric	select	vargraphic
char	describe	in	of	set	view
character	digits	indicator	off	smallint	void
check	distinct	insert	on	some	whenever
close	do	integer	open	sqlcode	where
collection	double	into	option	sum	with
commit	drop	is	optional	table	work
continue	escape	iso	or	this	year
count	eur	jis	order	time	
create	exists	key	outer	timestamp	
current	explain	language	precision	to	

RELATED REFERENCES

Object-Oriented Structured Query Language (*Advanced Programming Guide*)

Keywords for query support



is of dynamic type(<only> type_name , <only> type_name...)

Examples:

```
select e from empHome e where e is of dynamic type (managerType);
```

returns only those objects that are either of type `manager`, or a subtype of the `manager` type.

```
select e from empHome e where e is of dynamic type (only managerType);
```

returns only those objects that are of type `manager`, and *not* subtypes of `manager`.

```
select e from empHome e where e is of dynamic type (managerType, studentType);
```

Note: Multiple type names can be in the list.

The type name

Can either be a CORBA IDL name, or an IDL name that is mapped to Java.

The query

```
select e from home e where e is of dynamic type (::Module1::eClass,  
::Module1::dClass);
```

is the same as the query

```
select e from home e where e is of dynamic type (Module1_eClass,  
Module1_dClass);
```

type_name()

An OOSQL function that returns the type name. It will return the interface name of the associated home, which is usually a CORBA IDL interface name. It can also return Java class names.

If the interface name of home1 is Module1::eClass with a subhome with interface Module1::dClass, then the query

```
select e.type_name() from home1 e; will return a list of strings that are  
either Module1::eClass or Module1:dClass.
```

treat ... as

You have to specify a type name. The type name can be either a CORBA IDL name, or an IDL name that is mapped to Java.

This does a safe downcast operation, returning null if the downcast cannot be done.

```
select e from personHome e where 1 = ((treat e as empClass) -> method1(  
123));
```

method1 is defined only empClass, not personClass. If the object is not a empClass then CAST returns null and the equal predicate returns null.

RELATED CONCEPTS

“Polymorphic homes” on page 581

RELATED REFERENCES

“OOSQL keywords” on page 154

Null value tolerance with sentinel values



Databases have the concept of null value for a field in a record. A *null value* indicates that no value has been set for the field. CORBA objects, however, have no corresponding null value concept. There is no way to represent a null value in a CORBA object.

When Object Builder encounters a null value in a field, it converts it into an arbitrary value that is then stored in the data object. If the data object is then written back into the persistent object, and back into the database, it is the arbitrary value, rather than null, which is written back into the database. The value in the database field is no longer null.

If you want to change this behavior, you can set up a sentinel value for a data object attribute. A *sentinel value* is a particular value which you want to be recognized as representing null. When Object Builder reads in a null, it will use the designated sentinel value, rather than an arbitrary value, to represent the field in the data object attribute. When it is writing back to a database, if it encounters a sentinel value attribute in the data object, it will write null into the database field.

Important: The sentinel values do not deliver null values to the client programmer. The client programmers (to the business object interface) and the OOSQL query writers will be exposed to the sentinel values that you, the data object implementation/persistent object developer, choose.

You can set a sentinel value for any data object attribute that is mapped to a persistent object, except for structure attributes and attributes which are designated as not null in the mapping (for example, keys).

Notes:

- Unless you specifically set a sentinel value for an attribute, the default behavior of Object Builder is to assign arbitrary values; null value tolerance is not the default behavior.
- If you write a query that tests whether a nullable attribute is null, be sure to check whether its value is the sentinel value, as well.

Foreign keys and null tolerance

When retrieving a foreign key from a database, the normal behavior is to build a key object with the data retrieved from the persistent object, and then to call the `findByPrimaryKey` method to get a pointer to the foreign object. However, if the key data is null, an arbitrary value is assigned to the key. The `findByPrimaryKey` method would often fail because the arbitrary value would be invalid, and the “correct” null value would be returned. However, there is the chance that the arbitrary value would be valid, and the `findByPrimaryKey` would return a non-null value.

Another consequence of the default behaviour is that a CORBA nil object reference will turn into a foreign key consisting of arbitrary (zeroes and empty strings) values. If the underlying table contains a corresponding foreign key constraint, a managed object with a nil object reference will usually fail to insert, resulting in an exception at runtime.

To avoid this kind of mistake, you can set an option on foreign keys so that if a null value is detected in the foreign key's persistent object, nil will be returned to the user immediately (the `findByPrimaryKey` method will not be called). Also, when inserting, it inserts a null foreign key instead of a series of arbitrary non-null values. (Of course, if the underlying table's foreign key also has a not-null constraint, then this too will fail.)

Note: Unless you specifically set the **Check for null** option for a foreign key, the default behavior of Object Builder is to assign an arbitrary value and to call the `findByPrimaryKey` method; null value tolerance is not the default behavior.

RELATED CONCEPTS

"Attributes" on page 698

RELATED TASKS

"Setting sentinel values for null field values" on page 701

"Checking for null foreign key values" on page 298

"Editing a data object implementation" on page 819

"Mapping a data object to a DB persistent object" on page 703

Creating a component for PA data

You can create a component for accessing existing transactional information by importing the relevant PA bean into Object Builder, and deriving a component from it, as follows:

1. "Creating a PA schema by importing a PA bean" on page 862
2. "Customizing PA bean query methods" on page 160
3. "Adding a persistent object from a PA schema" on page 860
4. "Adding a data object from a PA persistent object" on page 815
5. "Adding a business object from a data object" on page 784
6. "Implementing methods" on page 752
7. "Adding file adornments" on page 240
8. "Adding method adornments" on page 242
9. "Adding a key" on page 826
10. "Adding a copy helper" on page 830
11. "Adding a managed object" on page 871
12. "Adding resource methods to a sessional business object" on page 164

For an introduction to this functionality, follow these tutorials:

1. "Tutorial: Unit test for procedural adaptors" on page 166
2. "Tutorial: Creating a component for PA data (bottom-up)" on page 167

Once you have assembled a component for PA data, you can then build DLLs for the component, and package the component into an application.

RELATED CONCEPTS

Components (*Programming Guide*)
Schema (*Programming Guide*)
“Procedural adaptor bean (PA bean)” on page 159
Session Service (*Advanced Programming Guide*)
Transaction Service (*Advanced Programming Guide*)
Connections to a tier-3 system (*System Management*)

RELATED TASKS

“Chapter 4. Creating a component” on page 127
“Configuring builds” on page 549
“Building the JAR files” on page 561
“Packaging applications” on page 574
Configuring a new ECI connection to a tier-3 CICS® region (*System Management*)
Configuring a new HOD connection to a tier-3 system (*System Management*)
Configuring a new SAP connection to a tier-3 system (*System Management*)
Configuring a new APPC connection to a tier-3 system (*System Management*)
Configuring the iPAAServices application onto an application server (*System Management*)

RELATED REFERENCES

“Naming objects” on page 128
“Internationalization of data” on page 132

Enterprise Access Builder (EAB)

Enterprise Access Builder (EAB) is a set of class frameworks and development tools in VisualAge for Java 3.0 that enable you to move your applications from a front-end transaction system such as Customer Information Control System (CICS), or Information Management System (IMS), to an object-oriented programming environment. Procedural Adaptor (PA) beans that are created using EAB can be imported into Object Builder as PA schemas and PA persistent objects.

Enterprise Access Builder (EAB) used to be referred to as CICON, which stood for Customer Information Control System (CICS) and Information Management System (IMS) Connection, in previous releases of Component Broker.

RELATED CONCEPTS

Persistent object (*Programming Guide*)

Schema (*Programming Guide*)
“Procedural adaptor bean (PA bean)”

RELATED TASKS

“Creating a PA schema by importing a PA bean” on page 862
“Adding a persistent object from a PA schema” on page 860

Procedural adaptor bean (PA bean)

▶CHANGED

A PA bean is a bean in VisualAge for Java that inherits from the `com.ibm.ivj.eab.paa.EntityProceduralAdapterObject` class. PA beans, built using Enterprise Access Builder (EAB), wrap existing transactions for reuse in Component Broker.

PA beans are imported into Object Builder as PA schemas. By default, a PA persistent object is generated for each bean that you import, but you can create one yourself, for the PA schema. The PA persistent object uses the definition of the PA schema to make calls to the PA bean.

RELATED CONCEPTS

Persistent object (*Programming Guide*)
Schema (*Programming Guide*)
“Enterprise Access Builder (EAB)” on page 158

RELATED TASKS

“Creating a PA schema by importing a PA bean” on page 862
“Adding a persistent object from a PA schema” on page 860

Java data type mappings

The PA bean that you import into Object Builder cannot have arrays as either attribute types, or method parameter types, or method return types. The only types that are supported are those listed in the table below:

PAO Type	IDL Type
int	long
float	float
double	double
boolean	boolean
short	short
byte	octet
void	void
char	char
	wchar

PAO Type	IDL Type
java.lang.String	string
	wstring

Note: Depending on the type you select when you import the bean, *char* is converted to either *char*, or *wchar*, and *java.lang.String* is converted to either *string*, or *wstring*.

RELATED CONCEPTS

Persistent object (*Programming Guide*)

RELATED TASKS

“Creating a PA schema by importing a PA bean” on page 862

Supported platforms for connectors



The following table shows which connectors are supported on which platforms for Object Builder procedural application adaptor support.

Connector	OS/390	AIX	HP-UX	Windows NT	Solaris
HOD	N	Y	N	Y	Y
SAP	N	Y	N	Y	Y
ECI	N	Y	N	Y	Y
APPC (LU6.2)	N	Y	N	Y	Y
OTMA	Y	N	N	N	N
EXCI	Y	N	N	N	N
IMS-APPC (OS/390 version)	Y	N	N	N	N
Generic	Y	Y	N	Y	Y

Customizing PA bean query methods

If you import a PA bean that is queryable, the Query Methods page is dynamically added to the wizard. The query methods are listed in the Query Methods folder. Each of the query methods of the bean is associated with an OOSQL WHERE clause.

You can determine the subset and the type of data that a selected PA bean's query method will return. If you want a query method to fetch every element in the procedural application adaptor (PAA) home, you do not have to specify a WHERE clause for that particular query method. If you want the method to return only certain elements in the PAA home, you must specify the search criteria using a WHERE clause.

To customize a PA bean's query method, follow these steps:

1. Select the query method in the Query Methods folder. A set of controls appear on the page.
2. Type a clause in the **WHERE Clause** field that indicates which of the elements in the PAA home have to be returned. If you need a substitution string, you can specify it. Follow the "WHERE clause syntax" on page 866 rules when you construct the clause.

If you use parameters in the WHERE clause, you must map each of them to an attribute of similar type in the PA bean. See the task: "Mapping query method parameters to PA bean attributes".

RELATED CONCEPTS

"Enterprise Access Builder (EAB)" on page 158

"Procedural adaptor bean (PA bean)" on page 159

Persistent object (*Programming Guide*)

Application adaptor (*Programming Guide*)

Session Service (*Advanced Programming Guide*)

Transaction Service (*Advanced Programming Guide*)

RELATED TASKS

"Creating a PA schema by importing a PA bean" on page 862

"Adding a data object implementation" on page 807

"Mapping query method parameters to PA bean attributes"

RELATED REFERENCES

"WHERE clause syntax" on page 866

Mapping query method parameters to PA bean attributes

You can determine the subset and the type of data that a selected PA bean's query method will return. For queryable beans, you can do this using the Query Methods page, either when you import the bean, or when you edit the properties of the bean.

You use a WHERE clause to define the search criteria. Object Builder performs syntax checking on the WHERE clause. If you want to fetch every element in the procedural application adaptor (PAA) home, then you do not need a WHERE clause for that particular query method. If you do use a WHERE clause, you

may encounter situations when you have to use parameters. You would use a parameter when a value may not be determined until run time, or when the end user will be inputting the value.

OOSQL parameters in a WHERE clause are preceded by a colon (:). For example, in the following WHERE clause, p1 is the parameter, and it can take variable values:

```
WHERE AMOUNT > :p1
```

Whenever you use a parameter in a WHERE clause, you must map it to an attribute of the same type in the PA bean. If the WHERE clause contains at least one parameter, you cannot complete the criteria specification if you do not provide the mapping.

The parameters that you use in the clause appear in the Query Methods folder, beneath the selected query method, in the WHERE Clause Parameters folder.

To provide the mapping, follow these steps:

1. Select the parameter from the WHERE Clause Parameters folder (the parameter name will not include the colon from the WHERE statement). The parameter's details such as its name and data type are displayed on the page.
2. Type the name of an attribute of the PA bean, which is of the same type as the parameter, in the PA Attribute field. You can also click the list button, and select one of the PA bean's attributes of the same type.

Tip: If you are not sure of the data types of the PA bean's attributes, do not use the Query Methods page when you are first importing the bean. Follow these steps:

1. Import the PA bean. (From the pop-up menu of the User-Defined PA Schemas folder, select **Import Bean**. Specify, or select the bean, indicate its key attributes, and click **Finish**.)
2. Then examine the properties of the PA schema that is created from the PA bean. (From the pop-up menu of the PA schema in the User-Defined PA Schemas folder, select **Properties**.)
3. View the attribute types of the bean on the Attributes page.
4. Go to the Query Methods page of the same wizard, and specify the search criteria for the bean's query methods. You can now use parameters in the WHERE clause, and provide the mapping between them and the relevant PA bean attributes using the procedure that is explained above.

RELATED CONCEPTS

"Enterprise Access Builder (EAB)" on page 158 "Procedural adaptor bean (PA

bean)" on page 159
Persistent object (*Programming Guide*)
Application adaptor (*Programming Guide*)
Session Service (*Advanced Programming Guide*)
Transaction Service (*Advanced Programming Guide*)

RELATED TASKS

"Creating a PA schema by importing a PA bean" on page 862
"Adding a data object implementation" on page 807
"Customizing PA bean query methods" on page 160

RELATED REFERENCES

"WHERE clause syntax" on page 866

Handling exceptions thrown by PA bean push-down methods

In business applications, the client's knowledge of exceptions that are thrown in certain situations is crucial.

If the business logic for this transaction is stored in a procedure in the CICS or IMS™ backend and made available through a push down method, then an exception can be raised in such an event. The exception can be converted to an appropriate exception as described above, be sent back to the client program, and shown to the user.

For example, in a banking transaction, an error condition may result in the event that there are insufficient funds in an account.

To handle exceptions that are thrown by PA bean push-down methods, follow these steps:

When you import the PA bean, Object Builder examines the bean to determine if there are any push-down methods defined on it. If so, these methods are added to the PA schema and to the PA persistent object. Any exceptions that these methods throw are also logged.

During code generation, an intermediary class (POIF) is defined to bridge the gap between the PA persistent object, which is written in C++ and the PA bean, which is written in Java. Exceptions that are thrown by the push-down methods are also defined on this intermediary class.

This intermediary class is normally hidden, but is exposed for the definition and handling of exceptions. When an exception is thrown by a push-down method in the PA bean, that exception is caught, and converted into a C++ exception defined on this intermediary class. The data in the original Java exception is lost, as is its inheritance.

This new C++ exception is internal, and is not meant to be exposed to any client program. You must write code in the push-down method of the data object implementation to catch it and handle it appropriately.

If you want the exception to be handled in the business object, or by the client program, you must convert it into an exception which is defined on the data object. Follow these steps:

1. Define an exception in the data object.
2. Add the exception to the push-down method on the data object.
3. Convert the C++ exception to the data object exception. To do this, you must change the implementation of the push-down method in the data object. Where the call to the push-down method on the PA persistent object is made, catch the C++ exception and in its place throw the data object exception you defined in step 1.

This handles the exception from the PA bean up to the data object. You can now handle this exception in the business object implementation. If you want the client program to handle the exception you must convert it into an exception defined on the business object. Follow these steps:

1. Define an exception in the business object.
2. Add the exception to the push-down method on the business object.
3. Convert the data object exception to the business object exception. To do this, you must change the implementation of the push-down method in the business object. Where the call to the push-down method on the data object is made, catch the data object exception and in its place throw the business object exception you defined in step 1.

This is an overview of the process you follow. For more detailed steps, refer to the *Programming Guide*.

RELATED CONCEPTS

Data object (*Programming Guide*)

Persistent object (*Programming Guide*)

“Procedural adaptor bean (PA bean)” on page 159

“Push-down methods” on page 759

RELATED TASKS

CICS and IMS application adaptor exception handling (*Programming Guide*)

Adding resource methods to a sessional business object

When a business object uses Session Service, you can provide your own code to be called during some of the normal processing for those services. You can do this by calling the `endResource()`, the `checkpointResource()`, and the `resetResource()` method that you define on the business object, in both C++ and Java implementations.

You cannot call these methods that are used for resource management when the target platform is OS/390.

Follow these steps:

1. From the pop-up menu of the business object implementation, select **Properties**. The Business Object Implementation wizard opens to the Name and Data Access Pattern page.
2. Under the section: "Session Service" on page 248, select the **Provides resource support** check box.

By selecting it, you indicate that you want Object Builder to create the `endResource()` method, the `checkpointResource()` method, and the `resetResource()` method on the business object.

When you select the business object implementation in the Tasks and Objects pane, you see these methods that were created for the implementation by Object Builder, in the Framework Methods folder. All of them have empty method bodies.

3. To add your own code for these methods, first select the method from the Framework Methods folder, and from its pop-up menu, select **Properties**.
4. On the Implementation page of the Method Implementation wizard, select the **Use the implementation defined in the Source** pane radio button, and click **Finish**.

The method is now editable in the Source pane, when you select the method in the Methods pane.

Note: You can also select the **Use an external file** option, if you have the code stored in either an external template file, or a normal file.

5. Provide your own code for the method body in the Source pane.
This method of the business object implementation that contains your code is called by the framework when the corresponding method of the same name is called on the managed object's mixin.
6. If you have not yet added a managed object for your component, add one now: From the pop-up menu of the business object implementation, select **Add Managed Object**. Select **Session Service** as the service to be used by the business object, and specify parents, if any, for the implementation.
7. Generate code for the managed object: From the pop-up menu of the object, select **Generate > Selected > All files**, or **Generate > Selected > .cpp**

The `.cpp` file that is generated contains the resource management methods that contain your code. For each of these methods, if you want to write a separate method to contain your code, you must call this method from within the resource management method.

RELATED CONCEPTS

Business object (*Programming Guide*)

Session Service (*Advanced Programming Guide*)

RELATED TASKS

“Generating code” on page 551

Tutorial: Unit test for procedural adaptors



The stand-alone session support is used to provide a similar test environment to that provided by the Component Broker run time.

When testing a PA bean outside of Component Broker, a stand-alone session service is provided as part of the stand-alone Common Connector Framework (CCF) classes (that is, as part of the `com.ibm.ivj.communications` package), and is therefore available when you use those classes.

To use this stand-alone session service, the unit test case of the PA bean needs to perform the following actions:

- If you are using a sessional connector, invoke the static method `com.ibm.ivj.communications.Session.startSession()` before the PA bean is constructed. There currently is no facility for testing APPC in VisualAge for Java.
- invoke the static method `com.ibm.ivj.communications.Session.endSession(tf)` when the PA bean is no longer needed (that is, just before the end of the unit test program, or as appropriate if the unit test needs to perform more comprehensive testing with sessions).

The parameter *tf* is a boolean parameter. If its value is set to *true*, it indicates that the session is to be checkpointed (which means that all changes are committed and are to be kept); if it is set to *false*, it means that the session is to be reset.

The unit test scenario requires the following steps to be done in VisualAge for Java:

1. Before the call to `startSession()`, create the desired `connectionSpec` (for example, `HODConnectionSpec`) and set appropriate values for the host name and port.
2. After the call to `startSession()`, create an instance of the PA bean's key. Then, on the PA bean, call the static method `find()` and pass in the instance of the key as a parameter. The `find` method will return an instance of the PA bean.
3. On the instance of the PA bean, set the `connectionSpec` to the `connectionSpec` created in step 1.

Note: The connectionSpec set in the create, retrieve, update, and delete methods will take precedence over any previous connectionSpec that may have been set.

4. Once the connectionSpec is set, you can make calls on the instance of the PA bean, as desired.

This unit test scaffolding can be kept in place even when the PA bean is deployed in a Component Broker scenario because the connectionSpec passed in from CB will take precedence over any previously set connectionSpec.

RELATED CONCEPTS

“Enterprise Access Builder (EAB)” on page 158

“Procedural adaptor bean (PA bean)” on page 159

Connections to a tier-3 system (*System Management*)

RELATED TASKS

“Creating a component for PA data” on page 157

“Tutorial: Creating a component for PA data (bottom-up)”

“Working with container instances” on page 883

Tutorial: Creating a component for PA data (bottom-up)

This tutorial assumes that you have successfully installed and configured Component Broker. You will create a component with procedural adaptor persistence.

For a complete list of prerequisites that you must install on your system, see the *Getting Started with Component Broker* documentation for the appropriate platform.

Note: Procedural Adaptor Object (PAO) beans, which are created with both VisualAge for Java version 3.0 and version 2.0 are supported with this release of Component Broker.

Enabling the IBM Component Broker CICS and IMS application adaptor functionality

For the bean to be found during import, ensure that the JAR file (beans.jar), which contains the bean class you are to import, is in your system CLASSPATH variable.

Restriction: The CLASSPATH variable may not have contents longer than 1780 characters. The longer the name of your installation directory, the less space left for the CLASSPATH value (Therefore, it is recommended that you use the default installation directory x:\Cbroker). If the CLASSPATH exceeds this limit, you will get a run-time error (“line too long”) when you start Object Builder.

This is because commands (such as ob.bat), which invoke the Object Builder functions, prefix the Object Builder .jar files to the class path, and then invoke the Java code to run Object Builder.

Creating a new project

1. Start Object Builder.
2. The Open Project wizard opens to the Project Directory page. Type a name and path for the project directory (for example, e:\scenarios\ABeCashAcct).
3. Click **Finish**.

Note: If the project directory has never been used before, and contains no models, Object Builder confirms with you if you want to create a model in the directory. It then prompts you for a new model name. It shows you a default model name, which it assumes is the same as the directory name for the project. You can either accept that name, or change it. Click **OK**.

4. Click **Yes**, to create a new project.

Importing the PA bean

The class name for this bean is paa.samples.cics.appc.acct.ABeCashAcctPAO.

1. In the Tasks and Objects pane, select the User-Defined PA Schemas folder, and from its pop-up menu, select **Import - Bean**. The Import Procedural Adaptor Bean wizard opens to the Bean Selection page.
2. You can choose to either type the name of the bean class, or select the JAR file containing the file, and then select the bean class. Select the **Enter bean name** radio button, and type the name of the class (paa.samples.cics.appc.acct.ABeCashAcctPAO) in the field.
3. Click **Next**. The Names and Connectors page opens. Type the name of the module and the persistent object to be associated with the PA schema. You can also select the connector type to be used to access objects. Select LU6.2 as the connector type. This is the type of connector ABeCashAcctPAO uses.



When you select OS/390 as the development (target) platform (**Platform > Constrain > 390**), only the EXCI, OTMA, IMS APPC, and Generic connector types are available for selection.

Note: When you select either NT and 390, or AIX and 390 as the development platforms, all the connector types are available for selection. However, in this scenario you must not select 390 either alone, or in combination with one of NT or AIX, as the sample bean is for an ECI connector, and is not valid on OS/390: if you select 390, you will not be able to select ABeCashAcctPAOPO as the type of your persistent object.

4. Click **Next**. Select res_type and account_ID (two of the properties of the bean) from the **Properties** box, and move them to the **Key Attributes** box, by clicking the >> button.

5. Click **Next**. The Variable Type Specification page opens. Accept the defaults.
6. Click **Finish**, and the bean will be imported into Object Builder. The ABeCashAcctPAO schema and its associated persistent object ABeCashAcctPAOPO appear in the tree under User-Defined PA Schemas folder.

Connecting the imported bean with an application

We can connect the imported bean with either existing applications, or those created after the bean is imported. We will create a new application.

Creating the application objects (business object, data object, managed object)

Creating the ACashAcct business object file:

1. From the pop-up menu of the User-Defined Business Objects folder, select **Add File**.
2. The Business Object File wizard opens to the Name page.
3. Type ACashAcct in the **Name** field, and click **Finish**.
4. The ACashAcct file appears in the User-Defined Business Objects folder.

Creating the ACashAcct interface:

1. From the pop-up menu of ACashAcct, select **Add Interface**.
2. The Business Object Interface wizard opens to the Name page.
3. Type ACashAcct as the name of the interface in the **Name** field.
4. Click the arrow to the left of the page name, and select Attributes from the list. The page opens.
5. From the pop-up menu of the Attributes folder, select **Add**.
6. In the **Attribute Name** field, type `res_type` as the name of an attribute.
7. For the data type of the attribute, select *string* from the **Type** field.
8. Type 0 in the **Size** field.
9. Use the **Add Another** button to add the next attribute.
10. Add the attribute `balance`, of type *long* and the *string* attributes `account_ID`, `acct_type`, and `utilities`, using steps 6 - 9.
11. Click **Refresh** instead of **Add Another**, after you add the last attribute.
12. Click **Next**. The Methods page opens.
13. Click **Finish**. The ACashAcct interface appears under the ACashAcct file, in the folder.

Adding the key:

1. From the pop-up menu of the ACashAcct interface, select **Add Key**.

2. The Key wizard opens to the Name and Key Attributes page. From the **Business Object Attributes** box, select `res_type` and `account_ID`, and click the `>>` button to move them to the **Key Attributes** box.
3. Click **Finish**. The key `ACashAcctKey` appears beneath the `ACashAcct` interface.

Adding the copy helper:

1. From the pop-up menu of the `ACashAcct` interface, select **Add Copy Helper**.
2. The Copy Helper wizard opens to the Name and Attributes page.
3. Click the **All>>** button to select all the business object interface attributes as attributes of the copy helper.
4. Click **Finish**. The copy helper, `ACashAcctCopy` appears under the `ACashAcct` interface.

Adding the business object implementation and the data object interface:

1. From the pop-up menu of the `ACashAcct` interface, select **Add Implementation**.
2. The Business Object Implementation wizard opens.
3. Select **Delegating** as the **Pattern for Handling State Data**.
4. From the **Data Object Interface** section, make sure that **Create a new one now** is selected.
5. Click the arrow to the left of the page name, and select `Key` and `Copy Helper` from the list. The page opens. Make sure that `ACashAcctKey` is selected as the key, and `ACashAcctCopy` is selected as the copy helper.
6. Turn to the Data Object Interface page.
7. Click the **All>>** button to select all the attributes of the business object as state data for the data object.
8. Click **Finish**. The business object implementation `ACashAcctBO` appears under the `ACashAcct` interface, and the data object interface `ACashAcctDO` appears as a node beneath the implementation.

Adding the data object implementation and connecting the `ABeCashAcctPAOPO` persistent object:

1. From the pop-up menu of the `ACashAcctDO` interface, select **Add Implementation**.
2. The Data Object Implementation wizard opens to the Name and Platform page.
3. Accept the default names, and select `NT` and `AIX` as the deployment platforms.
4. Click **Next**. The Behavior page opens.
5. From the **Environment** section, select **BOIM with any key**.
6. From the **Type of Persistence** section, select **Procedural Adaptors**.

7. Click **Next**. The Implementation Inheritance page opens.
8. Verify that the class IPAAExtLocalToServer appears under the Parents folder.
9. Click the arrow to the left of the page name, and select Associated Persistent Objects from the list. The page opens. From the pop-up menu of the Persistent Object Instances folder, select **Add**
10. Type iABeCashAcctPAOPO in the **Instance Name** field.
11. Click **Next**. The Attributes Mapping page opens.
Note: The following three steps may be done automatically for you if the names of the attributes are the same in the persistent object and in the data object, and if the persistent object is being associated with the data object for the first time.
12. Select the attribute *res_type* of the data object from the Attributes folder, and from its pop-up menu, select **Primitive**.
13. Click the list button, and select the attribute iABeCashAcctPAOPO.phone of the persistent object from the **Persistent Object Attribute** field. You have just defined a one-to-one mapping between the data object and the persistent object.
14. Repeat steps 12 and 13 for all the other attributes in the folder, mapping them one-to-one.
15. Click **Next**. The Methods Mapping page opens.
16. Select the insert() special framework method from the folder, and from its pop-up menu, select **Add Mapping**.
17. Click the list box, and select iABeCashAcctPAOPO.insert() from the **Persistent Object Method** field.
18. Repeat steps 16 and 17 for all the methods update(), retrieve(), del(), and setConnection(), using a one-to-one mapping.
19. Click **Finish**.

The data object implementation, ACashAcctDOImpl will now appear under the ACashAcctDO interface, and the ABeCashAcctPAOPO persistent object will appear under the ACashAcctDOImpl data object implementation.

Adding the managed object:

1. From the pop-up menu of the ACashAcctBO business object implementation, select **Add Managed Object**.
2. Under **Service to Use**, select **Transaction Service** (automatically selected for you if the platform is OS/390).
3. Click **Finish**.

The managed object appears under the business object implementation.

Exporting as XML

If you want to reuse the component that you just created in other scenarios, you can export it in XML format:

1. Click **File > Export Model**. The Export Model wizard opens.
2. Click **Finish**.
3. If asked whether you want to create the export directory, select **Yes**.

XML files that represent the elements of your component are exported to current project's \Working\Export directory.

Generating the application code

From the pop-up menu of the ACashAcct file in the User-Defined Business Objects folder, select **Generate > All**. Code generation will begin, and you can monitor the progress in the bottom left corner of Object Builder's window.

Configuring the build

Add the client DLL:

1. From the pop-up menu of the Build Configuration folder, select **Add Client DLL**. The Client DLL wizard opens.
2. Type ACashAcctC in the **Name** field.
3. Click the page title and turn to the Client Source Files page.
4. Click the **All >>** button to select all the client source files.
5. Click **Finish**.

The ACashAcctC DLL will appear in the Build Configuration folder.

Add the server DLL:

1. From the pop-up menu of the Build Configuration folder, select **Add Server DLL**. The Server DLL wizard opens.
2. Type ACashAcctS in the **Name** field.
3. Click **Next**.
4. Click the >> button to add the ACashAcctC dll to the list of Libraries to link with.
5. Click **Next**.
6. Click the **All >>** button to select all the server source files.
7. Click **Finish**.

The ACashAcctS DLL will appear in the Build Configuration folder.

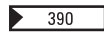
Building the DLLs

Generate the configuration:

From the pop-up menu of the Build Configuration folder, select **Generate > All**. Code generation will begin.

Creating a container instance

1. From the pop-up menu of the Container Definition folder, select **Add Container Instance**. The Container wizard opens.
2. Type ACashAcctContainer in the **Name** field.



If you are developing an application intended for deployment on OS/390 (the **Platform** > **Constrain** > **390** menu choice is checked), you are now done. The rest of the container definition is handled through the System Management user interface.

3. Click the arrow to the left of the page name, and select Service from the list. The Service page opens. Select **Use PAA Transaction Service**.
4. On the Service Details page, specify a name of your choice for the connection. Select **LU6.2** for the connector type used by the session.
5. Click **Finish**.

The ACashAcctContainer will appear in the Container Definition folder.

Configuring the application

Add an application family:

1. From the pop-up menu of the Application Configuration folder, select **Add Application Family**. The Application Family wizard opens.
2. Type ACashAcctApp in the **Name** field.
3. Click **Finish**.

The ACashAcctApp family will appear in the Application Configuration folder.

Add an application:

1. From the pop-up menu of the ACashAcctApp application family, select **Add Application**. The Application wizard opens.
2. Type ACashAcct in the **Name** field.
3. Click **Finish**.

The ACashAcct application will appear under the ACashAcctApp family.

Add the application's managed object:

1. From the pop-up menu of the ACashAcct application, select **Add Managed Object**. The Managed Object Configuration wizard opens.
2. Click the list box of the **Managed Object** field, and select ACashAcctMO ACashAcctMO from the list.
3. Click **Next**.
4. From the pop-up menu of the Implementations folder, select **Add**.
5. Click the list box of the **Data Object Implementation** field, and select ACashAcctDOImpl from the list.

6. Click **Next**.
7. Click the list box of the **Name** field, and select ACashAcctContainer from the list.
8. Click **Finish**.

The ACashAcctMO managed object will appear under the Acct application.

Generate the application family:

From the pop-up menu of the ACashAcctApp family, select **Generate**.

Building the ACashAcct application (client and server)

Set up the environment:

You had added the location of the .jar file that contains the bean you import to your system class path variable: CLASSPATH. This location is required for import, and for the server to find the bean. So, reboot your system for the new environment variables to take effect. The server will then be able to find the bean.

Starting the build

1. Go to the working\NT directory. For this tutorial, this should be located in e:\scenarios\ABeCashAcct.
2. Type `nmake -f all.mak cpp java`
3. The ACashAcct application should be built.
4. Copy ACashAcctS.dll and ACashAcctC.dll to the CBroker\bin directory to place them in your system path.

Installing the application

To install the application, you must be logged on to DCE, and the System Manager User Interface must be running. Then you load and configure the application by following the steps below.

Loading the application onto System Management

1. Start the System Manager User Interface, if it is not already started.
2. Enable Control actions by selecting **View > View Level > Control**.
3. Expand **Host Images**, and select your host name.
4. From the pop-menu, select **Load application** to open the Load application dialog. Select
e:\scenarios\ABeCashAcct\Working\NT\CashAcctApp\ACashAcctApp.dll.

Configuring the application with System Management

1. Create the server:
 - a. From the **Tasks** menu, select **Create Servers**.

- b. In the Management Zone window, select **Available Items > SampleApplication Zone**. Click **Next**.
 - c. In the Configuration window, select **Available Items > Sample Configuration**. Click **Next**.
 - d. In the Server Group window, type ACashAcctServerGroup for the name of the server group. Click **Next**. If the server group does not exist, click **Yes** to create the server group.
 - e. In the Server window, type ACashAcctServer for the Name of the server. Click **Finish**.
2. Configure the server:
 - a. Click on the **Tasks** menu and select **Configure Servers**.
 - b. In the Select Applications To Configure window, select **Available Applications > iPAAServices** and **ACashAcct**. Click **Add** to move the applications to Applications To Configure. Click **Next**.
 - c. In the Management Zone window, select **Available Items > Sample Application Zone**. Click **Next**.
 - d. In the Configuration window, select **Available Items > Sample Configuration**. Click **Next**.
 - e. In the Select Servers To Configure Applications on window, select **Available Server Groups > ACashAcctServerGroup**. Click **Add** to move the available server group to Servers To Configure Application On. Click **Finish**.
 3. Configure the APPC connection:
 - a. Expand **Management Zones > Sample Application Zone > Configurations > Sample Configuration > APPC Connections** and select **APPC_ACashAcct_Server**.
 - b. From the pop-up menu, select **Properties**. The Properties Editor opens.
 - c. Click the **Main** tab.
 - d. Change the **Fully qualified Local LU name** field to match the local LU6.2 LU that you will use to communicate with your CICS/IMS system (for example, PAA01001).
 - e. Change the **Fully qualified Partner LU name** field to match the partner LU6.2 LU that you will use to communicate with your CICS/IMS system (for example, USIBMZP.CICS4).
 - f. Change the **Mode Name** field to match the mode name that you will use to communicate with your CICS/IMS system (for example, LU62PS).
 - g. Change the **Remote Procedure Type** field to match the type of program with which you will be communicating (for example, CICS_DPL or CICS_DTP). The CICS_DPL flavor appends eight bytes (converted to the target code page) that correspond to the CICS application to which the DTP program should EXEC CICS LINK.

- h. Change the **Transaction Program Name** field to match the CICS transaction program (TP) that you will run (for example, BDPL or BDTP).
 - i. Change the **CICS Program Name** field to match the CICS program name that you will run under the transaction program (for example, BECASHAC).
 - j. Change the **transaction type** field to be either optimistic or pessimistic. Pessimistic initiates the conversation as sync-level 2 for the entire transaction while optimistic only talks sync-level 2 during the prepare and commit parts of the transaction.
 - k. Click **OK** to validate and accept the changes.
4. (Optional): Enable security service for the application server and the name server:
 - a. Expand **Management Zones > Sample Application Zone > Configurations > Sample Configuration > Server Groups**, and select **ACashAcctServerGroup**.
 - b. From the pop-up menu, select **Properties** to open the Properties Editor.
 - c. In this notebook:
 - 1) Select the **Security Service** tab.
 - 2) Change the value for the **Enable security** field from no to yes.
 - 3) Change the **Delegate credentials** to None.
 - 4) Change the **Credential mapping** to Simple.
 - 5) Change the value for the **data system principal** field to the user ID that the server will use when connecting to the CICS system.
 - 6) Change the value for the **data system password** field to the password that the server will use when connecting to the CICS system.
 - 7) Change the value for the **Enable security** field from no to yes.
 - 8) Click **OK**. The changes are applied and the Properties Editor closes.
 - d. Expand **Management Zones > Network Zone > Configurations > Network Configuration > Name Servers** and select **Network Name Server**.
 - e. From the pop-up menu, select **Properties**, which opens the Properties Editor.
 5. (Optional): Enable security service for the client:
 - a. Expand **Management Zones > Network Zone > Configurations > Network Configuration > Client Styles**, and select **server name Default Client**.
 - b. From the pop-up menu, select **Properties**. The Properties Editor opens.
 - c. In this notebook:

- 1) Select the **Security Service** tab.
 - 2) Change the value for the **Enable security** field from **No** to **Yes**.
 - 3) Click **OK**. The changes are applied and the Properties Editor closes.
6. Activate the configuration:
- a. Expand **Management Zones > Sample Application Zone > Configurations**, and select **Sample Configuration**.
 - b. From the pop-up menu, select **Activate**, automatically start the application server. Wait for the completion message in the Action Console window before you continue.

Building and running the test application

1. Copy ACashAcctCli.cpp and its associated makefile, ACashAcctCli.mak from e:\CBroker\samples\InstallVerification\PAA\Application\ACashAcctCli into the e:\scenarios\ABeCashAcct\Working\NT directory under the current Object Builder source directory, and go to that directory.
2. Type set APP=ACashAcct;
3. Type nmake - f ACashAcctCli.mak to build the application.
Note: If your compile command fails due to an incorrect DB2 user ID and password error, run the following command before you run the make (AIX) or nmake (NT) command:

```
export IVB_DB2AUTH="USER test USING password"
set IVB_DB2AUTH=USER test USING password
```
4. When the build has finished, type ACashAcctCli to run the application.

RELATED CONCEPTS

Business object (*Programming Guide*)

Session Service (*Advanced Programming Guide*)

RELATED TASKS

"Generating code" on page 551

"Building the JAR files" on page 561

Tutorial: Creating a component for PA data (meet-in-the-middle)



Developing components for the CICS or IMS Application Adaptor is most automated when using the bottom-up approach to development. However, this may not be the most practical for your environment: you may want to run your component against a different backend (such as DB2) before you attempt to run it against a CICS or IMS system. In this situation, you would have defined both a business object and a data object in Object Builder, and your next task would be to connect it to a PA bean.

For a complete list of prerequisites that you must install on your system, see the *Getting Started with Component Broker* documentation for the appropriate platform.

In this exercise, we will use NT and AIX as the deployment platforms. It is assumed that you would have created a component to manage your data. That is, you would have already created the following objects:

- the business object file (CashAcct)
- the business object interface (CashAcct)
- the key (CashAcctKey)
- the copy helper (CashAcctCopy)
- the business object implementation (CashAcctBO)
- the managed object (CashAcctMO), which uses Session Service for this particular example

Note: The managed object does not have to be sessional. If the persistent object uses either the LU 6.2 APPC, IMS APPC, OTMA, or EXCI, then the managed object must be transactional. If it uses the Generic type, the managed object can be either sessional or transactional.

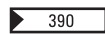
- the data object interface (CashAcctDO)

You will now import a PA bean.

Importing the PA bean

The class name for this bean is `paa.samples.cics.eci.acct.BeCashAcctPAO`.

1. In the Tasks and Objects pane, select the User-Defined PA Schemas folder, and from its pop-up menu, select **Import - Bean**. The Import Procedural Adaptor Bean wizard opens to the Bean Selection page.
2. You can choose to either type the name of the bean class, or select the JAR file containing the file, and then select the bean class. Select the **Enter bean name** radio button, and type the name of the class (`paa.samples.cics.eci.acct.BeCashAcctPAO`) in the field.
3. Click **Next**. The Names and Connectors page opens. Type the name of the module and the persistent object to be associated with the PA schema. You can also select the connector type to be used to access objects. Select ECI as the connector type. This is the type of connector `BeCashAcctPAO` uses.



When you select OS/390 as the development (target) platform (**Platform > Constrain > 390**), only the EXCI, OTMA, IMS APPC, and Generic connector types are available for selection.

Note: When you select either NT and 390, or AIX and 390 as the development platforms, all the connector types are available for selection. However, in this scenario you must not select 390 either alone, or in combination with one of NT or AIX, as the sample bean is for an ECI connector, and is not valid on OS/390: if you select 390, you will not be able to select `BeCashAcctPAOPO` as the type of your persistent object.

4. Click **Next**. Select `res_type` and `account_ID` (two of the properties of the bean) from the **Properties** box, and move them to the **Key Attributes** box, by clicking the `>>` button.
5. Click **Next**. The Variable Type Specification page opens. Accept the defaults.
6. Click **Finish**, and the bean will be imported into Object Builder. The `BeCashAcctPAO` schema and its associated persistent object `BeCashAcctPAOPO` appear in the tree under `User-Defined PA Schemas` folder.

To be able to connect a PA bean that you import with the component that you already have, you need to add a data object implementation that uses PAA Service (that one that uses procedural application adaptors).

To define a data object implementation, and connect the `BeCashAcctPAO` persistent object to it, follow these steps:

1. From the pop-up menu of the data object interface (`CashAcctDO`), select **Add Implementation**. The Data Object Implementation wizard opens to the Name and Platform page.
2. Accept the default names for the implementation class and its file. `CashAcctDOImpl` will be the name of the data object implementation you are defining.
3. Set the deployment platforms (the platforms on which this data object will be deployed) to NT and AIX.
Note:By default, the data object will be deployable to the set of platforms defined in the **Platforms > Constrain** menu. If you cannot select either NT, or AIX, or both as the deployment platforms, close the wizard, and first make sure that these options are selected on the **Platforms > Constrain** menu.
4. Click **Next**. The Behavior page opens.
5. In the **Environment** section, select **BOIM with any key**. “Environment” on page 249.
6. In the **Type of Persistence** section, select **Procedural Adaptors**. See the reference section: Type of Persistence.
7. In the **Data Access Pattern** section, accept the default, which is **Delegating**. See the reference section: “Data Access Pattern” on page 254.
8. Click **Next** till you open the Select Key and Copy Helper page. Select the key (`CashAcctKey`), and the copy helper (`CashAcctCopy`) to use with this implementation.
9. Click **Next**. The Associated Persistent Objects page. From the pop-up menu of the Persistent Object Instances folder, select **Add**.
10. Examine the list of persistent objects in the **Instance Name** field.

11. Select iBeCashAcctPAOPO from the list.
Note: If iBeCashAcctPAOPO does not appear in the list, check to make sure that the deployment platform for the data object implementation does indeed match that which you selected when you imported the PA bean. If it is different, change it as before using its **File Properties** pop-up menu option.
12. Now the PA persistent object is associated with the data object implementation, and the attributes of the implementation are mapped to corresponding ones of the PA bean. Similarly, methods of the implementation are mapped to corresponding methods of the PA bean. You can turn to the Attributes Mapping page, or the Methods Mapping page to examine the mappings.
13. Click **Finish**.

The data object implementation, CashAcctDOImpl will now appear under the CashAcctDO interface, and the BeCashAcctPAOPO persistent object will appear under the CashAcctDOImpl data object implementation.

Connecting the imported bean with an existing application

Keep the following points in mind:

- The deployment platforms of the data object implementation must match the deployment platforms of the persistent object. You choose the platforms of the data object implementation in its wizard, on the **Name and Platform** page. The platforms of the persistent object are chosen implicitly when you import the bean. At import time, the **Names and Connectors** page of the Procedural Adaptor Bean wizard reads in Object Builder's platform constraints. The appropriate connectors are enabled based on these constraints, and the platform of the schema and persistent object are determined by the connector chosen.
- Ensure that the Connector Type of the PA Schema (which you can view using the Properties wizard of the PA schema) is valid on all the platforms the data object implementation is to be deployed on.

For example, you cannot associate a persistent object whose PA Schema is LU 6.2 (APPC) with a data object implementation that is targetted for CB/390 (or with one that is targetted for multiple platforms including CB/390). This is because APPC is not supported as a connector type for CB/390. Similarly, a persistent object whose PA schema is OTMA cannot be associated with a data object implementation which is being targetted for NT, AIX, HP-UX or Solaris, since OTMA is only supported in CB/390.

In this tutorial, since we are using NT and AIX as the deployment platforms for the data object implementation, you could theoretically use PA schemas whose connector types are one of LU 6.2, HOD, ECI, SAP, or

Generic for association with the data object implementation. (The sample PA bean that we use is of the ECI connector type.)

- The only way in which you can associate a PA persistent object with an existing data object implementation is by using the Associated Persistent Object page of the data object implementation's Properties wizard. This is different for persistent objects that are backed by either Oracle or DB2: for them you can also use the **Add Persistent Object** menu item for that persistent object.

Exporting as XML

If you want to reuse the component that you just created in other scenarios, you can export it in XML format:

1. Click **File > Export Model**. The Export Model wizard opens.
2. Click **Finish**.

XML files that represent the elements of your component are exported to current project's \Working\Export directory.

Generating the application code

From the pop-up menu of the CashAcct file in the User-Defined Business Objects folder, select **Generate > All**. Code generation will begin, and you can monitor the progress in the bottom left corner of Object Builder's window.

Configuring the build

Add the client DLL:

1. From the pop-up menu of the Build Configuration folder, select **Add Client DLL**. The Client DLL wizard opens.
2. Type CashAcctC in the **Name** field.
3. Click the page title and turn to the Client Source Files page.
4. Click the **All >>** button to select all the client source files.
5. Click **Finish**.

The CashAcctC DLL will appear in the Build Configuration folder.

Add the server DLL:

1. From the pop-up menu of the Build Configuration folder, select **Add Server DLL**. The Server DLL wizard opens.
2. Type CashAcctS in the **Name** field.
3. Click **Next**.
4. Click the **>>** button to add the CashAcctC dll to the list of Libraries to link with.
5. Click **Next**.
6. Click the **All >>** button to select all the server source files.
7. Click **Finish**.

The CashAcctS DLL will appear in the Build Configuration folder.


Building the DLLs

Generate the configuration:

From the pop-up menu of the Build Configuration folder, select **Generate > All**. Code generation will begin.

Creating a container instance

1. From the pop-up menu of the Container Definition folder, select **Add Container Instance**. The Container wizard opens.
2. Type CashAcctContainer in the **Name** field.

 If you are developing an application intended for deployment on OS/390 (the **Platform > Constrain > 390** menu choice is selected), you are now done. The rest of the container definition is handled through the System Management user interface.

3. Click the arrow to the left of the page name, and select Service from the list. The Service page opens. Select **Use PAA Session Service**.
4. On the Service Details page, specify a name of your choice for the connection. Select **ECI** for the connector type used by the session.
5. Click **Finish**.

The CashAcctContainer will appear in the Container Definition folder.

Configuring the application

Add an application family:

1. From the pop-up menu of the Application Configuration folder, select **Add Application Family**. The Application Family wizard opens.
2. Type CashAcctApp in the **Name** field.
3. Click **Finish**.

The CashAcctApp family will appear in the Application Configuration folder.

Add an application:

1. From the pop-up menu of the CashAcctApp application family, select **Add Application**. The Application wizard opens.
2. Type CashAcct in the **Name** field.
3. Click **Finish**.

The CashAcct application will appear under the AcctApp family.

Add the application's managed object:

1. From the pop-up menu of the CashAcct application, select **Add Managed Object**. The Managed Object Configuration wizard opens.

2. Click the list box of the **Managed Object** field, and select CashAcctMO CashAcctMO from the list.
3. Click **Next**.
4. From the pop-up menu of the Implementations folder, select **Add**.
5. Click the list box of the **Data Object Implementation** field, and select CashAcctDOImpl from the list.
6. Click **Next**.
7. Click the list box of the **Name** field, and select CashAcctContainer from the list.
8. Click **Finish**.

The CashAcctMO managed object will appear under the Acct application.

Generate the application family:

From the pop-up menu of the CashAcctApp family, select **Generate**.

Building the CashAcct application (client and server)

Set up the environment:

You had added the location of the .jar file that contains the bean you import to your system class path variable: CLASSPATH. This location is required for import, and for the server to find the bean. So, reboot your system for the new environment variables to take effect. The server will then be able to find the bean.

Starting the build

1. Go to the working\NT directory. For this tutorial, this should be located in e:\scenarios\ABeCashAcct.
2. Type nmake -f all.mak
3. The CashAcct application should be built.
4. Copy CashAcctS.dll and CashAcctC.dll to the CBroker\bin directory to place them in your system path.

Installing the application

To install the application, you must be logged on to DCE, and the System Manager User Interface must be running. Then you load and configure the application by following the steps below.

Loading the application onto System Management

1. Start the System Manager User Interface, if it is not already started.
2. Enable Control actions by selecting **View > View Level > Control**.
3. Expand **Host Images**, and select your host name.

4. From the pop-menu, select **Load application** to open the Load application dialog. Select
e:\scenarios\BeCashAcct\Working\NT\CashAcctApp\CashAcctApp.ddl.

Configuring the application with System Management

1. Create the server:
 - a. From the **Tasks** menu, select **Create Servers**.
 - b. In the Management Zone window, select **Available Items > SampleApplication Zone**. Click **Next**.
 - c. In the Configuration window, select **Available Items > Sample Configuration**. Click **Next**.
 - d. In the Server Group window, type CashAcctServerGroup for the name of the server group. Click **Next**. If the server group does not exist, click **Yes** to create the server group.
 - e. In the Server window, type CashAcctServer for the Name of the server. Click **Finish**.
2. Configure the server:
 - a. Click on the **Tasks** menu and select **Configure Servers**.
 - b. In the Select Applications To Configure window, select **Available Applications > iPAAServices** and **CashAcct**. Click **Add** to move the applications to Applications To Configure. Click **Next**.
 - c. In the Management Zone window, select **Available Items > Sample Application Zone**. Click **Next**.
 - d. In the Configuration window, select **Available Items > Sample Configuration**. Click **Next**.
 - e. In the Select Servers To Configure Applications on window, select **Available Server Groups > CashAcctServerGroup**. Click **Add** to move the available server group to Servers To Configure Application On. Click **Finish**.
3. Configure the APPC connection:
 - a. Expand **Management Zones > Sample Application Zone > Configurations > Sample Configuration > APPC Connections** and select **APPC_CashAcct_Server**.
 - b. From the pop-up menu, select **Properties**. The Properties Editor opens.
 - c. Click the **Main** tab.
 - d. Change the **Fully qualified Local LU name** field to match the local LU6.2 LU that you will use to communicate with your CICS/IMS system (for example, PAA01001).
 - e. Change the **Fully qualified Partner LU name** field to match the partner LU6.2 LU that you will use to communicate with your CICS/IMS system (for example, USIBMZP.CICS4).

- f. Change the **Mode Name** field to match the mode name that you will use to communicate with your CICS/IMS system (for example, LU62PS).
 - g. Change the **Remote Procedure Type** field to match the type of program with which you will be communicating (for example, CICS_DPL or CICS_DTP). The CICS_DPL flavor appends eight bytes (converted to the target code page) that correspond to the CICS application to which the DTP program should EXEC CICS LINK.
 - h. Change the **Transaction Program Name** field to match the CICS transaction program (TP) that you will run (for example, BDPL or BDTP).
 - i. Change the **CICS Program Name** field to match the CICS program name that you will run under the transaction program (for example, BECASHAC).
 - j. Change the **transaction type** field to be either optimistic or pessimistic. Pessimistic initiates the conversation as sync-level 2 for the entire transaction while optimistic only talks sync-level 2 during the prepare and commit parts of the transaction.
 - k. Click **OK** to validate and accept the changes.
4. (Optional): Enable security service for the application server and the name server:
 - a. Expand **Management Zones > Sample Application Zone > Configurations > Sample Configuration > Server Groups**, and select **CashAcctServerGroup**.
 - b. From the pop-up menu, select **Properties** to open the Properties Editor.
 - c. In this notebook:
 - 1) Select the **Security Service** tab.
 - 2) Change the value for the **Enable security** field from no to yes.
 - 3) Change the **Delegate credentials** to None.
 - 4) Change the **Credential mapping** to Simple.
 - 5) Change the value for the **data system principal** field to the user ID that the server will use when connecting to the CICS system.
 - 6) Change the value for the **data system password** field to the password that the server will use when connecting to the CICS system.
 - 7) Change the value for the **Enable security** field from **no** to **yes**.
 - 8) Click **OK**. The changes are applied and the Properties Editor closes.
 - d. Expand **Management Zones > Network Zone > Configurations > Network Configuration > Name Servers** and select **Network Name Server**.

- e. From the pop-up menu, select **Properties**, which opens the Properties Editor.
5. (Optional): Enable security service for the client:
 - a. Expand **Management Zones > Network Zone > Configurations > Network Configuration > Client Styles**, and select **server name Default Client**.
 - b. From the pop-up menu, select **Properties**. The Properties Editor opens.
 - c. In this notebook:
 - 1) Select the **Security Service** tab.
 - 2) Change the value for the **Enable security** field from **No** to Yes.
 - 3) Click **OK**. The changes are applied and the Properties Editor closes.
 6. Activate the configuration:
 - a. Expand **Management Zones > Sample Application Zone > Configurations**, and select **Sample Configuration**.
 - b. From the pop-up menu, select **Activate**, automatically start the application server. Wait for the completion message in the Action Console window before you continue.

Building and running the test application

1. Copy CashAcctCli.cpp and its associated makefile, CashAcctCli.mak from e:\CBroker\samples\InstallVerification\PAA\Application\CashAcctCli into the e:\scenarios\ABeCashAcct\Working\NT directory under the current Object Builder source directory, and go to that directory.
2. Type set APP=CashAcct;
3. Type nmake - f CashAcctCli.mak to build the application.
4. When the build has finished, type CashAcctCli to run the application.

RELATED CONCEPTS

Business object (*Programming Guide*)

Session Service (*Advanced Programming Guide*)

RELATED TASKS

“Creating a PA schema by importing a PA bean” on page 862

“Generating code” on page 551

“Building the JAR files” on page 561

Creating a component for an inbound message

When you use the MQSeries application adaptor provided by Component Broker to create an application, the Component Broker business application integrates with non-Component Broker applications that are based directly on MQSeries.

These MQSeries application adaptor-backed Component Broker applications are of two types: those that either put messages to queues (outbound message applications), and those that get messages from queues (inbound message applications).

The business object of the component for an inbound message application communicates by retrieving messages from MQSeries applications.

If you are creating a new component, which talks by means of messages to MQSeries applications, you can create the entire component in Object Builder, starting with the business object interface and working your way down to a data object implementation that's derived from the component's state data.

To create the component directly in Object Builder, follow these steps:

1. "Creating a business object file" on page 775
2. "Adding a business object module" on page 777
3. "Adding a business object interface" on page 777
4. "Adding a key" on page 826
5. "Adding a copy helper" on page 830
6. "Adding a business object implementation and data object interface" on page 780
7. "Implementing methods" on page 752
8. "Adding a data object implementation" on page 807
9. "Adding a managed object" on page 871

For a scenario showing how to create a component for an inbound message application, see the "Tutorial: Creating an inbound message application" on page 188

For more information on using Object Builder with MQSeries-backed applications, see the related tasks for sections in the *MQSeries Application Adaptor Concepts and Development Guide*

RELATED CONCEPTS

Component (*Programming Guide*)

The Component Broker message processing model (*MQSeries Application Adaptor Concepts and Development Guide*)

An overview of message queueing with MQSeries (*MQSeries Application Adaptor Concepts and Development Guide*)

RELATED TASKS

“Chapter 4. Creating a component” on page 127

“Generating code” on page 551

“Tutorial: Creating an outbound message application” on page 203

Developing an MQSeries-backed application (*MQSeries Application Adaptor Concepts and Development Guide*)

Using Object Builder to develop the MQSeries sample applications (*MQSeries Application Adaptor Concepts and Development Guide*)

RELATED REFERENCES

The ICBMQGet interface (*MQSeries Application Adaptor Concepts and Development Guide*)

MQAA Transaction Service (page 596) (Container service)

“Naming objects” on page 128

“Internationalization of data” on page 132

Tutorial: Creating an inbound message application

Component Broker inbound message applications can be used to process a message queue, to which existing MQSeries applications can send messages. Inbound messages can be retrieved from a queue, but they can be retrieved only once (they are automatically deleted from the queue when they are retrieved).



MQSeries application adaptor support is available only in the Windows NT, Solaris and HP-UX environments. If you are developing a model for AIX, or OS/390, none of the MQSeries functions are available.

For a complete list of prerequisites that you must install on your system, see the *Getting Started with Component Broker* documentation for the appropriate platform.

To create an inbound message application, you will perform the following tasks:

- Add a business object file
- Add a business object interface
- Add a key
- Add a business object implementation
- Add a data object implementation
- Create the inbound message

- Add a managed object
- Generate code
- Define a message queue, and a queue manager in MQSeries
- Define a client DLL and a server DLL
- Build the DLLs
- Test your component
- Define an application family, and an application
- Configure the components with the application

Adding a business object file

1. From the pop-up menu of the User-Defined Business Objects folder in the Tasks and Objects pane, select **Add File**.
2. The Business Object File wizard opens to the Name and Platforms page. Specify a name for the business object file, and make sure that only **NT** is selected in the section: “Deployment platforms” on page 423.
3. Click **Finish**.

The business object file is created in the folder.

Adding a business object interface

1. From the pop-up menu of the business object file, select **Add Interface**.
2. The Business Object Interface wizard opens to the Name page. Type a name for the business object, and specify whether it is to be queryable. That is, whether attributes and methods are to be invoked on the managed object through a query.
3. Click **Next**. Define constructs for the interface, if required.
4. Click **Next**. The Interface Inheritance page opens. Replace the default parent interface (IManagedClient IManagedClient::IManageable) with IMessage IMessage::InboundMessage. The interface that you are defining will then be able to receive messages from queues that are managed by MQSeries application adaptors.
5. Click **Next** to define attributes on the Attributes page.
6. Click **Next** to define methods on the Methods page.
7. Click **Next** to specify any relationships this object has with other objects on the Relationships page.
8. Click **Next** to add comments if required for the object, and click **Finish**.

The business object interface is created in the User-Defined Business Objects folder.

Note: You do not have to add a copy helper when you create an object for an inbound message.

Adding a key for the business object

1. From the pop-up menu of the business object interface, select **Add Key**.
2. The Key wizard opens to the Name and Key Attributes page. Type a name for the key, or accept the default. Object Builder creates the default name from the name of the business object interface, adding Key as a suffix.
3. Click **Next**. The Implementation Inheritance page opens. Make sure that the key inherits from IMessageKey IMessageKey::InboundMessageKey.
4. Click **Next** to review the set of framework methods that will be implemented for the key.
5. Click **Next** to indicate which of the optional framework methods you want implemented for the key. You will have to provide your own implementation for these methods.
6. Click **Finish**.

The key appears in the User-Defined Business Objects folder.

You can add file adornments for the key, if you want to.

Adding a business object implementation

1. From the pop-up menu of the business object interface, select **Add Implementation**. The Business Object Implementation wizard opens to the Name and Data Access Pattern page. Appropriate implementation names are filled in for you (the business object file name and interface name plus BO). You can accept these defaults or replace them with your own names.
2. Set the types of behavior you want your business object to have. You can set the following properties:
 - “Pattern for Handling State Data” on page 245: Set the pattern to **Delegating**.
 - “Object Reference” on page 246
 - “Data Object Interface” on page 247: Select **Create a new one now**.
 - “Session Service” on page 248
 - “Deployment platforms” on page 423
3. Click **Next**. The Files to Include page opens. Object Builder automatically includes all the files that are required for MQSeries support.
4. Click **Next**. The Implementation Inheritance page opens. Make sure that IManagedClient::IManageable is listed as a parent under the Parent Class folder.
5. Click **Next**. The Implementation Language page opens. Select the language you want the business object to be implemented in. You can

select either Java or C++.

The default for this page is set in the Preferences notebook, on the Tasks and Objects page.

6. Click **Next**. The Attributes page opens. Specify any attributes you want to add to the business object implementation (in addition to the attributes you already specified in the business object interface).
7. Click **Next**. The Methods page opens. Specify any methods you want to add to the business object implementation (in addition to the methods you already specified in the business object interface).
8. Click **Next**. The Key and Copy Helper page opens. The key that you had defined, `InboundMessageKey`, is assigned for the implementation.
Note: No copy helper is required for the inbound message object implementation. If you create a copy helper, and select it, it will not be used unless you also create a specialized home (`InboundMessageQueue` does not use a copy helper).
9. Click **Next**. The Handle Selection page opens. If you select a handle, then the framework method `getHandleString` is implemented, which overrides the `getHandleString` method of `IManagedClient::IManageable`. The handle that you select determines the pattern that is used to form the string (that is, to turn the reference into a string, or to swizzle the pointer).
10. Click **Next**. If the business object implementation has parent classes with overrideable attributes, then the Attributes to Override page opens. You can use this page to select the attributes of the parent class that you want to override. The attributes of the defined key: `InboundMessage::objectKey`, `InboundMessage::correlatorKey`, and `InboundMessage::correlator` are automatically selected as attributes to be overridden in this business object implementation.
11. Click **Next**. If the business object implementation has parent classes with overrideable methods, then the Methods to Override page opens.
12. Click **Next**. The Relationships to Override page opens if the implementation inherits from parent implementations that have 1-n relationships defined among other objects. You can specify the relationship method implementations of the parent, if any, that you want to override.
13. Click **Next**. If the business object interface defines 1-n relationships, then the Object Relationships page opens. You can use this page to set the way that the object relationship will be implemented.
14. Click **Next**. The Data Object Interface page opens. The inbound message business object implementation is designed to always delegate the handling of the state data to the data object. So, select all attributes of the business object to be those of the data object as well (that is, to be state data of the component).


Note the following points:

- Appropriate data object names are filled in for you (the business object file name and interface name plus DO). You can accept these defaults or replace them with your own names.
 - If you implemented a one-to-many relationship as a **Local persistent reference**, then an attribute representing it appears here, so you can select to preserve it in the data object.
15. Click **Next**. The Data Object Methods page opens. Select the business object methods that you want to push down to the data object (that is, call equivalent methods to be defined in the data object).
 16. Click **Next**. The Summary of Framework Methods page opens.
Based on your selections on the previous pages of the wizard, this page displays the methods that your object implements. For example, if you selected a caching pattern to handle the essential state of your business object (on the first page), this list includes the `synchToDataObject` method that is required to keep the two sets of attributes synchronized. No action is needed.
 17. Click **Finish**. The business object implementation and data object interface appear in the User-Defined Business Objects folder, under your business object interface. The data object interface also appears in the User-Defined Data Objects folder.

Now that the business object implementation is defined, you can type the implementation code for each user-defined method.

Note:The business object implementation for this inbound object does not have the `insert()` method, and the `retrieve()` method is initialized to 0(off).

Adding the data object implementation

1. From the pop-up menu of the data object interface, select **Add Implementation**.
2. The Data Object Implementation wizard opens to the Name and Platform page. In the section: “Deployment platforms” on page 423, the platforms that are selected are those you had specified using **Platform > Constrain**.
 MQSeries capability is not supported on the AIX, OS/390, Solaris, or HP-UX platforms.
3. Click **Next**. The Behavior page opens. You can set the following properties: Set the types of behavior you want the data object to have. You can set the following properties:
 - “Environment” on page 249: Select **BOIM with any key**.
 - Type of Persistence: Select **MQ Adaptor**.
 - “Data Access Pattern” on page 254: The data access pattern is automatically set to **Local copy** and you cannot change it.

- “Handle for Storing Pointers” on page 255: Select a handle, or accept the default.
4. Click **Next**. The Implementation Inheritance page opens. `IMQAAExtLocalToServer IMQAAExtLocalToServer::IDataObject` is automatically set as the parent implementation class.
 5. Click **Next**. The Key and Copy helper page opens. `InboundMessageKey` is selected by default as the key to be used for the inbound message business object. No copy helper is required for the implementation.
 6. Click **Next**. The Summary of Framework Methods page opens. The CRUD methods along with the `setConnection()` method will be implemented for this data object implementation.
 7. Click **Finish**.

The data object implementation appears in the User-Defined Business Objects folder, beneath the data object interface.

Note: This data object implementation cannot be associated with a persistent object, and consequently, the pop-up menu choices of **Add Persistent Object and Schema** and **Select Persistent Object and Schema** are not available for it.

Creating the inbound message (providing code for the `InboundMessage::retrieve()` method)

Now, you must provide code for the `retrieve()` method. This method will be used to take the message off the message queue. You have to provide code (see the task “Implementing methods” on page 752), which extracts pieces of information from the byte sequence that is returned by the `ICBMQGet` interface, and then assign them to attributes of the data object implementation.

You must first add code for the `retrieve()` method so that it inspects the `correlatorKey` attribute in the data object implementation.

There are two ways in which you can create the inbound message, depending on whether the `correlatorKey` attribute is set. (The default key for an inbound message is `InboundMessageKey`, and one of its attributes is `correlatorKey`.)

If `correlatorKey` is set:

1. Construct an `ICBMQGet` object
2. Pass the `correlatorKey` attribute on the constructor

If `correlatorKey` is not set:

1. Construct the `ICBMQGet` object
2. Use the constructor which does not have the `correlatorKey` attribute

You create inbound messages by pulling them off a message queue through two methods **getNext()** and **get(in string correlator)**. Both these methods call `InboundMessage::retrieve()`. **getNext()** causes `retrieve()` to be called on an inbound message without setting its `correlatorKey` attribute. When you call **get(in string correlator)** on the home, the home creates an inbound message, then sets its `correlatorKey` attribute, and then calls `retrieve()` on it.

Adding the managed object

The managed object represents a message in the queue manager.

1. From the pop-up menu of the business object implementation, select **Add Managed Object**. The Managed Object wizard opens to the Names and Service page.

Appropriate names are filled in for you (the business object file name and interface name plus MO: `InboundMessage` `InboundMessage::InboundMessageMO`). You can accept these defaults or replace them with your own names.

2. The deployment platform (platform on which this managed object will be deployed) is set by default to **NT**.
3. Accept the default for the type of service, which is set to **Transaction Service**.
4. Click **Next**. The Implementation Inheritance page opens. By default, no inheritance is selected.
5. Click **Finish**.

The managed object appears in the User-Defined Business Objects folder, under your business object implementation.

Generating code

When you generate code for the business object, a new folder with the name of the key is created beneath NT folder. Within it is the key implementation and the key helper `.java` files. In the NT folder you will find the `.idl`, `.ih`, and `_I.cpp` files are created for the business object interface, implementation, key, key assistant, and managed object.

Now, you must define a message queue, and a queue manager in MQSeries.

Refer to the sections Using queues to interact with an MQSeries application and Appendix B. Configuring an MQSeries queue manager in the *MQSeries Application Adaptor Concepts and Development Guide*.

Defining a client DLL and a server DLL

Defining a client DLL

1. From the pop-up menu of the Build Configuration folder, click **Add Client DLL** to open the Client DLL wizard.

2. Name the DLL (InboundClient), and accept the default (NT) for the deployment platform.
3. Click the title and turn to the Client Source Files page. The items available include the business object interface (InboundMessage) and the key interface (InboundMessageKey).
4. Select all the client source files for the object, and add them to the **Items Chosen** list.
5. Click **Finish**.

The client DLL appears under the Build Configuration folder.

Defining a server DLL

1. From the pop-up menu of the Build Configuration folder, click **Add Server DLL** to open the Server DLL wizard.
2. Name the DLL (InboundServer), and accept the default (NT) for the deployment platform.
3. Click **Next** to turn to the Libraries to Link With page. InboundClient is listed in the **Items Available** box. Select InboundClient and add it to the **Items Chosen** list.
4. Click **Next** to turn to the Server Source Files page.
5. Select all the server source files for the object, and add them to the **Items Chosen** list. The items available include the business object implementation (InboundMessageBO) and the managed object (InboundMessageMO). You can use the **All Valid>>** button to move valid objects from the **Items Available** list to the **Items Chosen** list, and the **<<All Invalid** button to move invalid objects from the **Items Chosen** list to the **Items Available** list.
6. Click **Finish**.

Building the DLLs

To generate the makefiles, based on the configuration choices you made in the DLL wizards, follow these steps:

1. From the pop-up menu of the Build Configuration folder, click **Generate > All > All Targets**. This generates makefiles for all the DLL files defined in the folder and generates an all.mak file that calls the individual DLL makefiles. By choosing **All Targets**, you set the default behavior of the makefile, when it is built. You can still override this default when you build (for example, you can choose to build Java targets only, and override the default of all targets).

Note: You do not have to build the JAR files in the following situations:

- when the business object is implemented in C++; not Java
- when you do not want to code a client in Java

To build the DLL and (optionally) JAR files:

Note: You do not have to build the JAR files if you have a C++ business object; not a Java business object, or if you do not want to code the client in Java.


1. From the Build Configuration folder's pop-up menu, select **Build > Out-of-Date Targets > C++**.

You have to always generate C++ targets, even if you selected **Java** as your business object implementation language. The other objects of the component (such as the data object implementation and managed object) are still implemented in C++.

2. If you selected **Java** as your business object implementation language, then you have to also build the Java targets. From the Build Configuration folder's pop-up menu, select **Build > Out-of-Date Targets > Java**.

When you have a mixed-language application (some business objects are implemented in C++, some in Java) you must generate both C++ and Java targets for *all* components, even for those implemented in C++. The .jar files for C++ components allow Java components to interact with the C++ components on the server.

For this exercise, your application contains just a single component, so you only need to build the Java targets if you implemented the business object in Java.

 The clientIN.dll and serverIN.dll files are stored in the Working\NT\PRODUCTION directory.

If you have a Java business object, the clientIN.jar and serverIN.jar files are stored in the Working\NT\PRODUCTION directory.

If you had a Java client application, regardless of the language your component is implemented in, you would also select **Build > Out-of-Date Targets > Java Client Bindings** to generate Java client bindings (Working\NT\`<build style>`\JCB\JCBclientIN.jar). `<build style>` is one of the configuration directories: NOOPT, PRODUCTION, TRACE, TRACE_DEBUG. See Platform-specific information for more information.

Testing your component

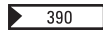
You will be using QuickTest to test the installed component with a QuickTest client application.

Defining an application family and application

An application family is a group of applications in System Management. An application defines a set of components that will operate together on the server. The application name you provide here is the name that will be used by System Management, when the application and its components are deployed.

Defining the application family

1. From the pop-up menu of the Application Configuration folder, select **Add Application Family**. The Application Family wizard opens to the Name page.
2. Type a name (InboundMessageAppFamily), and a description for the application family, and set its version.
3. Optionally, type names for the specific and non-specific DDL files that will be generated.

 When OS/390 is one of the deployment platforms, both the specific and non-specific DDL filenames must not exceed eight characters in length. The first character must be alphabetic, and the other characters (positions 2 to 8) must be alphanumeric.

The application family InboundMessageAppFamily appears in the Application Configuration folder.

Defining the application

Define the server application (InboundMessageApplication).

1. From the pop-up menu of the application family, InboundMessageAppFamily, in the Application Configuration folder, select **Add Application**. The Application wizard opens to the Name and Environment page.
2. Type a name (InboundMessageApplication), and a description for the application. The version of the application is set at the version of the application family.
3. Click **Finish**.

The application, InboundMessageApplication, appears under the application family, InboundMessageAppFamily.

Creating a new container instance

1. From the pop-up menu of the Container Definition folder, click **Add Container Instance**. The Container wizard opens to the Name page.
2. Type a name and description for the container. NT is selected as the deployment platform in the section: **Deployment Platforms**. See “Deployment platforms” on page 423.
3. In the **Number of Components** field, type an estimate for the number of managed objects this container will hold. This sets a lower limit on the size of the container’s hash table; additional space will be allocated when it is needed.
4. Click **Next**. The Workload Management page opens.

5. Specify whether the container is workload managing. If you check this option, you must also specify the policy group it will be configured with. For new policy groups, accept the default <New> entry.
6. Click **Next**. The Service page opens.
7. Select Use MQAA Transaction Service (page 596).
8. Click **Next**. The Services Details page now opens. Only the **Behavior for Methods Called Outside a Transaction** section is enabled. (See **“Behavior for Methods Called Outside a Transaction” on page 599**). Select **Throw an exception and abandon the call**. Then, type a name for the queue manager in the **Queue Manager Name** field.
9. Click **Next**. The Data Access Patterns page opens. Select the options on this page according to the options set for the objects the container will hold.

In the **Business Object** section, the default pattern is **Delegating**. It is recommended that you do not change it. (This option is set according to the pattern you defined for the business object implementation in the section **Pattern for Handling State Data** of the Business Object Implementation wizard’s Name and Data Access Pattern page.) (See **“Pattern for Handling State Data” on page 245**.)

In the **Data Object** section, **Local copy** is the default. You cannot change it since the data object implementation for the MQSeries application adaptor has no persistent object to delegate to. (This is the same as the option you select for the data object implementation’s **Data Access Pattern** in the Data Object Implementation wizard’s, Behavior page.) Since you selected **Delegating**, you have to indicate whether the data object uses Cache Service. Select the **Cache Service** check box if the data objects have their **Type of Persistence** set to either **DB2 Cache Service** or **Oracle Cache Service** (Data Object Implementation wizard, Behavior page).

10. Click **Finish**.

The new container is added to the Container Definition folder.

Configuring the components with the application

Configure InboundMessage’s managed object (InboundMessage InboundMessage::InboundMessageMO) with the application (InboundMessageApplication).

To add a managed object for the application, follow these steps:

1. From the Application Configuration folder, select your application.
2. From the application’s pop-up menu, select **Add Managed Object**. The Configure Managed Object wizard opens to the Selection page. Only managed object and primary key fields are filled in. Copy helper information blank.

The managed object `InboundMessageMO` `InboundMessageMO` is designated to the server DLL named `InboundServer`, and the primary key, `InboundMessageKey` `InboundMessageKey`, is designated to the client DLL, `InboundClient`.

3. Select the managed object (`InboundMessageMO`) from the drop-down list.
4. Click **Next**. The Data Object Implementations page opens.
5. From the Implementations pop-up menu, select **Add**.
6. Select the data object implementations that will be available to the application, and associated DLLs. Note that this is a packaging statement, and not a configuration statement. You can only select data object implementations whose type of persistence matches the service provided by the managed object.
7. Click **Next**. The Container page opens.

The only containers listed are those that are appropriate for the current managed object and selected data object implementations. If you did not create the container instance before you started configuring the managed object, you can select the **Create a new container for this managed object** check box to have Object Builder create a default container into which you can configure the managed object. The container will have behaviors that are appropriate for the managed object and selected data object implementations. If you select this check box, continue with step 8; otherwise continue with step 9.

Note: You have to ensure that the managed object is configured with a different container than that used by its home.

8. Click **Next**, and use the New Container page to define a container to use with this managed object. Indicate whether you want to use a workload managing container. If you select this option, then only workload managing containers are available in the Container list.

Note: If necessary, create a separate container instance for the managed object. If a managed object and its home are configured with the same container, the server will not activate.

9. Click **Next**. The Home page appears.
10. Define the home to use with this managed object. You can define a home instance of a default home provided with Component Broker, or define a home instance of a specialized home you created. If you specify a specialized home, you must also specify the DLL that contains it.

`IMessageHomeMO_InboundMessageQueueMO` is selected by default as the home.

`InboundMessageMMO::InboundMessageIMOFactory` is the default name in factory-finding registry

`InboundMessageMMO::InboundMessageIMOHome` is the default name in naming service registry

Only homes that are appropriate for the current managed object are shown. If your component is workload managed, you must select a workload managed home (BOIMHomeofRegWLMHomes, BOIMHomeofRegWLMQIHomes, or a specialized home that inherits from one of them). If your component is *not* workload managed, you can still select a workload managed home, although this will not make the component workload managed: the choice of container is what makes a component workload managed.

11. Select any other configuration options for the home
12. Click **Finish**. You have configured the managed object by choosing a copy helper and a key for it to work with, data object implementations for it to use, a container, a home, and the DLLs that contain it and the other objects. The managed object now appears in the Application Configuration folder, underneath the application you configured it for.

The InboundMessageMO managed object configuration appears under the InboundMessageApplication application, and the new container InboundMessageContainer appears in the Container Definition folder. You can review the properties of the container, including the service and data access patterns that have been selected for you, by clicking **Properties** from the container's pop-up menu.

Once you have finished adding managed objects to your server applications, and have completed the configuration of the applications in your application family, you can generate the installation image for your application family.

Generating the application installation information:

1. From the pop-up menu of the Application Configuration folder, click **Generate**.

The DDL that defines the applications for System Management is generated into
Working\platform\PRODUCTION\InboundMessageApplication.

Closing Object Builder:

1. Click **File > Save** to save your work.
2. Click **File > Exit** to close Object Builder.

Summary

You have created a component named InboundMessage that inherits from the InboundMessage interface.

You can find information on installing the component on the server in the *System Administration Guide*, "Configure a New Application Environment" chapter, or online in the topic Installing and configuring a new application.

You can find information on testing the installed component with a QuickTest client application in “Chapter 13. Testing applications with QuickTest” on page 611.

RELATED CONCEPTS

Components (*Programming Guide*)

The Component Broker message processing model (*MQSeries Application Adaptor Concepts and Development Guide*)

An overview of message queueing with MQSeries (*MQSeries Application Adaptor Concepts and Development Guide*)

Workload management (Using Object Builder) (*Advanced Programming Guide*)

RELATED TASKS

“Chapter 4. Creating a component” on page 127

“Generating code” on page 551

“Tutorial: Creating an outbound message application” on page 203

Developing an MQSeries-backed application (*MQSeries Application Adaptor Concepts and Development Guide*)

Using Object Builder to develop the MQSeries sample applications (*MQSeries Application Adaptor Concepts and Development Guide*)

RELATED REFERENCES

The ICBMQGet interface (*MQSeries Application Adaptor Concepts and Development Guide*)

MQAA Transaction Service (page 596) (Container service)

Creating a component for an outbound message

When you use the MQSeries application adaptor provided by Component Broker to create an application, the Component Broker business application integrates with non-Component Broker applications that are based directly on MQSeries.

These MQSeries application adaptor-backed Component Broker applications are of two types: those that either put messages to queues (outbound message applications), and those that get messages from queues (inbound message applications).

The business object of the component for an outbound message application communicates by putting messages to MQSeries message queues.

If you are creating a new component, which communicates by means of messages to MQSeries applications, you can create the entire component in

Object Builder, starting with the business object interface and working your way down to a data object implementation that's derived from the component's state data.

To create the component directly in Object Builder, follow these steps:

1. "Creating a business object file" on page 775
2. "Adding a business object module" on page 777
3. "Adding a business object interface" on page 777
4. "Adding a key" on page 826
5. "Adding a copy helper" on page 830
6. "Adding a business object implementation and data object interface" on page 780
7. "Implementing methods" on page 752
8. "Adding a data object implementation" on page 807
9. "Adding a managed object" on page 871

For a scenario showing how to create a component for an outbound message application, see "Tutorial: Creating an outbound message application" on page 203

RELATED CONCEPTS

Components (*Programming Guide*)

The Component Broker message processing model (*MQSeries Application Adaptor Concepts and Development Guide*)

An overview of message queueing with MQSeries (*MQSeries Application Adaptor Concepts and Development Guide*)

RELATED TASKS

"Chapter 4. Creating a component" on page 127

"Generating code" on page 551

"Tutorial: Creating an outbound message application" on page 203

Developing an MQSeries-backed application (*MQSeries Application Adaptor Concepts and Development Guide*)

Using Object Builder to develop the MQSeries sample applications (*MQSeries Application Adaptor Concepts and Development Guide*)

RELATED REFERENCES

The ICBMQGet interface (*MQSeries Application Adaptor Concepts and Development Guide*)


MQAA Transaction Service (page 596) (Container service)

"Naming objects" on page 128

"Internationalization of data" on page 132

Tutorial: Creating an outbound message application

Outbound messages are persistent in that they are written to a message queue, but they can neither be retrieved nor deleted from that queue as outbound messages.

 MQSeries application adaptor support is available only in the Windows NT, Solaris and HP-UX environments. If you are developing a model for AIX, or OS/390, none of the MQSeries functions are available.

For a complete list of prerequisites that you must install on your system, see the *Getting Started with Component Broker* documentation for the appropriate platform.

To create an outbound message application, you will perform the following tasks:

- Add a business object file
- Add a business object interface
- Add a copy helper
- Add a business object implementation
- Add a data object implementation
- Create the outbound message
- Add a managed object
- Generate code
- Define a message queue, and a queue manager in MQSeries
- Define a client DLL and a server DLL
- Build the DLLs
- Test your component
- Define an application family, and an application
- Configure the components with the application

Adding a business object file

1. From the pop-up menu of the User-Defined Business Objects folder in the Tasks and Objects pane, select **Add File**.
2. The Business Object File wizard opens to the Name and Platforms page. Specify a name for the business object file, and make sure that only **NT** is selected as the deployment platform in the section: "Deployment platforms" on page 423.
3. Click **Finish**.

The business object file is created in the folder.

Adding a business object interface

1. From the pop-up menu of the business object file, select **Add Interface**.
2. The Business Object Interface wizard opens to the Name page. Type a name for the business object, and specify whether it is to be queryable. That is, whether attributes and methods are to be invoked on the managed object through a query.
3. Click **Next**. Define constructs for the interface, if required.
4. Click **Next**. The Interface Inheritance page opens. Replace the default parent interface (IManagedClient IManagedClient::IManageable) with IMessage IMessage::OutboundMessage. The interface that you are defining will then be able to receive messages from queues that are managed by MQSeries application adaptors.
5. Click **Next** to define attributes on the Attributes page.
6. Click **Next** to define methods on the Methods page.
7. Click **Next** to specify any relationships this object has with other objects on the Relationships page.
8. Click **Next** to add comments if required for the object, and click **Finish**.

The business object interface is created in the User-Defined Business Objects folder.

Note: You do not have to add a key when you create an object for an outbound message.

Adding a copy helper

1. From the business object interface's pop-up menu, click **Add Copy Helper** to open the Copy Helper wizard.
2. Accept the default name, and select all attributes and add them to the **Copy Helper Attributes** list.
3. Click **Next**. The Implementation Inheritance page opens. The copy helper inherits by default from IMessage::OutboundMessageTemplate.
4. Click **Finish**.

The copy helper appears in the User-Defined Business Objects folder.

You can add file adornments for the copy helper, if you want to.

Adding a business object implementation

1. From the pop-up menu of the business object interface, select **Add Implementation**. The Business Object Implementation wizard opens to the Name and Data Access Pattern page.

Appropriate implementation names are filled in for you (the business object file name and interface name plus BO). You can accept these defaults or replace them with your own names.

2. Set the types of behavior you want your business object to have. You can set the following properties:
 - “Pattern for Handling State Data” on page 245: The pattern is set to **Delegating**.
 - “Object Reference” on page 246
 - “Data Object Interface” on page 247: Select **Create a new one now**.
 - “Session Service” on page 248
 - “Deployment platforms” on page 423
3. Click **Next**. The Implementation Inheritance page opens. Make sure that `IManagedClient::IManageable` is listed as a parent under the Parent Class folder.
4. Click **Next**. The Implementation Language page opens. Select the language you want the business object to be implemented in. You can select either Java or C++.

The default for this page is set in the Preferences notebook, on the Tasks and Objects page.

5. Click **Next**. The Attributes page opens. Specify any attributes you want to add to the business object implementation (in addition to the attributes you already specified in the business object interface).
6. Click **Next**. The Methods page opens. Specify any methods you want to add to the business object implementation (in addition to the methods you already specified in the business object interface).
7. Click **Next**. The Key and Copy Helper page opens. The copy helper that you had defined, `OutboundMessageCopy`, is assigned for the implementation.
8. Click **Next**. The Handle Selection page opens. If you select a handle, then the framework method `getHandleString` is implemented, which overrides the `getHandleString` method of `IManagedClient::IManageable`. The handle that you select determines the pattern that is used to form the string (that is, to turn the reference into a string, or to swizzle the pointer).
9. Click **Next**. If the business object implementation has parent classes with overrideable attributes, then the Attributes to Override page opens. You can use this page to select the attributes of the parent class that you want to override.

The `OutboundMessage::correlator` attribute is automatically selected to be overridden by this business object implementation, and so is the `objectKey` attribute.

10. Click **Next**. If the business object implementation has parent classes with overrideable methods, then the Methods to Override page opens.
11. Click **Next**. The Relationships to Override page opens if the implementation inherits from parent implementations that have 1-n relationships defined among other objects. You can specify the relationship method implementations of the parent, if any, that you want to override.
12. Click **Next**. If the business object interface defines 1-n relationships, then the Object Relationships page opens. You can use this page to set the way that the object relationship will be implemented.
13. Click **Next**. The Data Object Interface page opens. The outbound message business object implementation is designed to always delegate the handling of the state data to the data object. So, you must select all attributes of the business object to be those of the data object as well (that is, to be state data of the component).
Appropriate data object names are filled in for you (the business object file name and interface name plus DO). You can accept these defaults or replace them with your own names.
14. Click **Next**. The Data Object Methods page opens. Select the business object methods that you want to push down to the data object (that is, call equivalent methods to be defined in the data object).
15. Click **Next**. The Summary of Framework Methods page opens.
Based on your selections on the previous pages of the wizard, this page displays the methods that your object implements. For example, if you selected a caching pattern to handle the essential state of your business object (on the first page), this list includes the `synchToDataObject` method that is required to keep the two sets of attributes synchronized. No action is needed.
16. Click **Finish**. The business object implementation and data object interface appear in the User-Defined Business Objects folder, under your business object interface. The data object interface also appears in the User-Defined Data Objects folder.


Now that the business object implementation is defined, you can type the implementation code for each user-defined method.

Note:The business object implementation for this outbound object has the `insert()` method.

Adding the data object implementation

1. From the pop-up menu of the data object interface, select **Add Implementation**.
2. The Data Object Implementation wizard opens to the Name and Platform page. The deployment platforms that are selected in the **Deployment**

Platforms section are the same as the constraint platforms (those you had specified using **Platform > Constrain**). See “Deployment platforms” on page 423.

3. Click **Next**. The Behavior page opens. You can set the following properties: Set the types of behavior you want the data object to have. You can set the following properties:
 - “Environment” on page 249: Select **BOIM with any key**.
 - Type of Persistence: Select **MQ Adaptor**.
 You will be able to select **MQSeries Adaptor** only if neither AIX nor 390 are selected as the constraint platforms.
 - “Data Access Pattern” on page 254: The data access pattern is automatically set to **Local copy** and you cannot change it.
 - “Handle for Storing Pointers” on page 255: Select a handle, or accept the default.
4. Click **Next**. The Implementation Inheritance page opens. `IMQAAExtLocalToServer IMQAAExtLocalToServer::IDataObject` is automatically set as the parent implementation class.
5. Click **Next**. The Key and Copy helper page opens. `OutboundMessageCopy` is selected by default as the copy helper to be used for the outbound message business object. No key is required for the implementation.
6. Click **Next**. The Summary of Framework Methods page opens. The insert, retrieve, update, and delete methods, along with the `setConnection()` method will be implemented for this data object implementation.
7. Click **Finish**.

The data object implementation appears in the User-Defined Business Objects folder, beneath the data object interface.

Note: This data object implementation cannot be associated with a persistent object, and consequently, the pop-up menu choices of **Add Persistent Object and Schema** and **Add Persistent Object and Schema** are not available for it.

Creating the outbound message

Now, you must provide code for the `insert()` method. Object Builder provides implementation code for this method: a message body is created from the data object attributes, and it is sent to a message queue (the `ICBMQPut` object is created and the message is put to the queue). The message body is simply a data structure which contains the contents of the message. Since Object Builder does not know the layout of the data structure, it shows (in the sample code) how the attributes of the data object could be written to a hypothetical data structure in memory. You then have to modify this code to fit the actual structure of the message body. See the task “Implementing methods” on page 752.

Adding the managed object

The managed object represents a message in the queue manager.

1. From the pop-up menu of the business object implementation, select **Add Managed Object**. The Managed Object wizard opens to the Names and Service page.

Appropriate names are filled in for you (the business object file name and interface name plus MO). You can accept these defaults or replace them with your own names.

2. The deployment platform (platform on which this managed object will be deployed) is set by default to **NT**.
3. Accept the default for the type of service, which is set to **Transaction Service**.
4. Click **Next**. The Implementation Inheritance page opens. By default, no inheritance is selected.
5. Click **Finish**.

The managed object appears in the User-Defined Business Objects folder, under your business object implementation.

Generating code

When you generate code for the business object, a new folder with the name of the key is created beneath NT folder. Within it is the key implementation and the key helper .java files. In the NT folder you will find the .idl, .ih, and _I.cpp files are created for the business object interface, implementation, key, key assistant, and managed object.

Now, you must define a message queue, and a queue manager in MQSeries.

Refer to the sections Using queues to interact with an MQSeries application and Appendix B. Configuring an MQSeries queue manager in the *MQSeries Application Adaptor Concepts and Development Guide*.

Defining a client DLL and a server DLL

Defining a client DLL

1. From the pop-up menu of the Build Configuration folder, click **Add Client DLL** to open the Client DLL wizard.
2. Name the DLL (OutboundClient), and accept the default (NT) for the deployment platform.
3. Click the title and turn to the Client Source Files page. The items available include the business object interface (OutboundMessage) and the copy helper interface (OutboundMessageCopy).
Note: If you did define a key for the object, the key interface will also be present in the **Items Available** list.

4. Select all the client source files for the object, and add them to the **Items Chosen** list.
5. Click **Finish**.

The client DLL appears under the Build Configuration folder.

Defining a server DLL

1. From the pop-up menu of the Build Configuration folder, click **Add Server DLL** to open the Server DLL wizard.
2. Name the DLL (OutboundServer), and accept the default (NT) for the deployment platform.
3. Click **Next** to turn to the Libraries to Link With page. OutboundClient is listed in the **Items Available** box. Select OutboundClient and add it to the **Items Chosen** list.
4. Click **Next** to turn to the Server Source Files page.
5. Select all the server source files for the object, and add them to the **Items Chosen** list. The items available include the business object implementation (OutboundMessageBO) and the managed object (OutboundMessageMO).
6. Click **Finish**.

Building the DLLs

To generate the makefiles, based on the configuration choices you made in the DLL wizards, follow these steps:

1. From the pop-up menu of the Build Configuration folder, click **Generate > All > All Targets**. This generates makefiles for all the DLL files defined in the folder and generates an all.mak file that calls the individual DLL makefiles. By choosing **All Targets**, you set the default behavior of the makefile, when it is built. You can still override this default when you build (for example, you can choose to build Java targets only, and override the default of all targets).

To build the DLL and (optionally) JAR files:

1. From the Build Configuration folder's pop-up menu, select **Build > Out-of-Date Targets > C++**.


You always need to generate C++ targets, even if you selected **Java** as your business object implementation language. The other objects of the component (such as the data object implementation and managed object) are still implemented in C++.

2. If you selected **Java** as your business object implementation language, then you also need to build the Java targets. From the Build Configuration folder's pop-up menu, select **Build > Out-of-Date Targets > Java**.

When you have a mixed-language application (some business objects are implemented in C++, some in Java) you must generate both C++ and Java

targets for *all* components, even for those implemented in C++. The .jar files for C++ components allow Java components to interact with the C++ components on the server.

For this exercise, your application contains just a single component, so you only need to build the Java targets if you implemented the business object in Java.

 The clientIN.dll and serverIN.dll files are stored in the Working\NT\PRODUCTION directory.

If you have a Java business object, the clientIN.jar and serverIN.jar files are stored in the Working\platform\PRODUCTION directory.

If you had a Java client application, regardless of the language your component is implemented in, you would also select **Build > Out-of-Date Targets > Java Client Bindings** to generate Java client bindings (working\platform\<<build style>\JCB\JCBclientIN.jar). <build style> is one of the configuration directories: NOOPT, PRODUCTION, TRACE, TRACE_DEBUG. See Platform-specific information for more information.

For this exercise, you will be using QuickTest to run and test your application, and you do not need to build Java client bindings.

Testing your component

You will be using QuickTest to test the installed component with a QuickTest client application.

Defining an application family and application

Defining the application family(OutboundMessageAppFamily)

1. From the pop-up menu of the Application Configuration folder, select **Add Application Family**. The Application Family wizard opens to the Name page.
2. Type a name (OutboundMessageAppFamily), and a description for the application family, and set its version.

The application family OutboundMessageAppFamily appears in the Application Configuration folder.

Defining the application(OutboundMessageApplication).

1. From the pop-up menu of the application family, InboundMessageAppFamily, in the Application Configuration folder, select **Add Application**. The Application wizard opens to the Name and Environment page.

2. Type a name (OutboundMessageApplication), and a description for the application. The version of the application is set at the version of the application family.
3. Click **Finish**.

The application, OutboundMessageApplication, appears under the application family, OutboundMessageAppFamily.

Creating a new container instance

1. From the pop-up menu of the Container Definition folder, click **Add Container Instance**. The Container wizard opens to the Name page.
2. Type a name and description for the container. **NTIS** is selected as the deployment platform in the **Deployment Platforms** section. (See “Deployment platforms” on page 423.)
3. In the **Number of Components** field, type an estimate for the number of managed objects this container will hold. This sets a lower limit on the size of the container’s hash table; additional space will be allocated when it is needed.
4. Click **Next**. The Workload Management page opens.
5. Specify whether the container is workload managing. If you check this option, you must also specify the policy group it will be configured with. For new policy groups, accept the default <New> entry.
6. Click **Next**. The Service page opens.
7. Select Use MQAA Transaction Service (page 596).
8. Click **Next**. The Services Details page now opens. Only the **Behavior for Methods Called Outside a Transaction** section is enabled. (See “Behavior for Methods Called Outside a Transaction” on page 599) Select **Throw an exception and abandon the call**. Then, type a name for the queue manager in the **Queue Manager Name** field.
9. Click **Next**. The Data Access Patterns page opens. Select the options on this page according to the options set for the objects the container will hold.

In the **Business Object** section, the default pattern is **Delegating**. It is recommended that you do not change it. (This option is set according to the pattern you defined for the business object implementation in the section **Pattern for Handling State Data** of the Business Object Implementation wizard’s Name and Data Access Pattern page. See “Pattern for Handling State Data” on page 245.)

In the **Data Object** section, **Local copy** is the default. You cannot change it since the data object implementation for the MQSeries application adaptor has no persistent object to delegate to. (This is the same as the option you select for the data object implementation’s **Data Access Pattern** in the Data Object Implementation wizard’s, Behavior page.)

Since you selected **Delegating**, you have to indicate whether the data object uses Cache Service. Select the **Cache Service** check box if the data objects have their **Type of Persistence** set to either **DB2 Cache Service** or **Oracle Cache Service** (Data Object Implementation wizard, Behavior page).

10. Click **Finish**.

The new container is added to the Container Definition folder.

Configuring the components with the application

Configure OutboundMessage's managed object (OutboundMessage OutboundMessage::OutboundMessageMO) with the application (OutboundMessageApplication).

To add a managed object for the application, follow these steps:

1. From the Application Configuration folder, select your application.
2. From the application's pop-up menu, select **Add Managed Object**. The Configure Managed Object wizard opens to the Selection page. Only managed object and copy helper fields are filled in.
3. Select the managed object (OutboundMessageMO) from the drop-down list.
4. Click **Next**. The Data Object Implementations page opens.
5. From the Implementations pop-up menu, select **Add**.
6. Select the data object implementations that will be available to the application, and associated DLLs. Note that this is a packaging statement, and not a configuration statement. You can only select data object implementations whose type of persistence matches the service provided by the managed object.
In this case, select the data object implementation that you defined earlier.

7. Click **Next**. The Container page opens.

The only containers listed are those that are appropriate for the current managed object and selected data object implementations. If you did not create the container instance before you started configuring the managed object, you can select the **Create a new container for this managed object** check box to have Object Builder create a default container into which you can configure the managed object. The container will have behaviors that are appropriate for the managed object and selected data object implementations. If you select this check box, continue with step 8; otherwise continue with step 9.

Note: You have to ensure that the managed object is configured with a different container than that used by its home.

8. Click **Next**, and use the New Container page to define a container to use with this managed object. Indicate whether you want to use a workload

managing container. If you select this option, then only workload managing containers are available in the Container list.

Note: If necessary, create a separate container instance for the managed object. If a managed object and its home are configured with the same container, the server will not activate.

9. Click **Next**. The Home page appears.
10. Define the home to use with this managed object. You can define a home instance of a default home provided with Component Broker, or define a home instance of a customized home you created. If you specify a customized home, you must also specify the DLL that contains it. `IMessageHomeMO_OutboundMessageQueueMO` is selected by default as the home. `OutboundMessageMMO::OutboundMessageIMOFactory` is the default name in factory-finding registry. `OutboundMessageMMO::OutboundMessageIMOHome` is the default name in naming service registry. Only homes that are appropriate for the current managed object are shown. If your component is workload managed, you must select a workload managed home (`BOIMHomeofRegWLMHomes`, `BOIMHomeofRegWLMQIHomes`, or a specialized home that inherits from one of them). If your component is *not* workload managed, you can still select a workload managed home, although this will not make the component workload managed: the choice of container is what makes a component workload managed.
11. Select any other configuration options for the home
12. Click **Finish**. You have configured the managed object by choosing a copy helper and a key for it to work with, data object implementations for it to use, a container, a home, and the DLLs that contain it and the other objects. The managed object now appears in the Application Configuration folder, underneath the application you configured it for.

The `OutboundMessageMO` managed object configuration appears under the `OutboundMessageApplication` application, and the new container `OutboundMessageContainer` appears in the Container Definition folder. You can review the properties of the container, including the service and data access patterns that have been selected for you, by clicking **Properties** from the container's pop-up menu.

Once you have finished adding managed objects to your server applications, and have completed the configuration of the applications in your application family, you can generate the installation image for your application family.

Generating the application installation information:

1. From the pop-up menu of the Application Configuration folder, click **Generate**.

The DDL that defines the applications for System Management is generated into
Working\platform\PRODUCTION\OutboundMessageApplication.

Closing Object Builder:

1. Click **File > Save** to save your work.
2. Click **File > Exit** to close Object Builder.

Summary

You have created a component named OutboundMessage that inherits from Person, and which provides persistence in a database table both for its own attributes, and for the attributes it inherits from Person. You have implemented the attributes duplication pattern for inheritance with persistence.

You can find information on installing the component on the server in the System Administration Guide, “Configure a New Application Environment” chapter, or online in the topic Installing and configuring a new application.

You can find information on testing the installed component with a QuickTest client application in “Chapter 13. Testing applications with QuickTest” on page 611.

RELATED CONCEPTS

Components (*Programming Guide*)

The Component Broker message processing model (*MQSeries Application Adaptor Concepts and Development Guide*)

An overview of message queueing with MQSeries (*MQSeries Application Adaptor Concepts and Development Guide*)

Workload management (Using Object Builder) (*Advanced Programming Guide*)

RELATED TASKS

“Chapter 4. Creating a component” on page 127

“Generating code” on page 551

“Tutorial: Creating an inbound message application” on page 188

Developing an MQSeries-backed application (*MQSeries Application Adaptor Concepts and Development Guide*)

Using Object Builder to develop the MQSeries sample applications (*MQSeries Application Adaptor Concepts and Development Guide*)

RELATED REFERENCES

The ICBMQPut interface (*MQSeries Application Adaptor Concepts and Development Guide*)

MQAA Transaction Service (page 596) (Container service)

Reusing existing objects

When you have existing data in a database that you want to use as part of a new design, you need to assemble a component out of some objects that are designed and some that are imported. You can also reuse objects in multiple components; for example, several components might access the same database table.

When you assemble a component from existing objects, the main tasks involve mapping the data and attributes of the objects you connect. The objects can be created as follows:

- Create business objects in the **User-Defined Business Objects** folder.
- Import an SQL file and add DB persistent objects from the schemas created in the **DBA-Defined Schemas** folder.
- Import PA beans and add additional PA persistent objects from the PA schemas in the **User-Defined PA Schemas** folder.
- Create data objects in the **User-Defined Data Objects** folder by adding the data objects from the persistent objects. (Select **Add Data Object** from the pop-up menu of the persistent object.)

The objects can be mapped as follows:

- Map business objects to data objects in the business object implementation's wizard when the business object is not associated with a data object. (Use **Select Data Object Interface** from the pop-up menu of the implementation, and in the Data Object Interface Connection wizard, map the attributes and methods of the business object to those of the data object.)
- Map data objects to persistent objects in the data object implementation's wizard. (Add the persistent object created from the schema on the Associated Persistent Objects page, and then map the attributes and methods of the data object to those of the persistent object.)

When you reuse a schema, whether it is a DB schema or a PA schema, you can either add multiple persistent objects (adding a new persistent object to the schema for each use), or add a single persistent object to the schema, and reuse the persistent object. In either case, you add the persistent object to the schema, and then connect the persistent object with the data object in the data object implementation's wizard.

RELATED CONCEPTS

Component (*Programming Guide*)

"DDL" on page 137

"Procedural adaptor bean (PA bean)" on page 159

RELATED TASKS

“Working with components” on page 697

“Customizing referential integrity” on page 714

Creating a local-only object

A local-only object is not manageable, and cannot take advantage of Component Broker services. It can be used by a component or client process locally, but cannot be part of a component’s public interface, where it could be accessed remotely.

To create a local-only object in Object Builder, follow these steps:

1. “Creating a local-only object file” on page 217
2. “Adding a local-only object module” on page 218
3. “Adding a local-only object” on page 219

For an in-depth introduction and accompanying sample, see “Tutorial: Creating local-only objects” on page 220.

For rules on naming these objects, see “Naming objects” on page 128.

RELATED CONCEPTS

“Local-only objects”

RELATED TASKS

“Chapter 4. Creating a component” on page 127

RELATED REFERENCES

“Naming objects” on page 128

“Internationalization of data” on page 132

Local-only objects

A local-only object does not take advantage of the management features of the managed object framework (MOFW). It can have attributes and methods, which can be accessed by the component or client process that creates it. It cannot have persistent state, and it cannot be part of a business object’s public interface. However, it can be part of another local-only object’s public interface.

In Object Builder, a local-only object is defined as a file, optional module, and object implementation. A local-only object does not have separately defined interface and implementation. Because it has no persistent state, it has no data object; and because it is not manageable, it has no managed object.

RELATED CONCEPTS

“Object Builder” on page 1
Component (*Programming Guide*)

RELATED TASKS

“Creating a local-only object” on page 216

Creating a local-only object file

A local-only object file can contain one or more local-only objects, optionally organized into modules.

To create a local-only object file, follow these steps:

1. In the Tasks and Objects pane, locate the Local-Only Objects folder.
2. From the folder’s pop-up menu, click **Add File** to open the Local-Only Object File wizard.
3. Type a name for the file. This will be the name of the .idl file that defines the interface to the local-only object.
4. Click **Next**. The Constructs page opens.

Use the Constructs pop-up menu to add constants, enumerations, exceptions, structures, typedefs, and unions. Any constructs you add are scoped to every object in the file.

Note: To use the construct as a type within another construct, you must first click **Finish** and then re-open the wizard before you can use the type. The construct is not added to the current model until you click **Finish**.

5. Click **Next**. The Files to Include page opens.
IManagedLocal is included by default. This is the correct choice for a local-only object that represents a base class in your design. Also include the files for any referenced or related objects. Object Builder will automatically emit include statements for parent interfaces and local-only objects, and other IDL types that are used by constructs and interfaces within the local-only object file.
6. Click **Next**. The Comments page opens. Type any comments you want to include as comment lines in your generated IDL code.
7. Click **Finish**. The wizard closes, and your file is added to the Local-Only Objects folder. You can now add modules or local-only objects to the file.

Once you have created the file, you can modify it by selecting **Properties** from its pop-up menu. The Local-Only Object File wizard opens again, with your selections preserved.

RELATED CONCEPTS

“Local-only objects” on page 216

RELATED TASKS

- “Creating a local-only object” on page 216
- “Adding a local-only object module”
- “Adding a local-only object” on page 219
- “Tutorial: Creating local-only objects” on page 220

RELATED REFERENCES

- “Internationalization of data” on page 132
- “Naming objects” on page 128

Adding a local-only object module

If you plan to add multiple objects to a single file, you may want to store the objects in separate modules. Any constructs you add to a module are scoped only to the objects within that module. To add a module to a file, follow these steps:

1. From the Local-Only Objects folder in the Tasks and Objects pane, select your local-only object file.
2. From the file’s pop-up menu, select **Add Module**. The Local-Only Object Module wizard opens to the Name page.
3. Type a name for the module.
4. Click **Next**. The Constructs page opens.
Use the Constructs pop-up menu to add enumerations, exceptions, structures and so on.
Note: To use the construct as a type within another construct, you must first click **Finish** and then reopen the wizard before you can use the type. The construct is not added to the current model until you click **Finish**.
5. Click **Next**. The Comments page opens. Type any comments you want to include as comment lines in your generated code.
6. Click **Finish**. The wizard closes, and your module is added to the Local-Only Objects folder, underneath the file.

You can now add local-only objects to the module.

RELATED CONCEPTS

- “Local-only objects” on page 216

RELATED TASKS

- “Creating a local-only object” on page 216
- “Adding a local-only object” on page 219
- “Tutorial: Creating local-only objects” on page 220

RELATED REFERENCES

“Internationalization of data” on page 132

“Naming objects” on page 128

Adding a local-only object

To add a local-only object to a file or module, follow these steps:

1. In the Local-Only Objects folder in the Tasks and Objects pane, select the file or module that will contain the local-only object.
2. From the pop-up menu for the file or module, select **Add Interface**. The Local-Only Object wizard opens to the Name page.
3. Type a name for the local-only object.
4. Click **Next**. The Constructs page opens.

Use the Constructs pop-up menu to add constants, enumerations, exceptions, typedefs, structures, or unions. Any constructs you add are scoped to this object only.

Note: To use the construct as the type of an attribute, method return, method exception, or construct member, you must first click **Finish** and then re-open the wizard before you can use the type. The construct is not added to the current model until you click **Finish**.

5. Click **Next**. The Interface Inheritance page opens.
By default, the local-only object inherits from `IManagedLocal::INonManageable`. This is the correct choice for a local-only object that represents a base class in your design. You can also choose `IManagedLocal::ILocalOnly`. (The difference is that `INonManageable` objects are streamable.) If your object had a parent, you would specify the parent object on this page.

6. Click **Next**. The Attributes page opens.
To specify attributes for your interface, select **Add** from the Attributes pop-up menu (for example, the `CarPolicy` interface could have the attributes “make” and “model”).
Note: For most attribute types, a default initializer value is provided. When there is no suitable default (for example, an attribute whose type is an enumeration), you should assign your own initializer value, if necessary.

7. Click **Next**. The Methods page opens.
To specify methods for your interface, select **Add** from the Methods pop-up menu.
8. Click **Next**. The Summary of Framework Methods page opens. Review the framework methods the object has.
9. Click **Next**. The Comments page opens. Type any comments you want to include as comment lines in your generated code.

10. Click **Finish**. Your new local-only object is added to the Local-Only Objects folder, with the attributes and methods you specified.

You should now see the object's interface in the Inheritance pane, and any methods you defined for your interface should appear under the **User-Defined Methods** folder in the Methods pane. Click on a method or attribute in the Methods pane to provide an implementation for it in the Source pane.

RELATED CONCEPTS

"Local-only objects" on page 216

RELATED TASKS

"Creating a local-only object" on page 216

"Tutorial: Creating local-only objects"

RELATED REFERENCES

"Internationalization of data" on page 132

"Naming objects" on page 128

Tutorial: Creating local-only objects

Objectives

To create local-only (non-managed) objects that can be used to get or set multiple attributes of the component

To add methods to a component that use the local-only objects



To generate the code for the component and its objects

To build the DLLs for the component and its objects

To define the application configuration information for the component

Before you begin

You need the following installed on your system:

- A CB Server
- A CB Client
- CB tools, including Samples
- DB2 Universal Database
- DB2 SDK
-  VisualAge for C++ and (for Java applications) VisualAge for Java
-  IBM C and C++ Compilers for AIX V3.6.4 and (for Java applications) VisualAge for Java

For a complete list of prerequisites that you must install on your system, see the *Getting Started with Component Broker* documentation for the appropriate platform.

You should be familiar with the Component Broker programming model, as described in the *IBM Component Broker Programming Guide*. At minimum, you should have read chapter 1, the introduction to the programming model.

Sample files

There are equivalent samples for this exercise. The samples include:

- A BusinessObjects project that contains the finished Object Builder model
- A zipped Rational Rose model that contains definitions for the business object, key, copy helper, and data object interface
- Documentation for the sample, including instructions on running a Java client for the sample

The samples are in the following directories (under <CBroker>, the install directory):

For C++:

```
samples\Tutorial\Fundamentals\LocalOnly\BusinessObjects
samples\Tutorial\Fundamentals\LocalOnly\Rose\LocalOnly.zip
samples\Tutorial\Fundamentals\LocalOnly\Docs\LocalOnly.html
```

There is no Java sample for this exercise.

Description

This exercise describes how to create a transient component, and then enhance its interface with the use of local-only objects.

You can create local-only objects for a variety of different purposes. These particular local-only objects will be used by the client, or by other components on the server, to either get all the attributes of the component, or set all the read-write attributes of the component, with a single call. This is similar to the function a copy helper performs in component creation (collapsing multiple calls to the server into a single call). In fact, copy helpers are special cases of local-only objects.

The following tasks do not give explicit instructions for every step, but should at least get you into the right wizards. If you are experiencing problems, click the Help button within a wizard, or go to the Help pulldown in Object Builder.

For this exercise, you will complete the following tasks:

1. Creating the project
2. Creating a business object interface
3. Adding a key and copy helper
4. Adding a business object implementation

5. Adding a data object implementation
6. Adding a managed object
7. Creating the “set” local-only object
8. Creating the “get” local-only object
9. Adding methods to the component
10. Implementing the methods
11. Generating the code
12. Defining a client DLL and server DLL
13. Defining an application family and application
14. Configuring the component with the application
15. Testing the application

Creating the project

Create a sample project to hold your work. For example, e:\tutorials\localonly

1. Start Object Builder.
2. In the Open Project wizard, type a name and path for the project directory.
3. Click **Finish**.
4. When asked whether you want to create a new project, click **Yes**.

Creating the business object interface

Define a business object file (Sample1):

1. From the User-Defined Business Objects folder’s pop-up menu, click **Add File** to open the Business Object File wizard.
2. Name the file.
3. Click **Finish**. The file now appears under the folder.

Add a module (Sample1):

1. From the pop-up menu of the file, click **Add Module** to open the Business Object Module wizard.
2. Name the module.
3. Click **Finish**. The module now appears under the file.

Add an interface (Agent). The business object interface defines the interface for the whole component. It will be contained in the Sample1 module.

1. From the pop-up menu of the module, click **Add Interface** to open the Business Object Interface wizard.
2. Name the interface csAgent.
3. Click the page title and turn to the Attributes page.
4. Add the following attributes:
 - readonly float commissions

- float commPercent
 - float pendingPaycheck
 - string agentName
 - readonly long id
5. Set the size of agentName to 100. You should always provide a size for string attributes.
 6. Click **Next** and turn to the Methods page.
 7. Add the following method:
 - void payCommission (in float amount)
 8. Click **Finish**.

The Agent interface now appears under the Samplonl module. The attributes and methods appear in the Methods pane, when the interface is selected.

The attributes are represented as paired get and set methods, except for the id attribute, which was defined as read-only, and therefore only has a get method.

The interface does not have any business logic associated with it. The implementation of the interface is defined separately, in the business object implementation.

Adding a key and copy helper

Add a key (AgentKey). The key allows client applications to locate or create instances of the component on the server. It consists of an attribute or attributes of the business object interface that uniquely identify an instance of the component. In this case, the Agent id attribute is the appropriate choice.

1. From the interface's pop-up menu, click **Add Key** to open the Key wizard.
2. Accept the default name; select the id attribute and add it to the **Key Attributes** list.
3. Click **Finish**.

Even though the id attribute of the business object interface is read-only, the id attribute of the key is both readable and writable. The client application can set the value of the id on the key, and use it either to initialize a new instance of the component, or to locate an existing instance of the component, on the server.

AgentKey appears under Agent.

Add a copy helper (AgentCopy). The copy helper allows client applications to create and initialize instances of the component on the server, using one call to set numerous attributes, rather than one call per attribute.

1. From the interface's pop-up menu, click **Add Copy Helper** to open the Copy Helper wizard.

2. Accept the default name; select all listed attributes and add them to the **Copy Helper Attributes** list.
3. Click **Finish**.

AgentCopy appears under Agent.

Adding a business object implementation and data object interface

Add a business object implementation (AgentBO) and data object interface (AgentDO). The business object implementation contains the actual business logic of the component, including the method implementations. Any state data attributes (those attributes that cannot be deduced or derived from other attributes) become part of the data object interface. The separation of business logic (in the business object) from state data (in the data object) allows issues such as data persistence and integrity to be partitioned from the rest of the business logic.

1. From the interface's pop-up menu, click **Add Implementation** to open the Business Object Implementation wizard.
2. Change the **Pattern for Handling State Data** to **Delegating**. This is easier to debug than the **Caching** pattern.
3. Click the page title and turn to the Implementation Language page.
4. Select the language you want the business object to be implemented in (**C++** or **Java**).
5. Click the page title and turn to the Key and Copy Helper page. The appropriate key and copy helper are already selected.
6. Click the page title and turn to the Data Object Interface page.
7. Select all attributes and add them to the **State Data** list (to be preserved in the data object).
8. Click **Finish**.

AgentBO appears under Agent, and AgentDO appears under AgentBO.

Adding a method implementation

Type in the implementation for the method `payCommission`. The appropriate implementation logic for the data get and set methods, and for framework methods required by the programming model, are calculated for you by Object Builder. You only need to provide implementations for methods you explicitly defined.

1. Click on the business object implementation. The Methods pane shows the user-defined method `payCommission`, and the various user-defined attributes (in the form of paired get and set methods).
2. Click on `long id()`. The `id` attribute only has a get method, because it is read-only, as defined in the business object interface. The provided implementation appears in the Source pane.

- Review the provided implementation for `long id()`. The get method for the `id` attribute delegates directly to its equivalent attribute in the data object, as defined by the **Delegating** pattern chosen in the business object implementation.
- In the Methods pane, click on the `payCommission` method: `void payCommission (in float amount)`. The signature for the method appears in the source pane, based on the definition you provided in the business object interface and the language you selected in the business object implementation. The method does not have an implementation yet. You must provide the implementation for user-defined methods.
- In the Source pane, provide the following implementation for `payCommission`:

C++

```
float tmp;
tmp = amount * iDataObject->commPercent();
iDataObject->commissions(tmp);
iDataObject->pendingPaycheck(iDataObject->pendingPaycheck() + tmp);
```

Java

```
float tmp;
tmp = amount * iDataObject.commPercent();
iDataObject.commissions(tmp);
iDataObject.pendingPaycheck(iDataObject.pendingPaycheck() + tmp);
```

You can continue to the next step. When you click on other objects, the implementation will disappear from the Source pane, but the code you typed is now part of the project model, and will be generated as part of the source code for the component (in the file `Samplon1B0_I.cpp` for a C++ business object, or `_AgentB0Base.java` for a Java business object).

Adding a data object implementation

Add a data object implementation (`AgentDOImpl`). The data object implementation defines the way in which you want to handle the component's state data.

- From `AgentDO`'s pop-up menu, click **Add Implementation** to open the Data Object Implementation wizard.
- Accept the default name and platform settings, and click **Next** to turn to the Behavior page.
- Set the following patterns:
 - Environment - BOIM with any key**
The component will be locatable by its key (instead of being locatable by a UUID).
 - Type of Persistence - Transient**
The component's data will not have persistence beyond the lifespan of the component instance. As a result, the component does not require a

persistent object (which would manage the mapping of the data to a persistent datastore, such as a database).

- **Data Access Pattern - Local copy**

This is the only option available for a transient data object. There is no persistent datastore to delegate to.

4. Click the page title and turn to the Key and Copy Helper page. AgentKey and AgentCopy should already be selected.
5. Click **Finish**.

AgentDOImpl appears under AgentDO.

Adding a managed object

Add a managed object (AgentMO). The managed object mediates the interaction between the client application and the component, or between other components and this component. It exposes the business object interface to the client application and other components, and accesses any relevant services before and after a call.

1. From the business object implementation's pop-up menu, click **Add Managed Object** to open the Managed Object wizard to the Name and Service page.
2. Accept all the defaults and click **Finish**.
3. Click **File > Save** to save your work before continuing to the next step.

AgentMO appears under AgentBO.

You have now completely defined a transient component. You are ready to go on to the next half of the exercise, in which you define the local-only objects, and customize the component to use them.

Creating the "set" local-only object

The SetRWAgentLO object will have attributes that match all the writable attributes of the Agent component. A client or component can create an instance of the local-only object, then set all its attributes, convert the object into a string, and pass the string to the Agent component's setRWAttributes method. The method takes the string and reconstitutes the SetRWAgentLO object, then updates component attributes with the values in the local-only object.

Local-only objects cannot be exposed as part of a component's interface: typically, if they must be passed between components or between the client and a component, they are passed as strings of data that can be reconstructed by the receiver.

Create the local-only object file (Sampgslo):

1. From the pop-up menu of the Local-Only Objects folder, click **Add File** to open the Local-Only Objects File wizard.
2. Name the file.
3. Click **Finish**.

Add the local-only object module (Sampgslo):

1. From the pop-up menu of the file, click **Add Module** to open the Local-Only Objects Module wizard.
2. Name the module.
3. Click **Finish**.

Add the local-only object (SetRWAgentLO), with attributes that match the read-write attributes of Agent, as well as an attribute that represents a reference to its matching Agent component.

1. From the pop-up menu of the module, click **Add Local-Only Object** to open the Local-Only Object wizard.
2. Name the object SetRWAgentLO.
3. Click the title and turn to the Interface Inheritance page. By default, local-only objects inherit from IManagedLocal::INonManageable. This is the same inheritance a copy helper has, and provides the object with streamability, something it would not get from ILocalOnly. Object Builder overrides the `internalize_from_stream` and `externalize_to_stream` methods, which makes for much easier flattening and rehydrating of the object's state.
4. Click **Next** and turn to the Attributes page.
5. Add the following attributes:
 - float commPercent
 - float pendingPaycheck
 - string agentName


You do not need to create entries for the other Agent attributes, because they are read-only: you cannot set them once the component is created.

6. Set the size of agentName to 100. This matches the size of the agentName in Agent. You should always provide a size for string attributes.
7. Add the following attribute:
 - Samptran::Agent agent

This serves as a reference to the Agent component whose attributes the object will set.

8. Click **Finish**.

The SetRWAgentLO object appears under the module. You can review its attributes and framework methods in the Methods pane. By clicking on the

attributes or methods, you can review their default implementations in the Source pane. The attributes and methods in the Methods pane have both C++ and Java implementations. You can switch between languages by clicking on the down-pointing “v” button  on Method pane’s toolbar.

Local-only objects, unlike business objects and data objects, are *not* separated into interface and implementation.

Compare the attributes and methods of the local-only object to those of the copy helper in the User-Defined Business Objects pane. They should look very similar: they have the same inheritance, most of the same attributes, and in fact a similar purpose.

Creating the “get” local-only object

The GetAllAgentLO object will have attributes that match all the attributes of the Agent component. A client or component calls an Agent’s getAllAttributes method, which creates the local-only object, populates its attributes with the component attribute values, then converts the object to a string and returns it to the caller. The client or calling component receives the string and reconstitutes the local-only object, which it can then query locally for component data.

Add the local-only object (GetAllAgentLO) to the existing local-only object module (Sampgslo), with attributes that match all the attributes of Agent, as well as an attribute that represents a reference to a matching Agent component.

1. From the pop-up menu of the module, click **Add Local-Only Object** to open the Local-Only Object wizard.
2. Name the object GetAllAgentLO.
3. Click the title and turn to the Interface Inheritance page. By default, local-only objects inherit from IManagedLocal::INonManageable. This is the same inheritance a copy helper has, and provides the object with streamability, something it would not get from ILocalOnly. Object Builder overrides the internalize_from_stream and externalize_to_stream methods, which makes for much easier flattening and rehydrating of the object’s state.
4. Click **Next** and turn to the Attributes page.
5. Add the following attributes:
 - read-only commission
 - float commPercent
 - float pendingPaycheck
 - string agentName
 - read-only long id

6. Set the size of `agentName` to 100. This matches the size of the `agentName` in `Agent`. You should always provide a size for string attributes.
7. Add the following attribute:
 - `Sampran::Agent agent`

This serves as a reference to the `Agent` component whose attributes the object will set.

8. Click **Finish**.

The `GetAllAgentLO` object appears under the module. You can review its attributes and framework methods in the `Methods` pane. By clicking on the attributes or methods, you can review their default implementations in the `Source` pane. It is substantially similar to the previously created local-only object, and serves a complementary purpose.

Adding methods to the component

Now that you have created the local-only objects, you can add methods to the `Agent` component that will work with the objects. You need to add two methods:

- `setRWAttributes`
Takes a stringified `SetRWAgentLO` object, reconstitutes it, and updates the component attributes.
- `getAllAttributes`
Creates a `getAllAgentLO` object, updates its attributes, stringifies it, and returns it.

Define the methods:

1. From the pop-up menu of the `Agent` business object interface, click **Properties** to open the `Business Object Interface` wizard.
2. Click the title and turn to the `Methods` page.
3. Add the following methods:
 - `public ::ByteString getAllAttributes()`
 - `public setRWAttributes(::ByteString)`

The `IManagedClient` `ByteString` type is available in the type pull-down for the return type and parameter type.

4. Click **Finish**.

The methods appear in the `Methods` pane for the `Agent` interface. You are now ready to add implementations for them.

Implement `setRWAttributes(::ByteString)`:

1. Click on `AgentBO` (the implementation for the `Agent` interface).

- In the Methods pane, click on `setRWAttributes`. A skeleton implementation appears in the Source pane.
- Add the following implementation to the Source pane:

C++

```

Sampgslo::SetRWAgentLO_var AgentSetC
    = Sampgslo::SetRWAgentLO::_create();
AgentSetC->fromString(objString);
iDataObject->commPercent(AgentSetC->commPercent());
iDataObject->pendingPaycheck(AgentSetC->pendingPaycheck());
::CORBA::String_var stringVar1 = AgentSetC->agentName();
iDataObject->agentName(stringVar1);
Samplonl::csAgent_var AgentAsObject = AgentSetC->agent();
iDataObject->agent(AgentAsObject);

```

Implement `::ByteString getAllAttributes()`:

- Click on `AgentBO`.
- In the Methods pane, click on `getAllAttributes`. A skeleton implementation appears in the Source pane.
- Add the following implementation to the Source pane:

C++

```

Sampgslo::GetAllAgentLO_var GetAllAgentLOVar
    = Sampgslo::GetAllAgentLO::_create();
GetAllAgentLOVar->commissions(iDataObject->commissions());
GetAllAgentLOVar->commPercent(iDataObject->commPercent());
GetAllAgentLOVar->pendingPaycheck(iDataObject->pendingPaycheck());
GetAllAgentLOVar->id(iDataObject->id());
::CORBA::String_var stringVar1 = iDataObject->agentName();
GetAllAgentLOVar->agentName(stringVar1);
Samplonl::Agent_var AgentAsObject = iDataObject->agent();
GetAllAgentLOVar->agent(AgentAsObject);
return GetAllAgentLOVar->toString();

```

You are now ready to generate the code. The generated code will contain the method implementations you provided.

Generating the code

You are now ready to generate the code for the objects you have defined. Generated code will appear in your project's `\Working\platform` directory (for example, `e:\tutorials\transient\working\NT`)

- From the pop-up menu of the User-Defined Business Objects folder, click **Generate > All**.
The code generation can take several minutes.
- From the pop-up menu of the Local-Only Objects folder, click **Generate > All**.

3. Once code generation is complete, review the contents of the `Working\platform` directory. All the source files for the component and for the local-only objects have been generated, and you can now define how to build them.

Defining a client DLL and server DLL

Configure the build process that will create the client and server DLLs.

The client DLL provides client applications with access to the component on the server, using the key and copy helper, and will also contain the local-only objects. You must also include the business object interface, which defines the methods and attributes of the component that the client can access.

Define the client DLL configuration and client DLL or library file (for this exercise, name them both `SamlonIC`).

1. From the pop-up menu of the Build Configuration folder, click **Add Client DLL** to open the Client DLL wizard to the Name and Options page.
2. Name the configuration. This is the name that uniquely identifies the configuration node.
3. Name the library as well. This is the name for the makefile and for the resulting DLL file. You can create multiple configuration nodes that produce different versions of the same DLL. For example, you could set different compile and link options to produce a development version and a deployment version of the same DLL.
4. For this exercise, the default configuration options are sufficient.
5. Click the title and turn to the Client Source Files page.
6. Select all the client files and add them to the **Items Chosen** list.
7. Click **Finish**.

The `SamlonIC` DLL configuration appears under the Build Configuration folder.

The server DLL is installed on the server to deploy the component, making it available for access by client applications and other components.

Define the server DLL configuration and server DLL file (for this exercise, name them both `SamlonIS`).

1. From the pop-up menu of the Build Configuration folder, click **Add Server DLL** to open the Server DLL wizard.
2. Name the configuration and target DLL file.
3. Click **Next** to turn to the Libraries to Link With page.
4. Add the client DLL file. The server DLL needs access to the client DLL because the client DLL defines the component's interface, and the component implementation inherits from it.

5. Click **Next** to turn to the Server Source Files page.
6. Select all the server source files for the component, and add them to the **Items Chosen** list.
7. Click **Finish**.

The SamlonIS DLL configuration appears under the Build Configuration folder.

Building the DLLs

To generate the makefiles, based on the configuration choices you made in the DLL wizards:

1. From the pop-up menu of the Build Configuration folder, click **Generate > All > All Targets**. This generates makefiles for all the DLL files defined in the folder and generates an all.mak file that calls the individual DLL makefiles. By choosing **All Targets**, you set the default behavior of the makefile, when it is built. You can still override this default when you build (for example, you can choose to build Java targets only, and override the default of all targets).

To build the DLL and (optionally) JAR files:

1. From the Build Configuration folder's pop-up menu, select **Build > Out-of-Date Targets > C++**.

You always need to generate C++ targets, even if you selected **Java** as your business object implementation language. The other objects of the component (such as the data object implementation and managed object) are still implemented in C++.

2. If you selected **Java** as your business object implementation language, then you also need to build the Java targets. From the Build Configuration folder's pop-up menu, select **Build > Out-of-Date Targets > Java**.

When you have a mixed-language application (some business objects are implemented in C++, some in Java) you must generate both C++ and Java targets for *all* components, even for those implemented in C++. The .jar files for C++ components allow Java components to interact with the C++ components on the server.

For this exercise, your application contains just a single component, so you only need to build the Java targets if you implemented the business object in Java.

 The SamlonC.dll and SamlonIS.dll files are stored in the working\NT\PRODUCTION directory.

 The libSamlonC.so and libSamlonIS.so files are stored in the working/AIX/PRODUCTION directory.

If you have a Java business object, the SamlonlC.jar and SamlonlS.jar are stored in the working\platform\PRODUCTION directory.

If you had a Java client application, regardless of the language your component is implemented in, you would also select **Build > Out-of-Date Targets > Java Client Bindings** to generate Java client bindings (working\platform\<build style>\JCB\JCBSamlonlC.jar). <build style> is one of the configuration directories: NOOPT, PRODUCTION, TRACE, TRACE_DEBUG. See Platform-specific information for more information.

For this exercise, you will be using QuickTest to run and test your application, which has its own build process that includes generation of client bindings.

Defining an application family and application

Define the application family (SamplonlF). An application family groups a set of applications so they can be installed as a unit.

1. From the pop-up menu of the Application Configuration folder, click **Add Application Family** to open the Application Family wizard.
2. Name the application family.
3. Click **Finish**.

The SamplonlF application family appears under the Application Configuration folder.

Define the server application (SamplonlApp). An application defines a set of components that will operate together on the server. The application name you provide here is the name that will be used by System Management, when the application and its components are deployed.

1. From the pop-up menu of the application family, click **Add Application** to open the Application wizard.
2. Name the application.
3. Click **Finish**.

The SamplonlApp application appears under the SamplonlF application family.

Configuring the component with the application

Configure the component's managed object (SamplonlMO SamplonlMO::AgentMO) with the application (SamplonlApp), including the home (BOIMHomeOfRegHomes) that will be used to find and create it, and the container (TransientObjects) that will provide it with access to services.

1. From the pop-up menu of the application, click **Add Managed Object** to open the Managed Object Configuration wizard.

2. In the **Managed Objectlist**, select the component's managed object. The other lists should be automatically populated with the appropriate selections.
3. Click **Next** and select the data object implementation. Click **Add Another**. The appropriate data object implementation and DLL will be selected. If there were more than one data object implementation for the component, or it was being built into more than one DLL, you would click **Add Another** again to add the extra information.
4. Click **Next** to turn to the Container page.
5. Check the **Create a new container** option. An extra page is inserted into the wizard, to allow you to name the new container and specify its policies.
6. Click **Next** to turn to the Create New Container page.
7. Name the container, and set its behavior for methods called outside a transaction.

Most of the container's behavior is defined for you, based on the managed object you are configuring, and the data object implementation you selected.

8. Click **Next** and review the home. The appropriate home is already selected, and the default options are acceptable. If there were more than one appropriate home, you could choose the home to use.
9. Click **Finish**.

The AgentMO managed object configuration appears under the SamplonApp application.

Generate the application installation information:

1. From the pop-up menu of the Application Configuration folder, click **Generate**.

The DDL that defines the applications for System Management is generated into `Working\platform\PRODUCTION\SamplonIF`.

Testing the application

You can test the application using QuickTest, with the help of some additional Java client files that ship with the samples. For instructions on setting up and running QuickTest with your application, see the samples documentation (under <CBroker>, the install directory):

For C++:

`samples\Tutorial\Fundamentals\LocalOnly\Docs\LocalOnly.html`

For Java:

Summary

You have created two local-only objects, and added methods to a component that uses them. A client application, or another component, can use the local-only objects with the methods to query all the attributes of the component with a single call, or set all the writable attributes of the component with a single call.

You can find information on installing the component on the server in the System Administration Guide, “Configure a New Application Environment” chapter, or online in the topic Installing and configuring a new application.

You can find information on testing the installed component with a QuickTest client application in “Chapter 13. Testing applications with QuickTest” on page 611.

Tracking data types in models

You can view the different data types that exist in a model, and the manner in which they are used within a model.

For either task, you must first open the Type Browser. Follow one of these methods:

- From the **File** menu, select **Display Type Browser**. The Type Browser opens.
- Click the **Browse** button next to the **Type** field when you are either defining or editing attributes and methods of either the interface, or the implementation; of either business objects, or data objects. You can also use the **Browse** button to open the Type Browser while you either define or edit constructs at file, module, or interface level in the case of business objects and data objects; and at the file and module level in the case of compositions.

Note: The search mechanism of the Type Browser will find the type no matter the level at which it is nested. That is, the type may be contained in an interface, which is itself contained in a module, the module being contained in a file.

To get a filtered view of the different data types in the model, follow these steps:

1. Open the Type Browser.
2. You can filter the types by the following different categories:
 - Type name pattern
 - Type scope

- Object scope
- Types to display

To filter by type name, follow these steps:

- Type either the name of a data type, or part of a type name in the **Pattern** field.
- Use the asterisk as a wild card to represent a group of characters. For example, if the types `round`, `goround`, and `merrygoround` exist, and you type `*round`, types named `round`, `goround`, and `merrygoround` will be found (that is, all types that end with the pattern `round`.) If you type `*go*`, the types `goround`, and `merrygoround` will be found (that is, all types that contain the pattern `go`.)
- To find types that begin with a particular group of characters, just type the characters in the field. (You do not have to add the asterisk as part of the search pattern, at the end.) For example, to find data types whose names begin with the letter `a`, just type `a` in the field; to find those whose names start with the characters `admin`, just type `admin` in the field.

To filter by type scope, follow these steps:

- In the **Type Scope** section, indicate whether you want to see the types that exist in all the projects, or only those that are only defined in the current project.
- Indicate your choice by selecting one of the radio buttons:
 - Show types from all projects**
 - Show types from current project only.**

To filter by object scope, follow these steps:

- In the **Object Scope** section, indicate whether you want to see either all objects, only the local-only objects (those that are not manageable, and have no persistent state - those that have no associated managed object, or data object), or only the framework objects.
- Indicate your choice by selecting one of the following radio buttons:
 - Show all objects**
 - Show local-only objects**
 - Show framework objects**

You can further filter the types to be displayed, after you have specified criteria in the previous sections by selecting whether you want to see types that have been defined as constructs, or interfaces. Indicate your choice by selecting one or more of the following check boxes:

- **Interfaces**
- **Structures**
- **Typedefs**
- **Enumerations**

- **Constants**
- **Unions**
- **Exceptions**

Note the following points:

- Only the **Exceptions** check box is available, and all other constructs types are not selectable when you are defining exceptions for methods. (Methods page of either the Business Object Interface wizard, or the Data Object Interface wizard.)
- Since exceptions cannot be used as attribute or construct types, the **Exceptions** check box is not available when you invoke the Type Browser when you are defining either attributes or constructs.

The types are listed in the panel. When you select a type from this panel, all types with that base name are listed in the panel beneath it. You can differentiate the scope at which the different similarly named types are defined by their qualifying names. For example, if a type named `thisint` is defined in the object interface (`thisint`), which is itself defined within a module (`thismod`), the fully qualified name of the type will be listed as:

```
thisfile thismod thisint thisint
```

where `thisfile` is the file within which the module `thismod` is defined.

To view the different ways in which a type is used within the model, follow these steps:

1. Open the Type Browser.
2. Filter the types according to the set of criteria you define by specifying a string pattern for the type name, and using the various options that are available with the **Show Options** button.
3. Select a type from the filtered list.
4. Click the **Show Usage** button. The **Type Usage** dialog box opens. See the reference: "Type Usage" on page 238. This box lists objects for which any of a set of criteria are met by the selected type. Each criterion has its separate list. If the type does not satisfy any of these criteria, you are informed that no usage relationships exist for that type.

Note: In the case of either key or copy helper classes, the only objects that can use them are business objects. Hence, only business object implementations, if there are any that are using those classes, will be listed as satisfying those criteria.

RELATED CONCEPTS

"Projects and models" on page 17

"Attributes" on page 698

"User-defined methods" on page 751

“Get and set methods” on page 755
“Framework methods” on page 757
“Special framework methods” on page 758
“Constructs” on page 770
“Local-only objects” on page 216

RELATED TASKS

“Working with attributes” on page 697
“Working with methods” on page 750
“Working with constructs” on page 769
“Creating a local-only object” on page 216
“Tutorial: Creating local-only objects” on page 220

RELATED REFERENCES

“Type Usage”

Type Usage

This dialog box informs you how the selected type is used by the model. It lists one or more of the following usage patterns, with the list of objects that fall under each pattern, depending on how the type is used in the model:

- Is inherited by
- Is an attribute type in
- Is a method return type in
- Is a parameter type of a method in
- Is a member of structures
- Is a member of unions
- Is the data type of typedefs
- Is used in configured managed objects
- Key is used by
- Copy helper is used by

If the type does not satisfy any of these criteria, you are informed that no usage relationships exist for that type.

Note: In the case of either key or copy helper classes, the only objects that can use them are business objects. Hence, only business object implementations, if there are any that are using those classes, will be listed as satisfying those criteria.

RELATED TASKS

“Tracking data types in models” on page 235
“Setting Object Builder preferences” on page 27

File and method adornments

►CHANGED

An *adornment* is a set of text that Object Builder inserts into a generated implementation file. Object Builder allows you to add *file adornments* to the beginning (prologue or prefix) or end (epilogue or suffix) of a file. It also lets you add *method adornments* near method definitions. Adornments can include comments, pragma statements, and any compilable code.

Warning: Object Builder does not inspect adornments for validity at any stage. You are responsible for ensuring that whatever is contained within the adornments is valid, compilable code, and that comments have correct comment tagging.

File adornments

With file adornments, you can insert text around multiple class definitions within a file. You can add file multiple adornments to a class or file, and you can specify the adornment as platform-specific. Ordering of the adornments within a generated file will be determined by the ordering of the adornments in the tree view.

Individual file adornments are located under the 'Prefixes' and 'Suffixes' nodes. Each 'Prefix' and 'Suffix' node has a pop-up menu option **Add Adornment**. This action opens the File Adornments wizard, which is similar to the Properties wizard of a method body. In the File Adornments wizard, you provide a name for the adornment you define, and you specify whether to get the adornment text from the Source pane or from an external file.

The File Adornments folder organizes any prolog or epilog content you want added to the generated files for the currently selected object in the Tasks and Objects pane. The File Adornments folder appears when you select a business object implementation, data object implementation, key, copy helper, or persistent object.

You can add adornments that serve as either file prefixes or suffixes, or interface prefixes or suffixes. The text that you specify for the adornment is included in the generated file the next time you generate code for the object.

Method adornments

Method adornments let you add text, such as method descriptions, to your generated code. The text is inserted in direct proximity to the method, depending on which option you select:

- Before the start of the method.
- Just after the opening brace "{" and before the body of the method.
- Just after the body of the method, before the closing brace "}".
- After the method body ends, immediately following the closing brace.

Method adornments assist in writing Javadoc-style comments in your Java code. The resulting generated code can be processed through Javadoc to produce HTML documentation of the classes and methods.

As with file adornments, you can specify for method adornments where the text comes from (a separate file or the Source pane), and for which target platforms the adornment is generated.

RELATED TASKS

“Generating code” on page 551

“Adding file adornments”

“Adding method adornments” on page 242

Adding file adornments

You can add file adornments as either prefixes or suffixes to an object’s file or interface. The adornment can include comments, pragma statements, and so on. You can add file adornments for the following objects:

- business object implementation
- data object implementation
- key
- copy helper
- persistent object

Warning: Object Builder does not inspect adornments for validity at any stage. You are responsible for ensuring that whatever is contained within the adornments is valid, compilable code, and that comments have correct comment tagging.

To add a file adornment, follow these steps:

1. Select the object in the Tasks and Objects pane. The File Adornments folder appears in the Methods pane.
2. From the pop-up menu of either File Prefixes, Interface Prefixes, File Suffixes, or Interface Suffixes in the File Adornments folder, select **Add**. The Add Adornments wizard opens to the Adornment Details page.
3. Type a name for the adornment in the **Name** field.
4. Select the language in which files will be generated when you generate code for the object. You can select either C++, or Java. C++ is selected by default. This gives you the choice of selecting either .cpp, .ih, or .idl files to hold the contents of the adornments. If you select only Java, you have the option of placing the adornment in either the generated .idl file, or the generated .java file.

Note: You are restricted in the choice of a language in this section depending on the implementation language that you selected for the

object. That is, you should select Java as a language on this page only if you have Java as the object's implementation language.

5. Select the generated file into which you want the adornment details to be placed.

If you select C++ as one of the languages in step 4, you have the option of placing these details in either the generated IDL file (select the **IDL** check box), the generated header file (select the **Header** button), or the generated implementation file (select the **Implementation** button).

If you select Java alone as the language in step 4, the only location option available for selection on this page is the IDL file, but that selection is optional: if you do not select the **IDL** check box, the adornment is placed in the generated .java file.

6. In the Text Source section, indicate whether you want to use code for the adornment that is contained in another file (select **Use an external file**), or whether you want to directly type the code for the adornment in the Source pane (select **Use text in Source pane**).

If you indicated that you wanted to use the text in the editor pane, the Source pane will now change from Read-Only to Insert mode, and you can start typing text to be used for the adornment that you are adding. When you are done, you can generate code for the object.

If you want to use an external file, you must provide its name in the **File Name** field, or click the **Browse** button, and use the Find File dialog box to search for the exact location, and select the file. You can generate code for the object straight away. The adornments will be piped directly from the specified source file.

7. Select one or more operating system platforms on which files for the object containing the adornments are to be generated.
8. Click **Finish**.

If you are creating either a file prefix or suffix, the adornment details, complete with the adornment name and the text that you either type or import from an external file will appear in the prolog section of the file that you select in step 5, if you are creating a prefix, and at the end of file if you are creating a suffix.

If you are creating either an interface prefix or suffix, the adornment details, complete with the adornment name and the text that you either type or import from a file will appear in the interface definition class in the file that you select in step 5.

Note the following points:

- If you edit a business object file by adding new files to be included, you must include these files as file adornment prefixes for keys and data object implementations that you define later for the business object, as well as for those that are already defined. These key and data object implementation

IDL files will not automatically include the headers. Include them by selecting **IDL** as the file location on the Adornment Details page.

- Prior to this release of Component Broker, Object Builder file adornments were represented as prologues and epilogues in implementation source files. During migration, these prologues will become file prefixes, and epilogues will become file suffixes in the present release's model.

RELATED CONCEPTS

"File and method adornments" on page 239

RELATED TASKS

"Generating code" on page 551

"Adding method adornments"

Adding method adornments



You can add method adornments to method objects. The adornment can contain any compilable text, but method adornments are most commonly used to add comments. You can add adornments to the methods of the following objects:

- business object implementations
- data object implementations
- keys
- copy helpers
- persistent objects
- local-only objects

You can add adornments to all types of methods that belong to these objects: user-defined methods, user-defined attribute get/set methods, framework methods, and user-defined relationships.

Warning: Object Builder does not inspect adornments for validity at any stage. You are responsible for ensuring that whatever is contained within the adornments is valid, compilable code, and that comments have correct comment tagging.

To add a method adornment:





1. Select the object in the Tasks and Objects pane. All the object's methods appear in the Methods pane.
2. Select the method to which you want add the adornment. From the method's pop-up menu select the appropriate option for adding:
 - **Add Adornment before Method Definition** - Inserts text before the method is declared. This is what you would use for Javadoc-type comments.

- **Add Adornment before Method Body** - Inserts text immediately after the opening brace “{”.
- **Add Adornment before Method End** - Inserts text immediately before the closing brace “}”.
- **Add Adornment after Method End** - Inserts text immediately after the closing brace.

The Add Adornments wizard opens to the Adornment Details page.

3. Enter a name for the adornment in the **Name** field.
4. Select the language in which files will be generated when you generate code for the object. You can select either C++, or Java. C++ is selected by default. You are restricted in the choice of a language in this section depending on the implementation language that you selected for the object. That is, you should select Java as a language on this page only if you have Java as the object’s implementation language.
5. In the Text Source section, indicate whether you want to use code for the adornment that is contained in another file (select **Use an external file**), or whether you want to directly type the code for the adornment in the Source pane (select **Use text in Source pane**). If you choose to use an external file, you must provide its name in the **File Name** field.
6. Select one or more operating system platforms. The adornment will be included in generated code that is targeted for only the platforms you select.
7. Click **Finish**.

When you add an adornment to a method, the tree in the Methods pane shows the adornment beneath the method. The adornments will be listed in the order in which they will appear in the method (for example, those placed before the method definition first, those after the closing brace last). The following symbols indicate what the position of the adornment is, relative to the method:

Symbol	Position
	Adornment placed before the method definition.
	Adornment placed immediately after the opening brace.
	Adornment placed immediately before the closing brace.
	Adornment placed immediately after the closing brace.

You can add more than one adornment in each position. This can be useful, for example, when a method has different parameters of behavior on different

platforms. For example, if a method has three parameters in its Windows implementation, and only two in its AIX implementation, you could create two adornments. The one for Windows would describe all three parameters, and you would select the **Windows** check box in the Platforms section of the Add Adornment wizard. The adornment for AIX would describe the two parameters, and you would select the **AIX** check box in the wizard.

About Javadoc comments

The text of a Javadoc comment starts with `/**` and ends with `*/`. Javadoc adornments should be placed before the method definition. For more information about Javadoc comment tagging and formatting, see Sun's Java website, <http://java.sun.com>.

RELATED CONCEPTS

"File and method adornments" on page 239

RELATED TASKS

"Generating code" on page 551

"Adding file adornments" on page 240

Business Object Behavior

Business object behavior encompasses the pattern to be used to handle the essential state of the business object, how the business object handles object references, whether, (only if the object is a sessional business object) you can provide your own code to be called during some of the normal processing for Session service, whether the object is to be associated with a data object, and the platform on which the business object is to be deployed.

A business object interface can have multiple implementations, depending on the quality of service required.

The following reference topics deal with business object behavior:

- "Pattern for Handling State Data" on page 245
- "Object Reference" on page 246
- "Data Object Interface" on page 247
- "Session Service" on page 248
- "Deployment platforms" on page 423

You can specify all these details when you define an implementation for the business object, on the Name and Data Access Pattern page of the Business Object Implementation wizard. This page has the same sections as the reference topics listed. In addition, besides providing the names for the business object implementation's file, module and interface, you can also

specify the platform on which the object is to be deployed. You can select from one or more of Windows NT, AIX, OS/390, Solaris, and HP-UX.

RELATED CONCEPTS

Business object (*Programming Guide*)

Data object (*Programming Guide*)

Object relationships (*Programming Guide*)

RELATED TASKS

“Adding a business object implementation and data object interface” on page 780

Pattern for Handling State Data

The implementation of the business object must have the specifications as to how the object has access to state data (data that is persistent). A business object can either have a part to play in the maintenance of its state, or it can delegate that responsibility entirely to its associated data object.

You can make this decision using the Name and Data Access Pattern page of the Business Object Implementation wizard when you add a business object implementation (**Add Implementation** from the pop-up menu of the business object interface in the User-Defined Business Objects folder, in the Tasks and Objects pane), or when you edit the object’s implementation that you have defined (**Properties** from the pop-up menu of the business object implementation in the User-Defined Business Objects folder). In the bottom-up case, when you create a business object from a data object, you can indicate the pattern to be used for accessing state data using the Implementation Specifications page of the Add Business Object wizard (**Add Business Object** from the pop-up menu of the data object interface in the User-Defined Data Objects folder).

The Pattern for Handling State Data section on this page has the following options:

- Delegating
- Caching
- Same as parent’s

Delegating

Select this radio button if you want to use the `IManagedObjectWithDataObject` class as the data access pattern. The maintenance of the business object’s state is delegated to the data object. The essential state is passed to the data object, which sends it back to the business object. All non-derived, non-essential state is still stored (cached) in the business object

Note: You must select this option if you are creating objects that are to be used with either procedural application adaptors, or MQSeries application adaptors.

Caching

This is the default pattern. This pattern uses the `IManagedObjectWithCachedDataObject` class for data access. Both the business object and the data object cache a local copy of the essential state. All essential state, and all non-derived, non-essential state, is cached in the business object. The business object does not delegate any getter or setter calls to the data object. Additional framework methods `syncToDataObject()` (which is used to load the business object with data contained in the data object), and `syncFromDataObject()` (which is used to send data from the business object back to the data object) are used to keep the cached copy of the attributes in correspondence with the data object attributes. Once this option is selected, the **Object Reference** section is activated.

Same as parent's

The pattern for handling state data, which is used by the parent of this interface, will be used for this implementation.

Note: This option is selected by default if the interface for this business object inherits from another business object interface. However, you still have to indicate the parent on the Implementation Inheritance page of this wizard, after you delete the default parent for business object implementations, which is `IManagedClient` `IManagedClient::IManageable`.

RELATED CONCEPTS

Business object (*Programming Guide*)

Object relationships (*Programming Guide*)

Cache Service (*Advanced Programming Guide*)

RELATED TASKS

"Adding a business object implementation and data object interface" on page 780

"Adding a business object interface" on page 777

"Creating a specialized home" on page 876

"Creating a container instance" on page 578

Object Reference

When you specify **Caching** as the pattern to be used for handling the essential state (state data) of the business object, you can select the **Use lazy evaluation** check box if you want the first copy of object references in the essential state to be fetched only when it is required, rather than automatically at startup.

You can make this decision using the Name and Data Access Pattern page of the Business Object Implementation wizard when you add a business object implementation (**Add Implementation** from the pop-up menu of the business object interface in the User-Defined Business Objects folder, in the Tasks and Objects pane), or when you edit the object's implementation that you have defined (**Properties** from the pop-up menu of the business object implementation in the User-Defined Business Objects folder). Even in the bottom-up case, when you create a business object from a data object, you can indicate your preference for accessing object references using the Name and Data Access Pattern page of the Add Business Object wizard (**Add Business Object** from the pop-up menu of the data object interface in the User-Defined Data Objects folder).

RELATED CONCEPTS

Business object (*Programming Guide*)

Object relationships (*Programming Guide*)

Cache Service (*Advanced Programming Guide*)

RELATED TASKS

"Adding a business object implementation and data object interface" on page 780

"Adding a business object interface" on page 777

"Creating a specialized home" on page 876

"Creating a container instance" on page 578

Data Object Interface

Note: This section is not available when you are adding a business object from a data object.

You can choose to have Object Builder create a data object interface along with the business object implementation you are defining, or you can create or select a data object interface later.

You can make this decision using the Name and Data Access Pattern page of the Business Object Implementation wizard when you add a business object implementation (**Add Implementation** from the pop-up menu of the business object interface in the User-Defined Business Objects folder, in the Tasks and Objects pane), or when you edit the object's implementation that you have defined (**Properties** from the pop-up menu of the business object implementation in the User-Defined Business Objects folder).

Note: this section is not available when you are adding a business object from a data object.

You can select one of the following radio buttons:

- Create a new one now
- Add or select one later

Create a new one now

Select this option if you want the data object to be derived from the business object. The data object is automatically created when you add a business object implementation. This is the default option.

Add or select one later

Select this option when you want to reuse an existing data object, which is stand-alone and not derived from a business object. This option enables you to match the interface and function requirements of the newly created top-down model with the classes developed from existing data.

RELATED CONCEPTS


Business object (*Programming Guide*)

RELATED TASKS

“Adding a business object implementation and data object interface” on page 780

“Adding a business object interface” on page 777

Session Service

 This section is not applicable, and therefore not available when the deployment platform is OS/390.

Use this section if you plan to make the business object sessional. When a business object uses Session Service, you can provide your own code to be called during some of the normal processing for those services. To do this, select the **Provides resource support** check box in this section.

Provides resource support

Select this check box to indicate that the business object implementation inherits from ISessions::Resource, the class that has the endResource(), the checkpointResource(), and the resetResource() methods in it. Object Builder creates these methods on the business object, and you can provide your own code for each of them. Your code will be called just before the corresponding method is called on the managed object's mixin.

RELATED CONCEPTS

Business object (*Programming Guide*)

Session Service (*Advanced Programming Guide*)

RELATED TASKS

“Adding a business object implementation and data object interface” on page 780

“Adding a business object interface” on page 777

“Creating a specialized home” on page 876

“Creating a container instance” on page 578

“Adding resource methods to a sessional business object” on page 164

Data Object Behavior

The behavior of a data object depends on various factors such as the environment for the business object, the implementation type of the data object and its storage options, and the pattern used by the data object for data access and storage of references.

The following reference topics deal with different aspects of the behavior of data objects:

- “Environment”
- Persistent Behavior and Implementation
- “Data Access Pattern” on page 254
- “Handle for Storing Pointers” on page 255
- “Deployment platforms” on page 423

You can specify all these details when you define an implementation for the data object on the Behavior page of the Data Object Implementation wizard. This page has the same sections as the reference topics listed.

RELATED CONCEPTS

Data object (*Programming Guide*)

Persistent object (*Programming Guide*)

Data object customization for cardinality relations (*Programming Guide*)

RELATED TASKS

“Working with data objects” on page 795

Environment

The environment for a component has to be either conducive to production or deployment. The environments that use the Business Object Application Adaptor (previously known as BOIM) are for production, when we implement a real application with persistent data.

You can select the environment for your component in the Behavior page of the Data Object Implementation wizard either when you add a data object

implementation (**Add Implementation** from the pop-up menu of the data object interface in the User-Defined Business Objects folder, in the Tasks and Objects pane), or when you edit the data object implementation you have defined (**Properties** from the pop-up menu of the data object implementation in either the User-Defined Business Objects folder, or the User-Defined Data Objects folder).

You have the following environment options:

- BOIM with UUID key
- BOIM with any key
- Same as parent's

BOIM with UUID key

This environment uses the Business Object Instance Manager (BOIM), commonly known as Business Object Application Adaptor, with the Universally Unique Identity (UUID) key. Select this implementation to create unique server data objects with transient data for supporting the business object. This option is useful for short-lived business objects that do not have to persist after your application has finished executing. The type of persistence is automatically set to **Transient**, and cannot be changed. A local copy of the essential state is used for data access.

If the data object's environment is **BOIM with UUID key** (page 250), the copy helper should inherit from `IManagedAdvancedServer::IUUIDCopyHelperBase`. The copy helper will only be usable by other components on the server: client applications should not create UUID components on the server.

BOIM with any key

Select this implementation to create server data objects with persistent data for supporting the business object. The data object will be installed in a business object application adaptor, and instances of the object will be located using keys. This is the option to select if you want to use a relational backend datastore, as shown in the Life Insurance example, or a procedural (PAO) backend (one that is supported by procedural application adaptors).

Important: Whenever you select this environment, you must associate a primary key with your managed object assembly.

This environment enables you to create a persistent object or use an existing one. If you use this option and create any persistent objects for this data object, you must use a customized container instance. The default container instances are only appropriate for objects with transient data.

When you select this option, all the options in the **Type of Persistence** section are available for selection. The default form for persistent behavior is set to **Embedded SQL**.

Same as parent's

Select **Same as parent's** when you want to use the implementation type that is specified for the parent of this interface. The datastore defined in the parent is used. If the parent has no persistent object, this newly created data object has no persistent back-end. However, if the parent uses **Embedded SQL**, the newly created data object inherits that behavior. A local copy of the essential state is used for data access.

If you are defining an implementation that inherits from another, this option will be selected.

RELATED CONCEPTS

Data object (*Programming Guide*)

Persistent object (*Programming Guide*)

Application adaptor (*Programming Guide*)

Transient data object customization - UUID key (production use) (*Programming Guide*)

Transient data object - any key (production use) (*Programming Guide*)

Data object customization and inheritance (*Programming Guide*)

"Container" on page 578

State data (*Programming Guide*)

Cache Service (*Advanced Programming Guide*)

Using sets of objects (Using reference collections) (*Programming Guide*)

RELATED TASKS

"Working with data objects" on page 795

"Adding a persistent object and schema" on page 833

"Customizing referential integrity" on page 714

"Creating a container instance" on page 578

"Configuring a managed object" on page 588

RELATED REFERENCES

Type of Persistence

"Data Object Implementation Inheritance" on page 257

Type of Persistence

The data object implementations you define differ from one another based on whether the associated data object is persistent or not, and on the type of service they use.

You can select the type of persistence in the Behavior page of the Data Object Implementation wizard either when you add a data object implementation (**Add Implementation** from the pop-up menu of the data object interface in the User-Defined Business Objects folder, in the Tasks and Objects pane), or when you edit the data object implementation you have defined (**Properties** from the pop-up menu of the data object implementation in either the User-Defined Business Objects folder, or the User-Defined Data Objects folder). Each type of persistence has a unique impact in terms of application performance, allocation of resources, and so on.

To be able to select any one of the different types of implementations, you must first select, on the same page of the wizard, the **BOIM with any key** environment. See “Environment” on page 249.

You have a choice of the following implementations:

- Transient
- Embedded SQL
- Cache Service
- Procedural Adaptors
- MQSeries Adaptor

Transient

This option is automatically selected and is the only one available when the data object implementation uses **BOIM with UUID key**. If you selected **BOIM with any key** for the environment, you can select this option if you want a transient managed object with a user-defined identity.

Embedded SQL

Select this option if embedded (static) SQL is to be used by the persistent object to access a DB2 database.

Cache Service

Use the **Cache Service** option if you want the CB server to hold cached copies (instances) of the data object in memory, accessing the corresponding rows in the relational database only when necessary. This results in improved performance when the values in the accessed row need to be read frequently but not updated as frequently. Select this option if you plan to use either a Oracle, or Informix backend (it is also optional for DB2).

 **390** **Cache Service** is not available when the target platform is OS/390.

Restriction:A given transaction cannot access more than one Informix database per CB server. To involve two Informix databases in a transaction, you must access each database from a different server.

Procedural Adaptors

Select this option if the data object implementation is to be connected to a persistent object that is created for an imported procedural adaptor (PA) bean.

MQSeries Adaptor

Select this option if you want to enable MQSeries applications to participate in distributed transactions. The MQSeries application adaptor makes use of the MQSeries' XAinterface for this purpose. If you select this option, no persistent object will be required to be associated with this data object implementation.

WIN HP-UX SOLARIS The MQSeries application adaptor can be used only if the platform for the object is Windows NT, HP-UX, Solaris, or a combination of these.

Note the following points when you configure a managed object for your application:

- You can select only those containers that match the data object implementation (and the managed object). For example, if your data object implementation uses **Embedded SQL**, only those containers that use embedded SQL (those without caching services) are shown. Similarly, if you defined the data object implementation to use **Procedural Adaptors**, and the related persistent object uses **Session Service**, the selection you made on the Names and Connectors page of the Import Procedural Adaptor Bean wizard, only containers that are configured for sessions are shown.
- You will be able to select only those data object implementations in the model (on the Data Object Implementations page of the Configure Managed Object wizard) that use the service you specified on the Names and Connectors page of the Import Procedural Adaptor Bean wizard.

RELATED CONCEPTS

Data object (*Programming Guide*)

Persistent object (*Programming Guide*)

Application adaptor (*Programming Guide*)

Data object implementation (*Programming Guide*)

"Container" on page 578 (*Programming Guide*)

State data (*Programming Guide*)

Cache Service (*Advanced Programming Guide*)

Using sets of objects (Using reference collections) (*Programming Guide*)

RELATED TASKS

"Working with data objects" on page 795

"Adding a persistent object and schema" on page 833

"Customizing referential integrity" on page 714

"Creating a container instance" on page 578

"Configuring a managed object" on page 588

RELATED REFERENCES

“Data Object Implementation Inheritance” on page 257

“DB2 data type mappings” on page 142

“Oracle data type mappings” on page 146

“Informix data type mappings” on page 148

Data Access Pattern

The data access pattern determines how the data object accesses data with the help of its persistent object.

You can set the data access pattern in the Behavior page of the Data Object Implementation wizard either when you add a data object implementation (**Add Implementation** from the pop-up menu of the data object interface in the User-Defined Business Objects folder, in the Tasks and Objects pane), or when you edit the data object implementation you have defined (**Properties** from the pop-up menu of the data object implementation in either the User-Defined Business Objects folder, or the User-Defined Data Objects folder).

Data access patterns for some data object implementations are predestined by Object Builder, depending on the type of service used, or the transient or persistent nature of the implementation.

If the type of persistence of the data object is **Embedded SQL**, you can select one of the following access patterns:

- Delegating
- Local copy

Delegating

The data object uses the attributes of its associated persistent object. The get and set methods of the data object call the corresponding get and set methods of the persistent object, which, in turn, access the persistent object attributes. A local copy of the data object attributes is maintained in the private members of the data object class. If the implementation uses either of the **Cache Service** options, or **Procedural Adaptors**, the data access pattern is automatically set to **Delegating**. This setting cannot be changed.

If **Delegating** is used, the essential state is passed from the persistent object to the data object, which sends it back to the business object. If you select **Local copy**, the data object caches a local copy of the essential state.

Local copy

The data object has its own local copy of its attributes. They are private members of the data object class that can be accessed directly by the data object’s methods. That is, the get and set methods are applied to private

copies of the attributes. The attributes of the persistent object are set only when the business object application adaptor makes a call that invokes the database during managed object activation, or transaction commit. The local copy is used for type conversion purposes as in the case when a mapping helper is used to map the attributes of the data object to the attributes of an associated persistent object. If the implementation is **Transient**, the data access pattern is automatically set to **Local copy**, and cannot be changed.

RELATED CONCEPTS

Data object (*Programming Guide*)

Persistent object (*Programming Guide*)

Application adaptor (*Programming Guide*)

Data object implementation (*Programming Guide*)

“Container” on page 578

State data (*Programming Guide*)

Cache Service (*Advanced Programming Guide*)

Using sets of objects (Using reference collections) (*Programming Guide*)

RELATED TASKS

“Working with data objects” on page 795

“Adding a persistent object and schema” on page 833

“Customizing referential integrity” on page 714

“Creating a container instance” on page 578

“Configuring a managed object” on page 588

RELATED REFERENCES

“Data Object Implementation Inheritance” on page 257

Handle for Storing Pointers

The design pattern you use for the data object implementation determines how object references are stored (made persistent rather than transient) for later retrieval and use. Object references are in a format that can be stored in a database. These persistent storage forms, when converted back to in-memory pointers, need no further transformation in order to point to the right object. These patterns are implemented using handles that Object Builder generates. The choice of a pattern is based on factors such as speed of execution and storage overhead.

You can select the handle to be used for storing pointers in the Behavior page of the Data Object Implementation wizard either when you add a data object implementation (**Add Implementation** from the pop-up menu of the data object interface in the User-Defined Business Objects folder, in the Tasks and Objects pane), or when you edit the data object implementation you have

defined (**Properties** from the pop-up menu of the data object implementation in either the User-Defined Business Objects folder, or the User-Defined Data Objects folder).

Note:Your selection of a handle does not affect the default mapping of object references on the Attributes Mapping page of this wizard. It is relevant only if you override the default setting of the persistent object and schema's attribute mapping from **Key Home** to **Primitive**.

You can select one of the following handles:

- Default
- Stringified object reference
- Object name
- Home name and key

Default

Select this option if you prefer to have the default handle used as the design pattern for swizzling pointers. The default handle is the one you select when you define the implementation for the corresponding business object on the Handle Selection page of the Business Object Implementation wizard is used.

Stringified object reference (SOR)

Select this option to distribute the object reference in the CORBA environment. This is the string form of an object reference. It helps in externalizing an object to a stream.

Object name

Select this option only for objects named using the Naming Service. An object thus named provides an interface that returns its name.

Home name and key

Select this option to implement specific relationships among CB Server objects. The handle that the data object uses to store references to other objects is composed of a home that stores the instance of the referenced object and a key that identifies the instance. This option is sometimes preferred over a stringified object reference (SOR) because it takes less storage space, and can be maintained more efficiently: transferring a home from one server to another will not break a home name and key reference, but it would an SOR.

RELATED CONCEPTS

Data object (*Programming Guide*)

Persistent object (*Programming Guide*)

Using handles (*Programming Guide*)

Naming Service (*Advanced Programming Guide*)

Application adaptor (*Programming Guide*)

Data object customization for cardinality relationships (*Programming Guide*)
 Top-down customizations (*Programming Guide*)
 Object relationships (*Programming Guide*)
 Using sets of objects (Using reference collections) (*Programming Guide*)

RELATED TASKS

“Working with data objects” on page 795

RELATED REFERENCES

“Data Object Implementation Inheritance”

Data Object Implementation Inheritance

▶CHANGED

Persistence	Default Parent Implementation	Platforms
Transient	IBOIMExtLocalToServer IBOIMExtLocalToServer::IDataObjectBase	All but OS/390
Transient	IBOIM390ExtLocalToServer IBOIM390ExtLocalToServer::IDataObject	OS/390
DB2 Cache Service	IRDBIMExtLocalToServer IRDBIMExtLocalToServer::ICachingServiceDataObject	All
Oracle Cache Service	IRDBIMExtLocalToServer IRDBIMExtLocalToServer::ICachingServiceDataObject	All
Informix Cache Service	IRDBIMExtLocalToServer IRDBIMExtLocalToServer::ICachingServiceDataObject	All
Procedural Adaptors	IPAAExtLocalToServer IPAAExtLocalToServer::IDataObject	All
MQSeries Adaptor	IMQAAExtLocalToServer IMQAAExtLocalToServer::IDataObject IRDBIMExtLocalToServer::IDataObject	NT
Embedded SQL	RDBIMExtLocalToServer IRDBIMExtLocalToServer::IDataObject	All
Transient(BOIM with UUID Key)	IBOIMExtLocalToServer IBOIMExtLocalToServer::IUUIDDataObject	All

RELATED CONCEPTS

Data object (*Programming Guide*)
 Persistent object (*Programming Guide*)

RELATED TASKS

“Working with data objects” on page 795

“Adding a data object implementation” on page 807

“Editing a data object implementation” on page 819

Objects to source files mapping

The following table summarizes the types of files produced for each of the component objects you can define in Object Builder. The objects appear in Object Builder’s Tasks and Objects pane.

Object	Source Files
Business object file	<i>filename.idl</i>
Business object module	<i>filename.idl</i> (Module and interface artifacts are contained in the IDL source file (<i>filename.idl</i>) of the business object file that contains them.)
Business object interface	
Key	<i>filenameKey.idl</i> <i>filenameKey.ih</i> <i>filenameKey_I.cpp</i> <i>classnameHelper.java</i> <i>_classnameImpl.java</i>
Copy helper	<i>filenameCopy.idl</i> <i>filenameCopy.ih</i> <i>filenameCopy_I.cpp</i> <i>classnameHelper.java</i> <i>_classnameImpl.java</i>
Business object implementation	<i>filename.idl</i> <i>filename.ih</i> <i>filename_I.cpp</i> (if it is a C++ business object) <i>filenameBase.java</i> (if it is a Java business object) <i>classnameKeyAssistant.idl</i> <i>classnameKeyAssistant.ih</i> <i>classnameKeyAssistant_I.cpp</i>
Data object file	<i>filename.idl</i>
Data object module	<i>filename.idl</i> (Module and interface artifacts are contained in the IDL source file (<i>filename.idl</i>) of the data object file that contains them.)
Data object interface	
Data object implementation	<i>filename.idl</i> <i>filename.ih</i> <i>filename.cpp</i>
DB schema	<i>filename.sql</i>

Object	Source Files
DB persistent object	<i>myPO.hpp</i> <i>myPO.sqx</i> (Embedded SQL) <i>myPO.sqx</i> (Cache Service)
PA persistent object	<i>myPO.idl</i> <i>myPO.hpp</i> <i>myPO.cpp</i> <i>myPOIFImpl.java</i>
Managed object	<i>filename.idl</i> <i>filename.ih</i> <i>filename.cpp</i>
Local-only object	<i>filename.idl</i> <i>filename.ih</i> <i>filename_I.cpp</i> <i>classnameHelper.java</i> <i>_classnameImpl.java</i>
Build Configuration folder	<i>projdefs.mk</i> <i>local.mak</i> <i>all.mak</i> <i>QT.bat</i> <i>QT.txt</i>
DLL build targets	<i>BuildTargetName.mak</i>
Application families	<i>AppFam.ddl</i> <i>AppFam.auto.ddl</i>
Containers	Not applicable
Enterprise beans	Not applicable

Note the following points:

- For a business object implementation, or a local-only object, either Java source or C++ source is generated, based on the implementation language selected in the object’s wizard.
- *classname* is the name of the key or copy helper class that you specify when you define the object.

RELATED CONCEPTS

Component (*Programming Guide*)

Naming conventions (*Programming Guide*)

RELATED TASKS

“Generating code” on page 551

RELATED REFERENCES

“Internationalization of data” on page 132

“Naming objects” on page 128

Chapter 5. Components working together

Creating a composite component

A composite component provides access to the methods and data of its member components. The member components provide their own persistence for the data the composite accesses. The composite can also define its own original methods and data, and provide persistence for its key attributes and original data.

To create a composite component, you need to:

1. Group components into a composition.
2. Create a composite business object based on the composition.
3. Create a composite key for the component.
4. Complete the rest of the component.

The steps for creating a composite component are as follows:

1. "Creating a composition file" on page 885
2. "Adding a composition module" on page 886
3. "Adding a composition" on page 886
4. "Creating a business object file" on page 775
5. "Adding a business object module" on page 777
6. "Adding a composite business object interface" on page 892
7. "Adding a composite key" on page 901
8. "Adding a copy helper" on page 830
9. "Adding a composite business object implementation and data object interface" on page 894
10. "Adding a data object implementation" on page 807
11. "Adding a managed object" on page 871

For a sample-based tutorial, see "Tutorial: Composite component creation" on page 267.

RELATED CONCEPTS

"Composite component" on page 262

RELATED TASKS

"Chapter 4. Creating a component" on page 127
"Working with compositions" on page 884

“Working with composite business objects” on page 891

“Working with composite keys” on page 900

RELATED REFERENCES

“Naming objects” on page 128

“Internationalization of data” on page 132

Composite component

A composite component is an access point to the data and behavior of one or more other components, which the composite component’s implementation delegates to. Typically, the other components are not directly accessible (in other words, the client cannot use the composite component to get a reference to one of the combined component instances); only specific data and behavior of the other components are available through the composite component’s delegation of attribute and method calls. The composite component may have its own data and methods as well.

A composite component can be of two kinds, based on the way its references to its constituent components are combined:

- **Conjunction composite**
All of the composite component’s references exist at once. In other words, at run-time the composite component has references to instances of each of its constituent components. All of the instances exist at the same time, and the composite combines their interfaces to provide a single access point to their data and behavior. This is the most common type of composite component.
- **Disjunction composite**
Only one of the composite component’s references exists at run-time. In other words, at run-time the composite component has a reference to an instance of only one of its constituent components. The composite component acts as a common interface for two or more mutually exclusive kinds of component, the choice of which is made when the composite component is created.

When you create a composite component, you start by defining the constituent components (in any of the standard ways, for example as a component for new DB data, legacy DB data, or PA data). Then you define the way in which the constituent components are combined in a composition object, and finally you create a new composite component based on that composition.

A composite component consists of the same objects as a normal component, with some differences to provide the compositing behavior:

- A (composite) business object, which is based on the composition.
- A (composite) key for the business object

- A data object, that stores the key attributes for the component, along with any attributes that are unique to this component (not derived from the constituent components).
- DB persistent object and DB schema (optional), that store the values of the key attributes, and the value of any attributes that are unique to the component.
- A copy helper and managed object.

RELATED CONCEPTS

“Composition”

“Composite business object” on page 264

“Composite key” on page 265

Component (*Programming Guide*)

RELATED TASKS

“Tutorial: Composite component creation” on page 267

“Creating a composite component” on page 261

Composition

A composition defines a combined interface for a group of components. In addition, it describes the implementation of the attributes and methods in the combined interface, which delegate to attributes and methods of the components in the group. For example, we might define a composition, `CompositeAccount`, that combines two components, `SavingsAccount` and `CheckingAccount`. The `CompositeAccount` interface might include an attribute `balance` that is defined as the sum of a `balance` attribute on the `SavingsAccount` component and a `balance` attribute on the `CheckingAccount` component.

Once you have defined the composition, you can create composite business objects that are based on the composition.

A composition does not have its own managed object; it is only accessible as part of the business logic of a composite component based on the composition. It is an abstraction of the combining and delegating logic needed to access the data and behavior of the components being combined. This logic is implemented in a local-only helper object, for use by the composite business object that is based on the composition.

When you package a composite component, be sure to include the source files for the composition class in the component’s server DLL or shared library file. Otherwise, the composition logic contained in the helper object will not be available to the composite component.

You can create compositions under the User-Defined Compositions folder, in Object Builder's Tasks and Objects pane. For each component that you add to a composition, the composition has:

- A managed object instance, of the same type as the component's managed object
- Attributes that delegate to the component attributes.
- Methods that delegate to the component methods.

You can edit which attributes and methods are included, and what they delegate to. You can also define attributes and methods that contain logic or data that is unique to the composition, and does not simply delegate to a combined component. This is useful for adding private helper functions to hold user-defined logic. For example, a composition `AllAccounts`, which combines the components `CheckingAccount` and `SavingsAccount`, could have a private helper method `addFloats`, which can take the two original balances (`CheckingAccount1.balance` and `SavingsAccount1.balance`) as arguments, and return their sum. You can then map `AllAccounts.balance` to the helper method. When you add a new method, you can supply its implementation (for example, `return arg1+arg2`) in Object Builder's Source pane (after you complete the composition, click on it in the Tasks and Objects pane; then select the method in the Methods pane, and complete its implementation in the Source pane).

RELATED CONCEPTS

"Composite component" on page 262

"Composite business object"

RELATED TASKS

"Tutorial: Composite component creation" on page 267

"Creating a composite component" on page 261

"Working with compositions" on page 884

Composite business object

A composite business object is part of a composite component. The business object is based on a composition, which defines the interface to one or more combined components.

When you base a business object on a composition, the business object automatically gets the attributes and methods defined in the composition (except for the composition's references to its constituent components). The business object attributes and methods have implementations that delegate to their equivalents in the composition helper object. As with any other business object, you can also define other attributes and methods that are unique to the composite component, and do not delegate to a composition. You can make these attributes persistent through a DB schema.

The composition has a component instance for each component it composites. It does not, however, deal with managed object configuration issues such as how and when to find or create these instances. This information is instead provided in the composite business object. This allows you to re-use the pure combining logic of the composition in multiple versions of a composite component, each version providing different managed object configuration information. You provide the information for finding and creating the managed object instances in the composite business object implementation. The instances are then used by the business object, in conjunction with the logic in the composition helper, to delegate its attribute and method calls appropriately.

Each composite business object must have a composite key, in which the key attributes of the composite business object can be mapped to key attributes of the combined components. If the attributes have a simple mapping, you can define the mapping in the Key wizard and have the appropriate logic generated by Object Builder. If you require a more complex mapping, you can edit the provided mapping methods (for example, `get_SavingsAccount_accountNo`) and provide your own implementations.

You can use the composite component's data object to store a secondary source for an attribute. If a delegating call to an attribute fails (for example, because the combined component that provides it is unavailable), the composite component will return the value in the data object instead of fail. This is particularly useful for composite components that use the disjunction pattern. In the disjunction pattern, only one of the combined component instances is available at run-time, which means that any unique attributes of the other combined components are unavailable. The data object can provide a secondary source for these unique attributes, which is used when the current component instance does not provide them.

RELATED CONCEPTS

"Composite component" on page 262

"Composition" on page 263

"Composite key"

Business object (*Programming Guide*)

RELATED TASKS

"Tutorial: Composite component creation" on page 267

"Creating a composite component" on page 261

"Working with composite business objects" on page 891

Composite key

A composite key is the key object for a composite component. As with a regular key, the composite key defines attributes of its component that are to

be used to find a particular instance of the component on the server. The key consists of one or more of the business object attributes, which must contain enough information to uniquely identify an instance. For a composite key, these business object attributes may optionally be used to identify the components that make up the composition.

A common pattern for locating the contributing components of a composition is to make the identity of the composite component the union of the identities of the contributing components. In other words, the composite key attributes are equivalent to the various key attributes of the components in the composition.

For example:

- A composite component AllAccounts is based on the composition AccountComposition, that combines two other components, SavingsAccount and CheckingAccount.
- The key attribute for SavingsAccount is accountNo.
- The key attribute for CheckingAccount is accountNo.
- The key attributes for AllAccounts are savingsAccountNo and checkingAccountNo, each of which is mapped to its equivalent accountNo attribute in SavingsAccount and CheckingAccount.

The composite key contains enough information to uniquely identify the AllAccounts component, and also to locate the equivalent SavingsAccount and CheckingAccount components. There is no need to maintain persistent references from the composite component to its constituent components; if you can find AllAccounts, you have enough information to find SavingsAccount and CheckingAccount.

When you use this pattern (the identity of the composite component as the union of the identities of its constituent components), you can provide a mapping between the attributes of the composite key and the attributes of keys for the combined components. You can define simple mappings between the two sets of attributes in the composite key's Key wizard.

For example, given the following objects and key attributes:

- AllAccountsKey is the composite key for AllAccounts, and has two key attributes:
 - savingsAccountNo
 - checkingAccountNo
- AccountComposition is the composition on which AllAccounts is based, and combines two components:
 - SavingsAccount, with the key attribute accountNo, defined in the key object SavingsAccountKey

- CheckingAccount, with the key attribute accountNo, defined in the key object CheckingAccountKey

The attributes in the composite key AllAccountsKey would be mapped as follows:

- savingsAccountNo maps to accountNo in SavingsAccountKey
- checkingAccountNo maps to accountNo in CheckingAccountKey

For simple mappings such as this one (where the attributes are of the same type, and the mapping is one-to-one), the mapping information will be used to generate implementations of the `get_` methods (for example, `get_SavingsAccount1_accountNo`) in the composite business object implementation. If a mapping is complex or not provided at all, then you need to provide your own implementation for these methods.

RELATED CONCEPTS

“Composite component” on page 262

“Composition” on page 263

“Composite business object” on page 264
Key (*Programming Guide*)

RELATED TASKS

“Tutorial: Composite component creation”

“Creating a composite component” on page 261

“Working with composite keys” on page 900

“Editing get and set methods” on page 756

Tutorial: Composite component creation

This tutorial provides instructions for creating a composite component, that consolidates the interfaces of two other components.

For a complete list of prerequisites that you must install on your system, see the *Getting Started with Component Broker* documentation for the appropriate platform.

After you complete this tutorial, you will have two ordinary components, SavingsAccount and CheckingAccount, and a composite component, AllAccounts, that provides access to the data in SavingsAccount and CheckingAccount through a single combined interface.

The following tasks do not give explicit instructions for every step, but should at least get you into the right wizards. If you are experiencing problems, click the Help button within a wizard, or go to the Help pulldown in Object Builder.

Creating the project

Create a sample project to hold your work.

1. Start Object Builder.
2. In the Open Project wizard, type a name and path for the project directory (for example, e:\scenarios\composite).
3. Click **Finish**.
4. When asked whether you want to create a new project, click **Yes**.

Creating the SavingsAccount component

Create a simple component representing a savings account at a bank. For the sake of simplicity, you will be accepting the default for most of the object settings, and using transient data (no persistent objects or schemas).

Define the SavingsAccount interface:

1. From the User-Defined Business Objects folder's pop-up menu, click **Add File** to open the Business Object File wizard.
2. Name the file SAFile.
3. Click **Finish**. The file now appears under the folder.
4. From the file's pop-up menu, click **Add Module** to open the Business Object Module wizard.
5. Name the module SAModule.
6. Click **Finish**. The module now appears under the file.
7. From the module's pop-up menu, click **Add Interface** to open the Business Object Interface wizard.
8. Name the interface SavingsAccount.
9. Click the page title and turn to the Attributes page.
10. Add the following attributes:
 - readonly long accountNo
 - readonly float balance
11. Click **Next** to turn to the Methods page.
12. Add the following methods:
 - void credit (in float amount)
 - void debit (in float amount)
13. Click **Finish**. The interface now appears under the module.

Add a key:

1. From the interface's pop-up menu, click **Add Key** to open the Key wizard.
2. Select accountNo as the key attribute.
3. Click **Finish**. The key now appears under the interface.

Add a copy helper:

1. From the interface's pop-up menu, click **Add Copy Helper** to open the Copy Helper wizard.
2. Select all the attributes to be part of the copy helper.
3. Click **Finish**. The copy helper now appears under the interface.

Add a business object implementation and data object interface:

1. From the interface's pop-up menu, click **Add Implementation** to open the Business Object Implementation wizard.
2. Click the page title and turn to the Key and Copy Helper page.
3. Select SavingsAccountKey and SavingsAccountCopy.
4. Click the page title and turn to the Data Object Interface page.
5. Select all attributes as state data (to be preserved in the data object).
6. Click **Finish**. The business object implementation appears under the business object interface, and the data object interface appears under the implementation.

Add a data object implementation:

1. From the data object interface's pop-up menu, click **Add Implementation** to open the Data Object Implementation wizard.
2. For the sake of simplicity, set the environment to **BOIM with any key** and the form of persistence to **Transient**. This saves you the step of defining the database or procedural adaptor that would normally provide persistence for the data.
3. Click the page title and turn to the Key and Copy Helper page.
4. Select SavingsAccountKey and SavingsAccountCopy.
5. Click **Finish**. The data object implementation appears under the data object interface.

Add a managed object:

1. From the business object implementation's pop-up menu, click **Add Managed Object** to open the Managed Object wizard.
2. Click **Finish**. The managed object now appears under the business object implementation.

Creating the CheckingAccount component

Create another simple component, in the same way, that represents a Checking account. The steps are substantially the same as for the previous task, so only the differences are noted here.

1. Add the CAFile file and CAModule module.
2. Add the CheckingAccount interface, with the following attributes and methods:
 - readonly long accountNo
 - readonly float balance

- long checkCount
 - void credit (in float amount)
 - void debit (in float amount)
3. Add a key, with accountNo as the key attribute.
 4. Add a copy helper, with all attributes selected.
 5. Add a business object implementation and data object interface, with all attributes represented in the data object interface, CheckingAccountKey selected as the key, and CheckingAccountCopy selected as the copy helper.
 6. Add a data object implementation with the **BOIM with any key** setting, transient data, and CheckingAccountKey and CheckingAccountCopy selected as the key and copy helper.
 7. Add a managed object.

Creating the composition

The composition defines the combined interface and delegating implementation for the composite component.

Add a file:

1. From the User-Defined Compositions folder's pop-up menu, click **Add File** to open the Composition File wizard.
2. Name the file ACFile.
3. Click **Finish**. The file appears under the folder.

Add a module:

1. From the file's pop-up menu, click **Add Module** to open the Composition Module wizard.
2. Name the module ACModule.
3. Click **Finish**. The module appears under the file.

Add the composition:

1. From the module's pop-up menu, click **Add Composition** to open the Composition Editor.
2. Click **Add** to display the Composition Palette.
3. Select SavingsAccountMO and CheckingAccountMO.
4. Click **Add** to add them to the **Objects to Composite** list.
5. Click **Close** to close the palette.
6. In the **Objects to Composite** list, you can see entries for both SavingsAccount1 and CheckingAccount1 (the default names for the SavingsAccountMO and CheckingAccountMO instances the composition will hold). Under each instance entry you can see its attributes and methods.

Above the list, you can see the **Composition Style** that is being applied to the selected objects to produce the resulting composition in the **Results** list.

7. Try selecting some other composition styles, and review the results.
8. Return to the **Conjunction without name matching** style.
9. In this style, attributes with conflicting names (such as `accountNo` and `balance`) are made unique by combining them with their instance names (for example, `SavingsAccount1_accountNo`). Attributes that are already unique (such as `checkCount`) are not renamed.
10. Click on the `checkCount` attribute to see the the delegating implementation of its getter method in the Current Republished Value pane.
11. Click **Setter** to see its setter method's implementation.
12. You can use the pop-up menu of the current value to remap the method to another value. For this exercise, simply accept the defaults.
13. Double-click on the `checkCount` attribute to see its properties. You can change the name of the attribute, but you cannot change its implementation details.
14. In the **Results** pane, click on the parent folder (named **Untitled** by default). This folder represents the composition itself.
15. Click the **Properties** tab to display the properties of the composition.
16. Name the composition `AccountComposition`.
17. Click **OK**. The composition appears under the module.

The composition is a complete implementation object. You can generate its IDL and C++ code by selecting **Generate > All** from its pop-up menu.

Click on the composition in the Tasks and Objects pane to review its attributes and methods in the Methods pane. Note that the managed object instances appear as attributes under the User-Defined Attributes folder.

Adding the composite component AllAccounts

Now that you have the composition, you can create a composite component based on the composition. This is similar to the procedure for creating a normal component, and only the differences are noted here.

Add an AAFile file and AAModule module, and then add the AllAccounts interface:

1. From the modules' pop-up menu, click **Add Interface** to open the Business Object Interface wizard.
2. Name the interface `AllAccounts`.
3. Check the **Composite** choice.
4. From the **Composition to Use** list, select `AccountComposition`.

5. Click the page title and turn to the Attributes page.
6. Review the list of attributes the component has received from its base.
7. You can delete or rename these attributes, and create new ones that are specific to the component (rather than taken from the composition). For this exercise, accept the default.
8. Click the page title and turn to the Methods page.
9. Review the list of methods the component has received from its base.
10. You can delete or rename these methods, and create new ones that are specific to the component (rather than taken from the composition). For this exercise, accept the default.
11. Click **Finish**. The interface appears under the module.

Add a composite key:

1. From the interface's pop-up, click **Add Key** to open the Key wizard.
2. Select SavingsAccount1_accountNo and CheckingAccount1_accountNo as key attributes.
3. Click **Next** to turn to the Composite Key page. This page is added to the wizard for keys of composite components.
4. On this page, you map the selected attributes of the composite component back to the original attributes in the keys of the grouped components.
5. In the Composite Key list (on the bottom), click SavingsAccount1_accountNo to select it.
6. In the Composite Key Element list (on the top), expand SavingsAccountKey and click its accountNo attribute.
7. Click **Add**. The mapping is added to the Composite Key list, under SavingsAccount1_accountNo.
8. Perform the same mapping for CheckingAccount1_accountNo to CheckingAccountKey::accountNo.
9. Click **Finish**. The key appears under the interface.

Add a copy helper:

1. From the interface's pop-up menu, click **Add Copy Helper** to open the Copy Helper wizard.
2. Select all attributes to be part of the copy helper.
3. Click **Finish**. The copy helper appears under the interface.

Add a business object implementation and data object interface:

1. From the interface's pop-up menu, click **Add Implementation** to open the Business Object Implementation wizard.
2. Select **Caching** as the pattern for handling state data.
3. Click the page title and turn to the Key and Copy Helper page.
4. Select AllAccountsKey and AllAccountsCopy.

5. Click the page title and turn to the Location page. This page is added to the wizard for business object implementations of composite components.
6. On this page, you define the component's relationship to the composited managed object instances (SavingsAccountMO and CheckingAccountMO), and provide information about the managed objects' locations.
7. Accept the default settings for both components (**Remove the instance when the composition is destroyed** is not checked, **Add the instance to the composition by: Find or create** is selected, and **Create the instance using its copy helper** is not checked).
8. The component will **not** destroy the composition's instances when the composition is destroyed. Any attempt to access a managed object instance will be resolved by finding it, if it exists, or creating it, if it does not. The instance will be created using its primary key (i.e., not its copy helper).
9. Accept the default pattern for locating the home (Factory Finder / Principal).
10. Accept the default location for the home (**Factory Finder Name**).
11. This information should match the **Name in Factory Finding Service Registry** for the managed object's home, in the application configuration information for the managed object (in the Managed Object Configuration wizard, Home page). Because the managed objects are not yet configured, you should accept the default for now.
12. Accept the default principal interface name.
13. Click the title and turn to the Data Object Interface page.
14. Add the key attributes (SavingsAccount1_accountNo and CheckingAccount1_accountNo) to the data object.
15. Click **Finish**. The business object implementation and data object interface appear under the business object interface.

Add a transient data object:

1. From the data object interface's pop-up menu, click **Add Implementation** to open the Data Object Implementation wizard.
2. Select **BOIM with any key** as the environment.
3. Select **Transient** as the form of persistence.
4. Click the page title and turn to the Key and Copy Helper page.
5. Select AllAccountsKey and AllAccountsCopy.
6. Click **Finish**. The data object implementation appears under the data object interface.

Add a managed object:

1. From the business object implementation's pop-up menu, click **Add Managed Object** to open the Managed Object wizard.

2. Click **Finish**. The managed object appears under the business object implementation.

The composite component is now defined.

Editing the composition

When you edit attributes or methods in the composition, the changes are automatically applied to the composite components based on the composition. In this task, you will consolidate the attributes `SavingsAccount1_balance` and `CheckingAccount1_balance` into a single balance attribute, that returns the sum of the two component's balances.

Consolidate the two balance attributes:

1. Locate the `AccountComposition` composition, in the `User-Defined Compositions` folder.
2. From the composition's pop-up menu, click **Properties** to open the Composition Editor.
3. In the Results pane, select the attributes `SavingsAccount1_balance` and `CheckingAccount1_balance`. Select the second attribute using the Ctrl key plus right-click, to both select it and display its pop-up menu at the same time.
4. From the pop-up menu, select **Equate**. The two attributes are consolidated into a single balance attribute.
5. Click on the new balance attribute to display its delegating behavior in the Current Republished Value pane.

By default, the consolidated attribute delegates to a sequence of the two source attributes. This means that it will return the balance of the last attribute in the sequence. To customize the method and have it return the sum of the two attributes, instead of just the last value in the sequence, you need to add some extra processing in the form of a private helper function.

Add a private helper function:

1. In the Results pane, display the pop-up menu for the `User-Defined Methods` folder and click **Add**. A new method with the default name `newOperation1` appears.
2. Click on the Properties tab to display the properties for the method.
3. Change its name to `addFloats`.
4. Change its return type to `float`.
5. Change its implementation to `Private`.
6. In the Results pane, expand the method to show the Parameters folder underneath it.

7. From the pop-up of the Parameters folder, click **Add**. A parameter with the default name newParameter1 is added to the folder. The properties of the parameter appear on the Properties page.
8. Change the parameter's name to arg1.
9. Change the parameter's type to float.
10. Add a second parameter named arg2, type float.

Change the delegation for balance:

1. In the Results pane, click on the balance attribute you consolidated earlier. The delegating behavior of its Getter method (to a <sequence> of SavingsAccount1.balance and CheckingAccount1.balance) appears in the Current Republished Value pane. There is no delegation for the Setter method, because the source balance attributes are read-only.
2. Delete the <sequence> node. It is replaced by an <empty> node.
3. From the pop-up menu of the <empty> node, click **Set value**. A list of attributes and methods with type or return type float appear.
4. Select addFloats as the value to map to. It replaces the <empty> node, and two parameter nodes (labelled ??) appear beneath it.
5. From the pop-up menu of the first ?? node, click **Set value**. Map the parameter to SavingsAccount1.balance. The selected attribute replaces the ?? node.
6. Map the second parameter in the same way, to CheckingAccount.balance.
7. Click **OK** to apply your changes to the composition, and return to the Object Builder main window.

Add the implementation for balance:

1. Click on the composition in the User-Defined Compositions folder. Its attributes and methods appear in the Methods pane.
2. Select the addFloats method in the Methods pane. Its skeleton implementation appears in the Source pane.
3. Add the following implementation to the Source pane:

```
return arg1+arg2;
```

The method delegation is complete. Calls to the combined balance attribute are automatically delegated to addFloats, which takes the two source balance attributes as parameters and returns their sum.

Review the changes in the composite component:

1. In the User-Defined Business Objects folder, locate the composite business object implementation AllAccountsBO.
2. Click on AllAccountsBO to display its attributes and methods in the Methods list.

It now has a balance attribute, that has replaced the SavingsAccount1_balance and CheckingAccount1_balance attributes.

3. Click on the balance attribute to display its implementation, which has already been filled in with appropriate delegation behavior.
4. Click **File > Save** to save your changes.

Configuring the build

Create client and server DLLs for the components. For this exercise, all the components will be configured into the same DLLs. The AllAccounts component is configured like a normal component. The AccountComposition composition is a server-only object.

If the AllAccounts component were configured into a separate DLL from the SavingsAccount and CheckingAccount components, then the AllAccounts DLLs would need to link with the other component's libraries (on the Libraries to Link With page of the wizards for the DLLs). This would be necessary to resolve the composite component's references to the composited managed objects.

Define the client DLL:

1. From the Build Configuration folder's pop-up menu, click **Add Client DLL** to open the Client DLL wizard.
2. Name the DLL AccountsClient.
3. Click the page title and turn to the Client Source Files page.
4. Select SAFile, SAFileKey, and SAFileCopy (the SavingsAccount client interfaces).
5. Select CAFile, CAFileKey, and CAFileCopy (the CheckingAccount client interfaces).
6. Select AAFile, AAFileKey, and AAFileCopy (the AllAccounts client interfaces).
7. Click **Finish**. The client DLL appears under the folder.

Define the server DLL:

1. From the Build Configuration folder's pop-up menu, click **Add Server DLL** to open the Server DLL wizard.
2. Name the DLL AccountsServer.
3. Click **Next** to turn to the Libraries to Link With page.
4. Select the AccountsClient library file.
5. Click **Next** to turn to the Server Source Files page.
6. Select SAFileBO, SAFileDO, SAFileDOImpl, and SAFileMO (the SavingsAccount server interfaces).
7. Select CAFileBO, CAFileDO, CAFileDOImpl, and CAFileMO (the CheckingAccount server interfaces).
8. Select AAFileBO, AAFileDO, AAFileDOImpl, and AAFileMO (the AllAccount server interfaces).

9. Select ACFile (the AccountComposition composition).
10. Click **Finish**. The server DLL appears under the folder.

Build the DLLs:

1. From the pop-up menu of the User-Defined Business Objects folder, click **Generate > All**.
2. Wait for the code generation to complete. The generated source files are placed in the project's \Working directory.
3. From the pop-up menu of the User-Defined Compositions folder, click **Generate > All**. The code for the composition is added to the \Working directory.
4. From the pop-up menu of the Build Configuration folder, click **Generate > All > All Targets**.
5. From the same pop-up menu, click **Build > All Targets**. The DLLs are built and placed in the project's \Working\PRODUCTION directory.

Configuring the application

Create the application family:

1. From the Application Configuration folder's pop-up menu, click **Add Application Family** to open the Application Family wizard.
2. Name the application AccountFamily.
3. Click **Finish**. The application family appears under the folder.

Create the application:

1. From the application family's pop-up menu, click **Add Application** to open the Application wizard.
2. Name the application AccountApplication.
3. Click **Finish**. The application appears under the application family.

Configure the SavingsAccount managed object:

1. From the application's pop-up menu, click **Add Managed Object** to open the Managed Object Configuration wizard.
2. Select SavingsAccountMO as the managed object. The other fields become filled in with appropriate defaults.
3. Click **Next** to turn to the Data Object Implementations page.
4. Select SavingsAccountDOImpl.
5. Click **Finish**. The correct container and home are selected by default. The managed object configuration appears under the application.

Configure the CheckingAccount managed object using the same steps, with the following differences:

1. Select CheckingAccountMO as the managed object.
2. Select CheckingAccountDOImpl as the data object implementation.

The managed object configuration appears under the application.

Configure the AllAccounts managed object using the same steps, with the following differences:

1. Select AllAccountsMO as the managed object.
2. Select AllAccountsDOImpl as the data object implementation.
3. Click the page title and turn to the Home page.
4. Review the **Name in Factory Finding Service Registry** path, and verify that it is the same as the location you provided in the Business Object Implementation wizard, Location page (in the **Factory Finder Name** field).
5. Click **Finish**. The managed object configuration appears under the application.

Generating the DDL files:

From the application family's pop-up menu, click **Generate**. The DDL files you need to install the application on the server are created and placed in the project's \Working\platform\PRODUCTION\AccountFamily\ directory.

You have now completely defined two components, a composition that combines their interfaces, and a composite component that allows access to the combined interfaces.

You can find information on installing components on the server in the System Administration Guide, "Configure a New Application Environment" chapter, or online in the topic Installing and configuring a new application.

You can find information on testing the installed components with a QuickTest client application in "Chapter 13. Testing applications with QuickTest" on page 611.

Defining relationships

The following tasks cover the different types of relationships you can define between components, in Object Builder:

- "Defining a 1-1 relationship" on page 279
- "Defining a 1-n relationship" on page 281
- "Defining a circular relationship" on page 283
- "Defining a foreign key pattern" on page 285
- "Storing an object reference as a handle" on page 288
- "Using complex relationships in SQL clauses" on page 289

RELATED CONCEPTS

Object relationships (*Programming Guide*)

“Foreign key patterns” on page 284
Model details (Object identity) (*Programming Guide*)
Data object customization for cardinality relations (*Programming Guide*)
Expanding the client programming interface (*Programming Guide*)
Handles (*Programming Guide*)
“Chapter 6. Inheritance” on page 299

RELATED TASKS

“Creating a child component” on page 306

Defining a 1-1 relationship

When you add an attribute whose type is another business object, you create a cardinality-to-1 relationship between the first object (which has the attribute) and the second object (which is the type of the attribute).

To create an attribute that references an object, follow these steps:

1. Open the Business Object Interface wizard (either by adding a new business object interface to a file or module, or by selecting **Properties** from the pop-up menu of an existing business object interface).
2. Click the page title and turn to the Attributes page.
3. From the Attributes pop-up menu, select **Add**.
4. Type the name of the attribute (for example, `currentClaim`).
5. From the **Type** drop-down list, select the type for the object that you want to reference (for example, `Claim`).
6. Enter any other information that defines the attribute.
7. Complete the remaining wizard pages, or click **Finish**.

The object you reference should already exist in Object Builder, at least as a business object interface declaration (without methods or attributes). If you have two objects that reference each other, create the references as follows:

1. Define the first interface (for example, `Policy`) without defining its methods or attributes.
2. Define the second interface (for example, `Claim`) with a reference to the first interface.
3. Go back and edit the first interface, to add a reference to the second interface.

RELATED CONCEPTS

Expanding the client programming interface (Using handles) (*Programming Guide*)
Business object (*Programming Guide*)
Object relationships (*Programming Guide*)

RELATED TASKS

“Defining relationships” on page 278

Distributed query

► NEW

Distributed queries are those that you perform over collections of objects that are found on different servers. Factories and factory finders play important roles in a distributed environment. Factory finders, with the help of location objects are responsible for locating the objects that you query, on the different servers in the distributed environment.

Structural and functional enhancements to the business object implementation, and data object implementation make distributed queries possible.

In the business object implementation, the `list()` method for a 1-n foreign key relationship has the capability of locating the home of the child interface using a factory finder name and factory key string, and then evaluating a query against that home. Before distributed query was supported, the `list()` method could only evaluate a query on the current server.

In the data object implementation, the getter for a home/key mapped object reference has the capability of locating the home of the parent interface using a factory finder name and factory key string.

When you perform a distributed query, you can select the style to be used for the query: either the current path expression form of the query, or the foreign key pseudo attribute form of the query. You can select the style in the Foreign Key Implementation section of the business object implementation’s Object Relationships page.

Note: The query engine cannot handle path expressions involving collections over multiple servers. But, Object Builder emits a foreign key query that uses a path expression to get from the child interface to the primary key attributes of the parent interface. In this situation, it is recommended that you surface the foreign key attributes of the child table in the business object to data object, and data object to data access object `MappedTypes` in the SM DDL, and then use these pseudo attributes in the query.

RELATED CONCEPTS

“Complex attributes and mapping patterns” on page 745

“Foreign key patterns” on page 284

Query Service (*Advanced Programming Guide*)

Concepts of factories (*Advanced Programming Guide*)

Concepts of factory finders (*Advanced Programming Guide*)

RELATED TASKS

“Defining a 1-n relationship”

“Defining a foreign key pattern” on page 285

Defining a 1-n relationship

You can define a one-to-many (1 to n) relationship between components. When you define a relationship, a set of patterned methods are added to the first component to support the relationship, to allow adding, deleting, and listing of its related objects.

To create a relationship between components, follow these steps:

1. Open the Business Object Interface wizard (either by adding a new business object interface to a file or module, or by selecting **Properties** from the pop-up menu of an existing business object interface).
2. Click the page title and turn to the Object Relationships page.
3. From the Relationships pop-up menu, select **Add**.
4. Type a name for the relationship.
5. Select the type for the objects that you want to define a relationship with.
6. Select whether you want the relationship that you are defining to be read-only.
7. Complete the remaining wizard pages (if this is a new interface), or click **Finish**.

If you do not select the read-only option for the relationship, the business object interface will now have methods for adding, removing, or listing objects of the selected type. For example, if you established a 1-n relationship from Policy to Claim, Policy would now have the methods addClaim, removeClaim, and listClaim. These methods allow a client to add and remove Claim instances through the Policy class, and to iterate through a list of the Claim instances to which Policy is related.

If you select the read-only check box on the Object Relationships page, the business object interface will have only the method for listing objects of the selected type; it will not have those for adding and removing objects.

If you are defining a 1-n relationship as part of a foreign key pattern, and if the foreign key reference is read-only (for example, if Customer’s attribute myAgent is read-only because it is part of CustomerKey), then you should make the foreign key relationship (for example, Agent’s relationship to Customer) read-only as well. Otherwise, the relationship’s add() and remove() methods will throw exceptions at run time. In this situation, besides maintaining the referenced objects, the application developer must manage the foreign key attributes by managing their instances.

Now set the implementation of the relationship in the business object implementation:

1. Open the Business Object Implementation wizard (either by adding a new business object implementation to the previous interface, or by selecting **Properties** from the pop-up menu of its existing business object implementation).
2. Click the page title and turn to the Object Relationships page. This page lists the relationships defined in the business object interface, as well as those that are declared on the parents of the business object implementation's business object interface.
3. Click on the relationship that you want to implement.
4. Under **Relationship Implementation**, set the type of implementation:
 - **Local persistent reference**
Select this option if you want to use a reference collection and tie its implementation to the data object implementation's "Environment" on page 249. If the data object implementation's environment is **BOIM with UUID key**, the collection will be transient, otherwise the collection will be persistent.
 - **Transient Reference Collection**
Select this option if you want your reference collection to be transient.
 - **Persistent Reference Collection**
Select this option if you want your reference collection to be persistent.
 - **User-Defined OOSQL Reference**
The relationship will be implemented using logic you provide. Only skeleton methods will be generated.
 - **Reference resolved using foreign key**
The relationship is implemented using the foreign key pattern. This is described in a separate task.
5. Click **Finish**.

You have defined the 1-n relationship. Depending on the type of implementation you selected, you will need to do additional work in the business object's framework methods, or in the data object.

1-n relationships with inherited classes

Generally, you set up a 1-n relationship between an object and another one that has an object reference to the first object. You can now have diagonal 1-n relationships as well (that is 1-n relationships between an object, and the child object of another object). Consider the following situations:

- There are three objects: Department, Person, and Employee. Person has a foreign key, and an object reference to Department. Employee is the child object of Person, and inherits Person's interface, but has no direct object reference to Department. A 1-n relationship is possible from Department to Employee.

- There are three objects: Person, Department, and Research. Research inherits Department's interface. Person has a foreign key, and an object reference to Department. Person does not have an object reference to the child object Research. You can set up a 1-n relationship from Research to Person.

RELATED CONCEPTS

Business object (*Programming Guide*)

Object relationships (*Programming Guide*)

"Foreign key patterns" on page 284

RELATED TASKS

"Defining relationships" on page 278

"Adding a business object interface" on page 777

"Adding a business object implementation and data object interface" on page 780

"Defining a 1-1 relationship" on page 279

"Defining a foreign key pattern" on page 285

RELATED REFERENCES

"Relationship Implementation" on page 293

Defining a circular relationship

When two components reference each other (through attributes or one-to-many relationships), the relationship is bidirectional, or circular.

Circular relationships cannot cross module boundaries. Both interfaces must be defined in the same module, or else they cannot be in modules at all (though they can be in separate files).

To create a circular relationship between two components, follow these steps:

1. Create the first interface, without its reference or relationship to the second interface.
2. Create the second interface, with its reference or relationship to the first.
3. Edit the first interface, and add its reference or relationship to the second.

A foreign key pattern is a specific case of a circular relationship. It is documented in full in the foreign key pattern task.

RELATED CONCEPTS

"Foreign key patterns" on page 284

RELATED TASKS

"Defining relationships" on page 278

"Defining a foreign key pattern" on page 285

Foreign key patterns

When a schema contains a foreign key reference (for example, the schema for Customer has a foreign key reference to Agent), this allows for more efficient relationships on the component level. For example, if Agent has a one-to-many relationship with Customer, calls to find a particular customer can be resolved on the database level, instead of on the business object level.

To take advantage of a foreign key reference on the component level, you need to define a component with a foreign key attribute (based on the foreign key reference), and then edit the component referenced by the foreign key attribute, to add a one-to-many relationship in the other direction (resolving references by foreign key).

For example, the component Agent has a one-to-many relationship with the component Customer, and the component Customer has an inverse object reference to the component Agent (each agent can have multiple customers, but each customer is represented by only one agent).

Foreign key relationships give better performance with SQL queries, because the references resolve directly to a database table, rather than indirectly through business object and data object attributes.

Once you define these relationships on the component level (a one-to-many relationship with foreign key support, and inverse references based on foreign keys), the foreign key attribute (for example, Customer's inverse reference to Agent) can be mapped to a foreign key in the imported .sql for the component.

When you are doing top-down programming, Object Builder currently will not identify foreign keys in .sql files that it generates. While you can define foreign key relationships using the foreign key pattern, and generate .sql files based on the component relationship, the generated .sql files will not contain a foreign key constraint. Query pushdown (which results in substantial performance benefits) can still occur even if the foreign key constraint is not present in the actual table definition. However, DB2 will not be able to access the table as efficiently as when there is a foreign key constraint.

However, Object Builder recognizes and retains (in the generated .sql files) any foreign key constraint that is in the model as a result of SQL import.

RELATED CONCEPTS

Object relationships (*Programming Guide*)

Data object customization for cardinality relations (*Programming Guide*)

RELATED TASKS

“Defining a foreign key pattern”
“Customizing referential integrity” on page 714

RELATED REFERENCES

“Foreign Key Implementation” on page 296

Defining a foreign key pattern



When a schema contains a foreign key reference (for example, the schema for Customer has a foreign key reference to Agent), this allows for more efficient relationships on the component level. For example, if Agent has a one-to-many relationship with Customer, calls to find a particular customer can be resolved on the database level, instead of on the business object level.

To take advantage of a foreign key reference on the component level, you need to define a component with a foreign key attribute (based on the foreign key reference), and a component with a one-to-many relationship (resolving references by foreign key).

To define these relationships, follow these steps:

1. Import the SQL DDL files that define the schemas for the related components (for example, `myDB.Customer` and `myDB.Agent`).
2. Create persistent objects from the schemas (for example, `CustomerPO` and `AgentPO`).
3. Create business object interfaces (for example, `Customer` and `Agent`). Specify their names only, do not specify their attributes or relationships.
Note: You cannot have a foreign key relationship between business object interfaces that are contained in different modules. The interfaces can be contained in the same module.
4. Complete the business object interface that owns the foreign key reference. Make sure the interface includes an object reference equivalent to the foreign key constraint in the table.

The object reference represents a many-to-one relationship (many Customers share one Agent). You create this relationship in the same way you would create a one-to-one relationship, by creating an attribute of the referenced type (for example, the business object interface is defined with an attribute `myAgent` of type `Agent`). This is the foreign key attribute. The foreign key attribute cannot be read-only, unless it is part of the primary key.

5. Create the business object implementation, key and copy helper, data object interface, and implementation for the owner of the foreign key reference (for example, `CustomerKey`, `CustomerCopy`, `CustomerBO`, `CustomerDO`, `CustomerDOImpl`).

Make sure the foreign key attribute (for example myAgent) is part of the component's state data, and is identified in the component's key.

6. Complete the business object interface and implementation referenced by the foreign key, and define its one-to-many relationship (for example, add a one-to-many relationship from Agent to Customer, so that each agent can have multiple customers).

To define a one-to-many relationship with references resolved by foreign key, follow these steps:

- a. Open the Business Object Interface wizard by selecting **Properties** from the pop-up menu of the business object interface.
- b. Click the page title and turn to the Object Relationships page.
- c. From the Relationships pop-up menu, click **Add**.
- d. Type a name for the relationship.
- e. From the **Object Type** drop-down list, select the interface that has the foreign key reference (for example, Customer).

If the foreign key reference is read-only (for example, if Customer's attribute myAgent is read-only because it is part of CustomerKey), then you should make the foreign key relationship (for example, Agent's relationship to Customer) read-only as well. Otherwise the relationship's add() and remove() methods will throw exceptions at run time. In this situation, besides maintaining the referenced objects, the application developer must manage the foreign key attributes by managing their instances.

- f. Click **Finish**.
- g. From the pop-up menu of the interface, click **Add Implementation** to open the Business Object Implementation wizard.
- h. Click the page title and turn to the Object Relationships page.
- i. Under the Relationships folder, click on the relationship you defined in the interface. You can now set the implementation behavior for the relationship.
- j. Under **Relationship Implementation**, click **Reference resolved using foreign key**.

Note the following points:

- This option is enabled only when one of the following criteria is satisfied:
 - both interfaces are defined in the same IDL file or both interfaces are defined in separate files but not within modules, or
 - the selected interface has a reference to the current interface
- If the interfaces that are to be used in the foreign key relationship are in different files, and either one or both interfaces are contained in modules, you cannot create the foreign key relationship.

- k. From the **Foreign Key Attribute** list, select the attribute of the object that you want to use as the foreign key in this relationship. The list only displays attributes with the same type as the current object (for example, Customer's attribute myAgent of type Agent).
 - l. In the **Home to Query** field, specify the home that will be used on the server to find objects of the selected type. The home that you select must be the same one that you configure with the target component's managed object.
Typically the home name is derived from the target managed object's name (for example, CustomerMOHome).
 - m. If you want to set up a foreign key relationship between a parent and a child object, where the parent and child homes reside on different servers, select the **Distributed query** check box. Object Builder then emits a query that does not use path expressions, into the `list()` method of the parent to return all child objects that reference back to the parent.
 - n. Optionally, you can specify a factory name, and a factory finder name.
 - o. Click **Finish**.
7. Complete the rest of the component objects (for example, AgentKey, AgentCopy, AgentBO, AgentDO, AgentDOImpl).
 8. Complete the component that owns the one-to-many relationship by mapping the data object implementation of the component to its equivalent persistent object (for example, map AgentDOImpl to AgentPO):
 - a. In the implementation's wizard, add an instance of the persistent object to the implementation's Associated Persistent Objects page.
 - b. Turn to the Attributes Mapping page and map the data object attributes to the persistent object attributes.
 - c. Turn to the Methods Mapping page and map the framework methods there to methods of the persistent object.
 9. Complete the component that owns the foreign key reference by mapping the data object implementation of the component to its equivalent persistent object (for example, map CustomerDOImpl to CustomerPO):
 - a. In the implementation's wizard, add an instance of the persistent object to the implementation's Associated Persistent Objects page.
 - b. Turn to the Attributes Mapping page and map the data object attributes to the persistent object attributes.
 - c. Map the foreign key attribute using the **Key Home** option, and then map the key attributes to their equivalents in the persistent object.
 - d. Turn to the Methods Mapping page and map the framework methods there to methods of the persistent object.

The foreign key pattern is now established.

Note: Models from older versions of Object Builder that have distributed queries will use the path expression form of the query in the list() method, and will have the default factory names and factory finder names. (The **Distributed query** check box on the Object Relationships page of the business object implementation will not be selected.)

RELATED CONCEPTS

“Foreign key patterns” on page 284
Component (*Programming Guide*)
“Home” on page 581

RELATED TASKS

“Defining relationships” on page 278
“Defining a 1-1 relationship” on page 279
“Defining a 1-n relationship” on page 281
“Mapping a data object to a DB persistent object” on page 703
“Mapping attributes using a key” on page 732
“Customizing referential integrity” on page 714

Storing an object reference as a handle

You can store object references either as handles, or as home key mappings. A handle is an encoding of the reference that can be used to recreate the reference. A home key mapping involves the persistence of the attribute values in the object’s key, as described in the foreign key pattern for storing a foreign key reference.

This task describes how to store an object reference as a handle.

Object Builder supports the following handle patterns for a persistent reference:

- Stringified Object Reference (SOR)
- Object Name
- Home Name and Key

The handle pattern used to store references to a particular object type is set in the business object implementation of that object. The handle pattern can be overridden, however, by the referencing object, as set in the data object implementation of the referencing object.

To set or change the default handle pattern for a particular object type, follow these steps:

1. Open the Business Object Implementation wizard (either by adding a new business object implementation to an interface, or by selecting **Properties** from the pop-up menu of an existing business object implementation).

2. Click the page title and turn to the Handle Selection page.
3. Select the handle that will be used by default to store references to this type of object.
4. Complete the remaining wizard pages (if this is a new business object implementation), or click **Finish**.

To override the default behavior and use a single storage pattern for references to all types of objects, follow these steps:

1. Open the Data Object Implementation wizard (either by adding a new data object implementation to an interface, or by selecting **Properties** from the pop-up menu of an existing data object implementation).
2. Turn to the Behavior page..
3. Under “Handle for Storing Pointers” on page 255, select the handle you want to use for swizzling pointers.
4. Complete the remaining wizard pages (if this is a new data object implementation), or click **Finish**.

RELATED CONCEPTS

Handles (*Programming Guide*)

Object relationships (*Programming Guide*)

Data object customization for cardinality relations (*Programming Guide*)

“Foreign key patterns” on page 284

RELATED TASKS

“Defining relationships” on page 278

“Defining a foreign key pattern” on page 285

“Adding a business object implementation and data object interface” on page 780

“Adding a data object implementation” on page 807

Using complex relationships in SQL clauses

In the SQL View Editor, you can specify conditions or relationships that must exist among rows of the various schemas, for them to be included in the view. You can do this on the Where page and the Having page of the Clauses pane. These conditions are also called predicates, and they can be combined in the following ways:

- All predicates must be satisfied (“AND”)
- At least one predicate must be satisfied (“OR”)
- A more complex arrangement of “AND”, “OR” and “NOT” conditions must be satisfied

Each of these conditions can be expressed by selecting the corresponding button (**And**, **Or**, or **Use Complex Relationships**) at the bottom of the page.

Whichever condition you specify, you can see a graphical representation of the logic behind the condition by clicking the **Edit Conditional Relationships** button.

To add a complex condition to the SQL clause when creating or editing a view, follow these steps:

1. Click the **Edit Conditional Relationships** button.
The Organize Logical Combination dialog opens. Each predicate is represented by either an entry in the list box on the left, or by a rectangle in the graph on the right. Predicates in the list box do not belong to any logical combination; those in the graph do.
2. You can manually change the logical combination of the predicates in the graph view by doing one of the following tasks:
 - Adding a predicate to the graph
 - Removing a predicate from the graph
 - Negating a predicate
 - Negating a combination of predicates

To add a predicate to the graph, follow these steps:

1. Select the predicate from the list in the left-hand frame. The statement of the predicate appears below the list box.
Note: If there are any other elements in the graph, you must select at least one with which the new predicate will be combined. For example, if you choose predicate 'A' from the list, and then choose predicate 'B' from the graph, you are then allowed to combine 'A' with 'B', with either an "AND" condition or an "OR" condition. You may choose as many predicates from the graph as you wish, and the editor will add the new predicate according to the combination you requested.
2. Once you have selected the predicates, you can select one of two buttons below the list box: **Add as And>>** and **Add as Or>>**. Your selection determines how the new predicate will be combined with the selected ones.

To remove a predicate from the graph, follow these steps:

1. Select the predicate with the mouse.
Note: As you move the mouse over a predicate in the graph, the text of the predicate appears.
2. Click the <<**Remove** button.

Notice the following indications on the graph:

- To the left of each predicate is a white NOT indicator. You can use it to negate a condition. As you move the mouse over this symbol, a red outline appears around the predicate. This implies that the NOT operator is

associated with that predicate. Click the NOT operator to negate the predicate indicated by the rectangle. Once a predicate is negated, its associated NOT operator appears red.

- To the left of each combination of predicates, there is also a yellow tilde. When you move the mouse over this tilde, a red outline appears around the entire logical combination.

To negate a predicate (specify “NOT” conditions), follow these steps:

1. Move your mouse over the predicate and click the NOT operator closest to it. The yellow tilde turns red, indicating that the predicate is negated.

To negate a logical combination, follow these steps:

1. Move your mouse over the combination of predicates and click the outermost NOT operator. The yellow tilde turns red, indicating that the combination as a whole is negated. For example, if the combination shows ‘A’ AND ‘B’, then by selecting the tilde for this combination, it becomes NOT (‘A’ AND ‘B’).

Note: When you close the dialog, one of the radio buttons: **And**, **Or**, or **Use Complex Relationships** will be selected according to the state of the arrangement. If **Use Complex Relationships** is selected, it implies that one of the following conditions exists in the arrangement:

- There are predicates left in the list box that are not yet placed in combination
- The picture in the graph is a combination of AND, OR and NOT conditions, and cannot be reflected using either a simple AND or a simple OR combination.

RELATED CONCEPTS

Schema (*Programming Guide*)

RELATED TASKS

“Working with DB schemas” on page 843

“Working with the SQL View Editor” on page 850

Design patterns and iterators

Design patterns

A design pattern describes a problem that occurs repeatedly in our environment. It then describes the core of the solution to the problem, in such a way that you can use this solution innumerable times without doing it the same way twice.

Design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context. One object’s pattern can be another one’s building block.

Design patterns are less specialized and smaller architectural elements than frameworks, but they are not frameworks, though they are more abstract than them.

Several design patterns can be contained in a framework, but the reverse is never true.

Design patterns must be implemented each time they are used, whereas you can embody frameworks in code and use them directly.

Design patterns can be used in any kind of application; frameworks always have a particular application domain.

Examples of design patterns are object factories, iterators, mediators, proxies and bridges.

Iterators

A collection is a group of objects, and objects model real-world entities. So, very often, you need to access either references to objects, or the objects themselves (their references or indirection is hidden).

An iterator is a design pattern that defines three operations to traverse a collection (access objects directly or indirectly in that collection):

- *reset* points to the start of a collection
- *next* increments the iterator's position
- *more* enables you to test if there are elements left in the iteration. This method returns *true* if there are more elements that you can access in the collection; it returns *false* if you have reached the end of the collection.

Every time they are used on the server, iterators, like all design patterns, must be implemented.

A data object iterator supports data objects that are backed directly by DB2 queries.

Dependencies within an IDL file

When you add modules, interfaces, or constructs to an IDL file, they are automatically re-ordered if necessary to resolve any internal dependencies.

You can view and change this order by displaying the wizard for an existing IDL file (select a business object file or data object file in the Tasks and Objects pane, and select **Properties** from its pop-up menu). The order appears on the Contents Ordering page.

When a construct or interface references another construct or interface that comes after it in the file, the dependency is resolved in one of two ways:

- If the dependency is within the same scope (the referencing and referenced element are both at the file level, or both in the same module), then a forward declaration is automatically included to resolve the reference.
- If the dependency is cross-scope (the referencing and referenced element are at different scopes), then the order must be changed because a forward declaration in IDL cannot be cross-scope.

You can view the type and scope dependencies for an IDL element by selecting it on the Contents Ordering page:

- An interface dependency is listed when the interface has an attribute, method return type, method parameter type, method exception type, object relationship type, construct type, or construct member type that references another interface or construct in the same file. If the referenced interface or construct is in another module, then the dependency is listed as being on the module.
- A construct dependency is listed when the construct is of a type, or contains a member of a type, that references another interface or construct in the same file. If the referenced interface or construct is in another module, then the dependency is listed as being on the module.
- A module dependency is listed when it contains an interface or construct that has a dependency.

The order of the contents is automatically checked for validity, and re-ordered if necessary, whenever you click **Finish** in the wizard for a module or interface contained in the file.

Note: You cannot have circular dependencies between constructs.

RELATED CONCEPTS

Interface Definition Language (*Programming Guide*)

RELATED TASKS

“Working with constructs” on page 769

Relationship Implementation

Relationship Implementation

In a one-to-many relationship, you can force the kind of reference collection to be used if you know in advance how you will be configuring the referenced, child managed object. You are limited in your choice of objects that you can select when you define a container to hold the components of your application, depending on your configuration of the managed object. You can explicitly choose one of the following reference collections:

- “Local Persistent Reference”
- Transient Reference Collection
- Persistent Reference Collection
- User-defined OOSQL Query “OOSQL Implementation” on page 295
- Reference resolved by foreign key “Foreign Key Implementation” on page 296

Transient Reference Collection

Select this option if you know in advance that you will be using a referenced child managed object that is configured for transient applications.

Persistent Reference Collection

Select this option if you know in advance that you will be using a referenced child managed object that is configured for use with persistent applications.

RELATED CONCEPTS

Using sets of objects (Using Reference Collections) (*Programming Guide*)

Object relationships (*Programming Guide*)

“Foreign key patterns” on page 284

“Home” on page 581

Query Service (*Advanced Programming Guide*)

RELATED TASKS

“Defining a 1-n relationship” on page 281

“Configuring a managed object” on page 588

“Creating a container instance” on page 578

“Defining a 1-1 relationship” on page 279

“Defining a foreign key pattern” on page 285

Local Persistent Reference

This option is selected by default. This is the only type of reference collection implementation used if there are no object references. This type of implementation can be used even if there is a reference between the objects.

In a one-to-many relationship, if the parent data object implementation (the one that has a relationship with many other objects) is **BOIM with UUID key**, we use a transient reference collection.

In a one-to-many relationship, if the parent data object implementation is **BOIM with any key** (either transient, embedded SQL, Cache service, or procedural adaptors), we use a persistent reference collection.

Note the following points:

- If you select this type of implementation, a read-only, protected attribute, with the same name as the relationship is created for the business object. It

will not appear on the Attributes page of the Business Object Interface wizard, but you will be able to select it as state data (on the Data Object Interface page) for any data object that you want to associate with this business object.

- If you later edit the business object implementation and change the relationship implementation, you will have to delete the attributes that were created for relationships using local persistent reference.

RELATED CONCEPTS

Using sets of objects (Using Reference Collections) (*Programming Guide*)

Object relationships (*Programming Guide*)

“Foreign key patterns” on page 284

“Home” on page 581

Query Service (*Advanced Programming Guide*)

RELATED TASKS

“Defining a 1-1 relationship” on page 279

“Defining a 1-n relationship” on page 281

“Defining a foreign key pattern” on page 285

RELATED REFERENCES

Type of Persistence

OOSQL Implementation

User-defined OOSQL query

If you select this option, the following additional methods will be implemented for the business object implementation: `add()`, `list()`, and `remove()`. You can customize the `list()` method using the Source pane. Type an OOSQL query for the method body. The `add()` and `remove()` method bodies are empty by default.

RELATED CONCEPTS

Using Sets of Objects (Using Reference Collections) (*Programming Guide*)

Object Relationships (*Programming Guide*)

“Foreign key patterns” on page 284

“Home” on page 581

Query Service (*Advanced Programming Guide*)

RELATED TASKS

“Defining a 1-1 relationship” on page 279

“Defining a 1-n relationship” on page 281

“Defining a foreign key pattern” on page 285

Foreign Key Implementation

► CHANGED

Reference resolved by foreign key

This option is available when there is an object reference between this business object and the one with which you are defining the relationship. This implementation uses an identifier to access the individual links between related objects.

This option is disabled unless the following conditions are met:

- **Inverse reference:** At least one attribute of the object with which you are defining a relationship for the current object must be of a type which is the interface of the current object. For example, there are two business object interfaces named Policy and Claim. To define a foreign key relationship from within Policy, to Claim, the Claim interface, or a parent thereof, must have at least one attribute of the type Policy.
- **Scoping:** The interfaces of the business objects that reference each other can be contained in the same module or in the global scope. You cannot have a foreign key relationship between business object interfaces that are contained in different modules.

Note the following points on permissions on the inverse reference:

- The attribute that is an inverse reference can either be editable (the associated business object interface has the add(), remove(), and list() methods but will throw exceptions if they are called); or read-only (the associated business object interface has neither the add() nor the remove() method, but only the list() method).
- This inverse reference will be read-only if you configured the relationship to be read-only when you defined or edited the business object interface (Object Relationships page of the Business Object Interface wizard).

?

Use this button to bring up the Foreign Key Assistant. It informs you whether the conditions required to create a reference collection using a foreign key are met.

The following fields are activated only if you are able to use a reference resolved by a foreign key:

- **Foreign Key Attribute**

Click the list button and select an attribute from the list. The attributes available for selection are attributes of the referenced object that reference the current object.

Restriction: Even if there is a key defined for another business object and it is designated as a foreign key, when you create a persistent object and schema for a business object referenced by the other object, it will not automatically create a foreign key in the schema.

- **Home to Query**
Specify the home of the object being referenced. This home that will be used on the server to find objects of the selected type. The home that you select must be the same one that you configure with the target component's managed object.
- **Factory Name**
Type the name of the factory from which the home inherits. This is the factory, in addition to the CORBA Life Cycle generic factory, from which every home inherits.
- **Factory Finder**
Type the name of the factory finder. This name is optional. If you provide it, it will be emitted in the generated code instead of the default *SERVERNAME-server-scope-widened* name, but you cannot provide a substitution value such as **SERVERNAME*.
- **Distributed Query**
Select this check box to indicate that the query is to be distributed over different components. That is, the interfaces of objects in the relationship can be deployed on different servers. Selection of this check box also changes the form of the query for foreign key relationships: the foreign key pseudo attribute form of the query is used. If this check box is not selected, the current path expression form of the query is generated.
By default, this check box is not selected, and it remains so for migrated models as well. That is, models from older versions of Object Builder that have distributed queries will use the path expression form of the query in the `list()` method, and will have the default factory names and factory finder names unless you explicitly change this default.

Restriction: If the primary key of the object of the 1-n relationship (the child object) includes an attribute whose type is of another business object interface, then the OOSQL query in the `list()` method of the object that has the relationship (the parent object) may not function properly and may need to be modified by hand.

RELATED CONCEPTS

Using Sets of Objects (Using Reference Collections) (*Programming Guide*)

Object Relationships (*Programming Guide*)

"Foreign key patterns" on page 284

"Home" on page 581

Query Service (*Advanced Programming Guide*)

Concepts of factories (*Advanced Programming Guide*)

Concepts of factory finders (*Advanced Programming Guide*)

RELATED TASKS

"Defining a 1-1 relationship" on page 279

"Defining a 1-n relationship" on page 281

“Defining a foreign key pattern” on page 285

“Mapping business object reference attributes” on page 744

Checking for null foreign key values



CORBA has no concept of an attribute having a null value. If you retrieve a null foreign key attribute from a database, the ensuing `findByPrimaryKey` method may or may not return the null value it should, depending on what arbitrary value Object Builder applied to the attribute (since it cannot use null).

Since this can produce unpredictable results, you tell Object Builder to immediately return a NIL value when it detects that a foreign key is null. The `findByPrimaryKey` method will not be called.

To set this option:

1. Select **Properties** from the Data Object Implementation’s pop-up menu.
2. In the Properties wizard, select the Attribute Mapping page.
3. Select a primary key attribute from the tree.
4. Select the **Check for null** check box.

With this option set, you will always get a NIL return value if the foreign key is null.

RELATED CONCEPTS

“Null value tolerance with sentinel values” on page 155

RELATED TASKS

“Setting sentinel values for null field values” on page 701

Chapter 6. Inheritance

You can inherit data and behavior between components in Object Builder.

You do not need to explicitly inherit between objects in the same component (for example, a business object and data object, or business object and copy helper). The relationship between the objects is handled by Object Builder.

You do not need to include any of the framework interface files for Component Broker frameworks that your components inherit from. This also is handled by Object Builder.

Child components can inherit full implementations from their parent component, or only the interface.

When you create a child component with interface inheritance, only the child business object interface needs to inherit from the parent. Then, in the child business object implementation, the inherited interfaces can be implemented (by selecting to override the parent methods and attributes in the Business Object Implementation wizard). The rest of the child component objects do not have inheritance.

When you create a child component with implementation inheritance, the child component objects generally inherit from their equivalent parent objects:

- Object Builder automatically generates in the child business object file the include statement that contains the parent business object file.
- The child business object interface must inherit from the parent interface.
- It may not be necessary to have a child key and copy helper. If the child has the same key attribute as the parent, it can re-use the parent's key. If the child does not add any new attributes, it can re-use the parent's copy helper. If you do add a child key and copy helper, then they can either inherit from their equivalents in the parent component, or they can contain selected attributes of the parent interface, without inheriting from the parent key or copy helper.
- The child business object implementation must inherit from the parent implementation.
- The child data object interface must inherit from the parent data object interface.
- The child data object implementation must inherit from the parent data object implementation.
- The child managed object must inherit from the parent managed object.

For data inheritance to work, the type of persistence provided by the parent and child data object implementations must be the same.

RELATED CONCEPTS

Components (*Programming Guide*)

“Inheritance and overriding in helper objects”

“Inheritance and overriding in business objects” on page 301

“Inheritance and overriding in data objects” on page 303

“Abstract base class inheritance” on page 303

“Choosing an inheritance pattern for persistence” on page 304

“Inheritance with attributes duplication” on page 307

“Inheritance with key duplication” on page 322

“Inheritance with a single datastore” on page 341

“Inheritance with views” on page 357

RELATED TASKS

“Creating a child component” on page 306

Inheritance and overriding

Inheritance and overriding in helper objects

When you create the key and copy helper for a child component, you have the option of including some or all of the parent’s equivalent attributes as part of the helper.

For a child’s key, you have the following options:

- Use the parent’s key.
If the child has the same key attributes as the parent, there is no need to create a separate key; you can simply re-use the one created for the parent. In the child’s Data Object Implementation wizard, on the Key and Copy Helper page, select the parent’s key.
- Use a mix of parent key attributes and child key attributes.
In the child’s Key wizard, on the Name and Key Attributes page, you have parent key attributes available for selection. Select some or all of these, and then select additional identifying attributes that are unique to the child.
- Use all the parent key attributes and additional child key attributes
In the child’s Key wizard, on the Name and Key Attributes page, select the child attributes you want to be part of the key. Do **not** select any of the parent attributes. Click **Next** to turn to the Implementation Inheritance page, and select the parent key to inherit from. The child’s key then inherits the parent’s key attributes, in addition to having the child key attributes specified on the previous page.

- Use only child key attributes.
If the parent object has no identity in common with the child, then there is no reason for their keys to be related. You can create an entirely new key to reflect the child's unique identity, which includes none of the parent's attributes, and has default inheritance only.

For a child's copy helper, you have the same choices. The choice that makes sense will depend on the creation scenarios in which you intend to use the copy helper.

RELATED CONCEPTS

"Chapter 6. Inheritance" on page 299

"Choosing an inheritance pattern for persistence" on page 304

Key (*Programming Guide*)

Copy helper (*Programming Guide*)

RELATED TASKS

"Creating a child component" on page 306

Inheritance and overriding in business objects

You can inherit both business object interface and business object implementation from the parent. In the business object implementation, you can select which attributes, methods, and relationships you want to override. Generally, you would do so if you wanted to change the way these attributes mapped to, or interacted with, the data object.

If you have business object implementation inheritance, you must have corresponding data object and data object implementation inheritance. Otherwise, your application will fail at run time.

For example, if you have the following scenario, where Employee and Person are business objects:

Employee inherits from Person,

EmployeeBO inherits from PersonBO,

then, EmployeeDO must inherit from PersonDO, and

EmployeeDOImpl must inherit from PersonDOImpl.

However, if you override an attribute, method or relationship from the parent business object implementation in the child business object, and since the child data object must inherit from the parent data object, you cannot push (delegate) any of the overridden attributes or methods from the child business object implementation to the child data object. If you do, the overridden attribute, relationship, or method will be defined twice, once through its association with the business object, and once through its inheritance from the parent.

There are three main situations in which you would override in the child's implementation:

- **Overriding all attributes and relationships and inheriting behavior**
You can inherit behavior (method implementations) from a parent class, while overriding all its attributes. This is only appropriate for parent classes that have no data in the data object. The parent will have a business object interface, business object implementation, and a data object interface that contains no data. The child will inherit from each of the parent objects.
- **Overriding all attributes, relationships, and behavior**
You can use a parent class for interface-only inheritance, by overriding all its attributes, relationships, and methods in the business object implementation. The child will inherit from the parent business object interface only. This pattern also applies to abstract base class inheritance. You can push that attribute down to the child data object as long as there is no data object inheritance.
- **Overriding no attributes or relationships, overriding some or all behavior**
There are no restrictions on overriding methods, except for PA push-down methods (which have the same restrictions as attributes).

Note the following points:

- It is completely safe to override all methods in the child business object.
- Overriding attributes in the child business object is dangerous as it could lead to a violation of CORBA and inheritance specifications as mentioned above, resulting in double definitions on the child.
- If you choose to override an attribute that is implemented by getters and setters on the parent business object implementation, in the child business object, it must be solely for the purpose of having a custom override; it should not be for delegation of the attribute to the data object. (When you have pure interface inheritance, there is no parent business object implementation, and therefore no corresponding parent data object, and this restriction does not apply.)
- You can have attributes overridden in the child business object, and not have them delegated to the data objects both at the parent and child levels. In this case, there would be no corresponding getters and setters that correspond to these attributes on the data objects, and you could provide your own implementations of these getter and setter methods.

RELATED CONCEPTS

"Chapter 6. Inheritance" on page 299

"Inheritance and overriding in helper objects" on page 300

"Inheritance and overriding in data objects" on page 303

"Abstract base class inheritance" on page 303

"Choosing an inheritance pattern for persistence" on page 304

RELATED TASKS

“Creating a child component” on page 306

Inheritance and overriding in data objects

You can inherit both interface and implementation from the parent. In the data object implementation, you can selectively map both local attributes and inherited attributes to an associated persistent object.

When you map an inherited attribute, the mapping overrides the parent’s mapping. The parent’s get and set methods are overridden. But, the mapping is essentially augmented: the child data object implementation’s get and set methods will reimplement some or all of the parent’s original mapping, and fold in its own mapping as well. It is the mapping of the CRUD methods (insert(), update(), retrieve() and del()), along with the setConnection() method mapping that actually control how much of the augmented mapping is used. In other words, the parent’s mapping will still be in effect for the parent, but will be overridden in the child.

If you map all the inherited attributes to the child’s persistent object, and you map the child data object implementation’s CRUD methods along with the setConnection() method only to the child’s persistent object, you are using the attributes duplication pattern of inheritance.

If you map only the parent’s key attributes to the child’s persistent object, and you map the child data object implementation’s CRUD methods along with the setConnection() method to both the parent’s and the child’s persistent objects, you are using the key duplication pattern of inheritance.

RELATED CONCEPTS

“Chapter 6. Inheritance” on page 299

“Inheritance and overriding in helper objects” on page 300

“Inheritance and overriding in business objects” on page 301

“Abstract base class inheritance”

“Choosing an inheritance pattern for persistence” on page 304

“Inheritance with attributes duplication” on page 307

“Inheritance with key duplication” on page 322

RELATED TASKS

“Creating a child component” on page 306

Abstract base class inheritance

The abstract class is one that cannot be constructed. It is used to represent a concept, and can be used as a base class for other classes.

You cannot construct an object of an abstract class. For example, if Person is

an abstract class, and it has two classes Employee and Customer that inherit from it, you can create an Employee object in the Employee home, and a Customer object in the Customer home, but you cannot create a Person object. That is, the abstract interface cannot be instantiated.

Abstract base classes are supported by Object Builder. There are some restrictions when you use them:

- To use this pattern, define a business object interface for the abstract class, and do not provide an implementation.
- Any business object interface that derives from the abstract interface and includes an implementation, must override all the attributes, relationships, and methods that were defined on the abstract interface.

For polymorphic support of abstract classes, refer to the section Support for abstract classes in the *Component Broker Programming Guide*.

RELATED CONCEPTS

“Polymorphic homes” on page 581

“Chapter 6. Inheritance” on page 299

“Inheritance and overriding in helper objects” on page 300

“Inheritance and overriding in business objects” on page 301

“Inheritance and overriding in data objects” on page 303

“Choosing an inheritance pattern for persistence”

RELATED TASKS

“Creating a child component” on page 306

RELATED REFERENCES

Support for abstract classes (*Programming Guide*)

Choosing an inheritance pattern for persistence

There are four main patterns for inheritance with persistence. For any of these patterns to work, you must **not** be overriding attributes in the business object implementation.

Your choice of inheritance pattern is based on three concerns:

- Identity: whether parent and child have the same identity (that is, they share the same key)
- Performance tradeoffs: whether performance or space efficiency is more important.
- Form of persistence: whether the parent has data to be persisted, and where and how the parent’s and child’s data is persisted.

Note: It is the mapping or lack thereof of the CRUD methods (insert(), update(), retrieve() and del()) along with the setConnection() method, and not the attributes mappings that defines and distinguishes the different mapping patterns:

- Attributes duplication pattern: the child's CRUD methods and setConnection() are mapped only to its own persistent object
- Key duplication pattern: the child's CRUD methods and setConnection() are mapped to both its own persistent object, and that of its parent
- Single table pattern: the child's CRUD methods and setConnection() are mapped only to its parent's persistent object. In this pattern, however, there is a possibility that you can while mapping, override more than just the key attributes of the parent (which is valid but wasteful), or you can override only a subset of the key attributes of the parent (which is invalid for this pattern).

The attributes duplication pattern

If the parent and child have different keys, you should probably use the attributes duplication pattern. This means that the child's datastore (table) provides persistence for all of its data, including inherited data. That is, the parent's attributes (except for state data) are duplicated in the child's table. The parent's table only provides persistence for instances of the parent, never for instances of the child. If you do not use the attributes duplication pattern when there are different keys, the parent's table will have two primary keys: the parent's key for the parent's data, and the child's key for the child's inherited data. It then becomes problematic to determine which data belongs to which object type.

The key duplication pattern and the single table with views pattern

If the parent and child have the same key, you can choose between the key duplication pattern and the single table with views pattern. The key duplication pattern will generally be more efficient in its use of space (because the persistent objects for each component contain only the data required for that component, and only the parent's key is duplicated in the child), and the views pattern will generally provide faster look-up time (because both local and inherited data are mapped to the same underlying table). The views pattern is based on views of the underlying database table, and requires that there be some unique attribute of the child that can be used to select appropriate views of the database.

If the parent and child have the same key, and the parent never actually exists on its own (for example, there are never any pure Person instances kept in the table), you can use the single datastore pattern instead of the views pattern. Views are only required to select out the different object types being stored, and if the table only provides persistence for child and inherited attributes, the views are unnecessary.

Recommendation: The inheritance with views pattern provides a way to discriminate rows based on managed object type. However, polymorphic homes in the run time along with the single table pattern provides better function. It is recommended that you use polymorphic homes instead of the views pattern.

Polymorphic homes and the single table pattern

If you use polymorphic homes, you can avail of the single table pattern only if the parent and child have the same primary key (as for the key duplication pattern).

RELATED CONCEPTS

- “Chapter 6. Inheritance” on page 299
- “Inheritance with attributes duplication” on page 307
- “Inheritance with key duplication” on page 322
- “Inheritance with a single datastore” on page 341
- “Inheritance with views” on page 357

RELATED TASKS

- “Creating a child component”
- “Defining a child with attributes duplication” on page 309
- “Defining a child with key duplication” on page 325
- “Defining a child with a single datastore” on page 342
- “Defining a child with views” on page 360

Creating a child component

You can create a child component in any of the following ways. Each task is illustrated by a tutorial, with an accompanying sample:

- “Defining a child with attributes duplication” on page 309
 - “Tutorial: Inheritance with attributes duplication” on page 310
- “Defining a child with key duplication” on page 325
 - “Tutorial: Inheritance with key duplication” on page 327
- “Defining a child with a single datastore” on page 342
 - “Tutorial: Inheritance with a single datastore” on page 344
- “Defining a child with views” on page 360
 - “Tutorial: Inheritance with views” on page 362

All of these patterns assume that you are **not** overriding attributes or relationships in the business object implementation.

These patterns differ primarily in the way the data object maps to the persistent object (in other words, the way that the object hierarchy is mapped to the datastore). The general tasks involved in that step are as follows:

- “Mapping a data object to the parent’s persistent object” on page 711
- “Mapping a data object to the child’s persistent object” on page 712

Once you have created the child component, you can build and package it:

1. “Building a child component” on page 380
2. “Packaging a child component” on page 381

RELATED CONCEPTS

“Chapter 6. Inheritance” on page 299

“Inheritance and overriding in business objects” on page 301

“Choosing an inheritance pattern for persistence” on page 304

Components (*Programming Guide*)

RELATED TASKS

“Developing in Object Builder” on page 19

RELATED REFERENCES

“Naming objects” on page 128

“Internationalization of data” on page 132

Inheritance with attributes duplication



If you have or want completely separate datastores for pure parent objects and parent objects that are also child objects, you can duplicate the attributes of the parent in the child’s datastore. For example, data for a Person who is not a Beneficiary is stored in the Person datastore, and data for a Person who is a Beneficiary is stored in the Beneficiary datastore.

You can duplicate the parent’s attributes in the child’s datastore when you create the persistent object and schema from the data object. By mapping the parent’s attributes to the child’s persistent object, you implicitly override the parent’s mapping. In other words, the parent’s mapping will still be in effect for the parent, but will be overridden in the child.

For example, if Person has a child Beneficiary, then Person has a datastore that holds Person’s attributes, and Beneficiary has a datastore that holds the total of Person’s attributes and Beneficiary’s attributes.

Advantages

The potential advantage to this approach is that you have a separate datastore for each type of object, regardless of its inheritance relationships. If it is important to maintain Person and Beneficiary in different datastores (for

example, in different tables, different databases, or through different PA beans), then this approach can support that distinction, while still providing a unified object-oriented interface to the data.

This approach also allows the parent and child to use different keys to access their data, so the child does not have to use the parent's key.

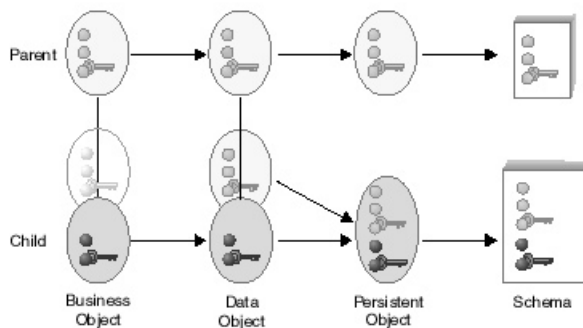
Also, you will never accidentally retrieve the wrong type of row into the managed object. For example, you will not retrieve a pure Person row when you are attempting to find an object from the Beneficiary home.

In this pattern, the parent and child tables are uncoupled and optimal for their respective business object types. Provided you do not map the child data object implementation's insert(), retrieve(), update(), and delete() methods to the parent data object implementation's persistent object (something you *must* not do), this pattern is fairly space efficient.

Disadvantages

The biggest disadvantage of this pattern is that a query designed to locate an object without foreknowledge of its type will require a sequence of distinct queries, one per data object implementation and table in the hierarchy, until a match is found. This is not an optimal representation in the relational database space.

Attributes Duplication Pattern



In this pattern:

- The parent's data object attributes and special framework methods are mapped to the parent's persistent object.
- The child's data object attributes, inherited attributes, and special framework methods are mapped to the child's persistent object.

RELATED CONCEPTS

"Chapter 6. Inheritance" on page 299

"Choosing an inheritance pattern for persistence" on page 304

RELATED TASKS

“Creating a child component” on page 306

“Tutorial: Inheritance with attributes duplication” on page 310

Defining a child with attributes duplication

This task covers the main steps necessary to create a component that inherits from another component already defined in Object Builder, and provides its own duplicated persistence for any inherited attributes. It does not cover every step; you should first be familiar with the tasks necessary to create a component without inheritance.

To create a child component in Object Builder, follow these steps:

1. Create the business object file.
2. Add the business object interface, and select the parent’s business object interface on the Interface Inheritance page.
3. If the child’s identity differs from the parent’s identity (in other words, it defines its own key attributes), you can add a key for the child. You can include attributes of the parent’s key either by selecting specific attributes on the Name and Key Attributes page, or include all the parent’s attributes by selecting the parent key on the Implementation Inheritance page. Do **not** do both.

If the child has the same key attributes as the parent, then you do not need to create a key for the child. You can simply re-use the parent’s key.

4. Add the copy helper. You can include attributes of the parent’s copy helper either by selecting specific attributes on the Name and Attributes page, or include all the parent’s attributes by selecting the parent copy helper on the Implementation Inheritance page. Do **not** do both.
5. Add the business object implementation:
 - a. Under Data Object Interface, click **Add or select one later**. This allows you to add the data object interface in a separate step, and define its parent.
 - b. Select the parent’s business object implementation on the Implementation Inheritance page.
 - c. Do **not** override any attributes on the Attributes to Override page.
 - d. Do **not** override any relationships on the Relationships to Override page.
 - e. Select any methods you want to override on the Methods to Override page.
6. Add the managed object, and select the parent’s managed object on the Implementation Inheritance page.
7. Add the data object interface:

- a. From the business object implementation's pop-up menu, click **Add New Data Object Interface**.
 - b. Select the attributes and methods of the business object you want represented in the data object.
 - c. You should select the parent data object interface on the Interface Inheritance page.
8. Add the data object implementation, and select the parent data object implementation on the Implementation Inheritance page.
 9. Add a persistent object and schema:
 - a. On the Attributes Mapping page, click **Attributes Duplication** (the horizontal partitioning pattern) to map the child's attributes and inherited attributes to the child's persistent object.
 - b. On the Methods Mapping page, map the special framework methods to the child's persistent object.

RELATED CONCEPTS

"Chapter 6. Inheritance" on page 299

"Choosing an inheritance pattern for persistence" on page 304

"Inheritance and overriding in helper objects" on page 300

Components (*Programming Guide*)

RELATED TASKS

"Creating a child component" on page 306

"Building a child component" on page 380

"Tutorial: Inheritance with attributes duplication"

Tutorial: Inheritance with attributes duplication

Objectives

To export a component from an existing sample

To import the component into a new project

To create a child component with DB persistence for its attributes and for its inherited attributes (duplicating the persistence provided by its parent)



To generate the code for the component

To build the DLLs for the component

Before you begin

You need the following installed on your system:

- A CB Server
- A CB Client
- CB tools, including Samples
- DB2 Universal Database
- DB2 SDK

-  VisualAge for C++ and (for Java applications) VisualAge for Java
-  IBM C and C++ Compilers for AIX V3.6.4 and (for Java applications) VisualAge for Java

For a complete list of prerequisites that you must install on your system, see the *Getting Started with Component Broker* documentation for the appropriate platform.

You should be familiar with the Component Broker programming model, as described in the *IBM Component Broker Programming Guide*. At minimum, you should have read chapter 1, the introduction to the programming model.

You should be familiar with the different models for inheritance and persistence, as described in “Choosing an inheritance pattern for persistence” on page 304.

You should be familiar with the attributes duplication pattern for inheritance and persistence, as described in “Inheritance with attributes duplication” on page 307.

Sample files

There is an equivalent sample for this exercise. The sample contains the results of all the inheritance tutorials. The sample includes:

- A BusinessObjects project that contains the finished Object Builder model
- A zipped Rational Rose model that contains definitions for the business object, key, copy helper, and data object interface
- Documentation for the sample, including instructions on testing it with QuickTest

The sample files are in the following directories (under <CBroker>, the install directory):

For C++:

```
samples\Tutorial\Fundamentals\Inherit\BusinessObjects
samples\Tutorial\Fundamentals\Inherit\Docs\Inherit.html
```

No Java sample is available. However, the only difference is in the business object implementations, where you can change the implementation language to **Java** to turn the C++ sample into a Java sample.

There are some differences between the sample project and the one you create in this exercise.

- The sample contains four different versions of the Beneficiary component, demonstrating the different patterns for inheritance with persistence. This exercise results in only one version of the Beneficiary component.
- The sample contains persistent objects with short names that match the package names. This exercise uses the default names generated by Object Builder.

Description

In this exercise you define a child component that provides its own persistence for inherited attributes (duplicating the persistence provided by its parent).

After you complete this exercise, you will have a component named Beneficiary that inherits from Person, and which provides persistence in a database table both for its own attributes, and for the attributes it inherits from Person.

The following tasks do not give explicit instructions for every step, but should at least get you into the right wizards. If you are experiencing problems, click the Help button within a wizard, or go to the Help pulldown in Object Builder.

For this exercise, you will complete the following tasks:

1. Exporting the parent component from the sample
2. Creating the new project
3. Importing the parent component
4. Creating a business object interface
5. Adding a key and copy helper
6. Adding a business object implementation
7. Adding a data object implementation
8. Adding a persistent object and schema
9. Adding a managed object
10. Generating the code
11. Configuring the database
12. Defining a client DLL and server DLL

Exporting the parent component

Open the existing sample:

1. Start Object Builder.
2. In the Open Project wizard, type the path for the inheritance sample model
(<CBroker>\samples\Tutorial\Fundamentals\Inherit\BusinessObjects)
3. Click **Finish**.

Export the Person component objects:

1. Under the User-Defined Business Objects folder, locate the PersonFile business object file.
2. From the pop-up menu of PersonFile, click **Export**. The Export XML wizard opens to the XML Export Directory page.
3. Click **Finish**.
udbo.PersonFile.xml is exported to the specified directory. The XML file defines the Person business object, copy helper, key, and managed object (Person, PersonBO, PersonKey, PersonCopy, PersonMO).
4. Locate the PersonFileDO data object file in the User-Defined Data Objects folder, and export it in the same way.
uddo.PersonFileDO.xml is exported to the specified directory. The XML file defines the Person data object (PersonDO and PersonDOImpl).
5. Locate the PersonFileGroup schema group in the DBA-Defined Schemas folder, and export it in the same way.
uddbschema.PersonFileGroup.xml is exported to the specified directory. The XML file defines the PersonFileGroup schema group, the schema it contains, and the schema's associated persistent object (PersonFileGroup, CBSampDB.P_PFINH, PPOPFINH).
6. Close Object Builder.

You are ready to create a new project, and import the XML files.

Creating the project

Create a sample project to hold your work. For example, e:\tutorials\adinher

1. Start Object Builder.
2. In the Open Project wizard, type a name and path for the project directory.
3. Click **Finish**.
4. When asked whether you want to create a new project, click **Yes**.

Import the parent XML files

Import the definition of the Person component:

1. Click **File > Import Model** to open the Import Model wizard to the XML File Selection page.
2. Select udbo.PersonFile.xml:
 - a. Click **Add Another**.
 - b. Type the path and name for the file, or click **Browse** to locate and select the file.
 - c. Click **Refresh**.
3. Select uddo.PersonFileDO.xml and uddbschema.PersonFileGroup.xml in the same way.
4. Click **Finish**.

The component objects for Person appear in the Tasks and Objects pane. You are ready to define a child component that inherits from Person.

Creating the business object interface

Define a business object file (AttributeDupFile):

1. From the User-Defined Business Objects folder's pop-up menu, click **Add File** to open the Business Object File wizard.
2. Name the file.
3. Click **Finish**. The file now appears under the folder.

Add a module (AttributeDupModule):

1. From the pop-up menu of the file, click **Add Module** to open the Business Object Module wizard.
2. Name the module.
3. Click **Finish**. The module now appears under the file.

Add an interface (Beneficiary) to the AttributeDupModule.

1. From the module's pop-up menu, click **Add Interface** to open the Business Object Interface wizard.
2. Name the interface Beneficiary.
3. Click the page title and turn to the Interface Inheritance page.
4. Add Person as a parent (replacing the default inheritance).
5. Click the page title and turn to the Attributes page.
6. Add the following attributes:
 - readonly long id
 - float claimPayments
7. Click **Finish**. The interface now appears under the module.

Adding the key and copy helper

Add BeneficiaryKey:

1. From the interface's pop-up menu, click **Add Key** to open the Key wizard.
2. Select id as a key attribute.
3. Add ssNo and name as key attributes (so Beneficiary's identity includes the key attributes for its parent).
Beneficiary's key now consists of the following:
 - long id (defined in Beneficiary)
 - string ssNo (defined in Person)
 - string name (defined in Person)
4. Click **Finish**. The key now appears under the interface.

Add BeneficiaryCopy:

1. From the interface's pop-up menu, click **Add Copy Helper** to open the Copy Helper wizard.
2. Add all attributes to the copy helper.
3. Click **Finish**. The copy helper now appears under the interface.

Adding the business object implementation

Add BeneficiaryBO:

1. From the interface's pop-up menu, click **Add Implementation** to open the Business Object Implementation wizard.
2. Set **Data Object Interface > Add or select one later** (you will create a new data object as a separate step).
3. Click the page title and turn to the Implementation Inheritance page.
4. Add PersonBO as a parent.
5. Click the page title and turn to the Key and Copy Helper page.
6. Select BeneficiaryKey and BeneficiaryCopy.
7. Click **Finish**. The business object implementation appears under the business object interface.

Adding the data object interface

Add BeneficiaryDO:

1. From the business object implementation's pop-up menu, click **Add New Data Object Interface** to open the Data Object Interface wizard.
2. Select all the business object attributes as state data (to be preserved in the data object).
3. Click the page title and turn to the Interface Inheritance page.
4. Add PersonDO as a parent.
5. Click **Finish**. The data object interface appears under the business object implementation.

Adding the data object implementation

Add BeneficiaryDOImpl:

1. From the data object interface's pop-up menu, click **Add Implementation** to open the Data Object Implementation wizard.
2. Click **Next** and turn to the Behavior page.
3. Set the following patterns:
 - **Environment - BOIM with any key**
 - **Type of Persistence - Embedded SQL**
 - **Data Access Pattern - Delegating**
4. Click **Next** to turn to the Implementation Inheritance page.
5. Add PersonDOImpl as a parent.
6. Click the page title and turn to the Key and Copy Helper page.

7. Select **BeneficiaryKey** and **BeneficiaryCopy**.
8. Click **Finish**. The data object implementation appears under the data object interface.

Adding the persistent object and schema

Add **BeneficiaryPO** and its associated schema:

1. From the data object implementation's pop-up menu, click **Add Persistent Object and Schema** to open the Add Persistent Object and Schema wizard.
2. Type a name for the schema group (**AttributeDupGroup**).
3. Type a name for the database (for example, **CBSampDB**).
4. Type a name for the table (**B_ADINH**)
5. Type a name for the schema file (**Beneficiary_ADINH**)
6. Click **Next** to turn to the Attributes Mapping page. Both **Beneficiary's** attributes and **Person's** attributes are displayed.
7. Click **Horizontal** (the attributes duplication pattern). This maps all attributes (both **Beneficiary's** and **Person's**) to attributes of **BeneficiaryPO**.

In this step, two things are happening:

- Because **BeneficiaryPO** does not actually exist yet, this step defines what attributes **BeneficiaryPO** contains.
 - By mapping **Person's** attributes to **BeneficiaryPO**, you are implicitly overriding the mapping in the **Person** component. **BeneficiaryDOImpl** now has its own copy of **Person's** attributes, which map to **BeneficiaryPO** instead of **PersonPO**. Also, the **create**, **retrieve**, **update**, **delete**, and **setConnection** methods are automatically mapped to the **BeneficiaryPO** but not the **PersonPO**. Examine the method mappings page to see this. (This is why **BeneficiaryMO** assemblies do not get persisted to the **Person** table.)
8. Click **Finish**. The persistent object and schema appear under the data object implementation.

Adding the managed object

Add **BeneficiaryMO**:

1. From the business object implementation's pop-up menu, click **Add Managed Object** to open the Managed Object wizard to the Name and Service page.
2. Accept all the defaults and click **Finish**.
3. Click **File > Save** to save your work before continuing to the next step.

Generating the code

You are now ready to generate the code for the objects you have defined.

Generated code will appear in your project's `\Working\platform` directory (for example, `e:\tutorials\inhadup\Working\NT`).

1. From the pop-up menu of the User-Defined Business Objects folder, click **Generate > All**.

The code generation can take several minutes.

2. Review the contents of the Working*platform* directory. All the source files for the component have been generated, and you can now define how to build them.

While you can view the generated source files for a particular object (by selecting **View Source** from its pop-up menu), you cannot edit the source files through Object Builder. If you edit them outside of Object Builder, you should restrict your changes to method implementations, and import your changes back into Object Builder before re-generating.

Configuring the database

You need to define (in DB2) the CBSampDB database and its tables (Person and Beneficiary) that your component will access. You should have a database administrator perform this procedure.

To configure the database and table, you need to enter the following commands from a DB2 command prompt.

```
create database CBSampDB
```

```
connect to CBSampDB
```

```
db2 -t -f P_PFINH.sql
```

```
db2 -t -f B_KDINH.sql
```

If you look inside the SQL files, you can see that the table for Beneficiary duplicates the columns defined in Person's table, as the name of the inheritance pattern (attributes duplication) suggests. Person's table definition is a subset of Beneficiary's table definition. Person's table will be used to persist Person components, and Beneficiary's table will be used to persist Beneficiary components.

Defining a client DLL and server DLL

Configure the build process that will create the client and server DLLs.

While "DLL" is the generic term used in Object Builder, the configuration for the build actually results in whatever the appropriate targets are for the selected platforms. For example, on AIX the build process creates shared library files (lib*.so). If you chose to create a Java business object, then in addition to DLLs there will also be JAR files. The names for these files are derived from the name you provide for the DLL file, within the DLL configuration node.

Start by defining the client DLL configuration and client DLL or library file (for this exercise, name them both clientIN). The client DLL provides client applications with access to the component on the server, using the key and copy helper. You must also include the business object interface, which defines the methods and attributes of the component that the client can access.

1. From the pop-up menu of the Build Configuration folder, click **Add Client DLL** to open the Client DLL wizard to the Name and Options page.
2. Name the configuration. This is the name that uniquely identifies the configuration node.
3. Name the library as well. This is the name for the makefile and for the resulting DLL file. You can create multiple configuration nodes that produce different versions of the same DLL. For example, you could set different compile and link options to produce a development version and a deployment version of the same DLL.
4. For this exercise, the default configuration options are sufficient.
5. Click the title and turn to the Client Source Files page.
6. Select all the client files and add them to the **Items Chosen** list.
7. Click **Finish**.

The clientIN DLL configuration appears under the Build Configuration folder.

Define the server DLL configuration and server DLL file (for this exercise, name them both serverIN). The server DLL is installed on the server to deploy the component, making it available for access by client applications and other components.

1. From the pop-up menu of the Build Configuration folder, click **Add Server DLL** to open the Server DLL wizard.
2. Name the configuration and target DLL file.
3. Click **Next** to turn to the Libraries to Link With page.
4. Add the client DLL file. The server DLL needs access to the client DLL because the client DLL defines the component's interface, and the component implementation inherits from it.
5. Click **Next** to turn to the Server Source Files page.
6. Select all the server source files for the component, and add them to the **Items Chosen** list.
7. Click **Finish**.

The serverIN DLL configuration appears under the Build Configuration folder.

Building the DLLs

To generate the makefiles, based on the configuration choices you made in the DLL wizards:

1. From the pop-up menu of the Build Configuration folder, click **Generate > All > All Targets**. This generates makefiles for all the DLL files defined in the folder and generates an all.mak file that calls the individual DLL makefiles. By choosing **All Targets**, you set the default behavior of the makefile, when it is built. You can still override this default when you build (for example, you can choose to build Java targets only, and override the default of all targets).

To build the DLL and (optionally) JAR files:


1. From the Build Configuration folder's pop-up menu, select **Build > Out-of-Date Targets > C++**.


You always need to generate C++ targets, even if you selected **Java** as your business object implementation language. The other objects of the component (such as the data object implementation and managed object) are still implemented in C++.

2. If you selected **Java** as your business object implementation language, then you also need to build the Java targets. From the Build Configuration folder's pop-up menu, select **Build > Out-of-Date Targets > Java**.

When you have a mixed-language application (some business objects are implemented in C++, some in Java) you must generate both C++ and Java targets for *all* components, even for those implemented in C++. The .jar files for C++ components allow Java components to interact with the C++ components on the server.

For this exercise, your application contains just a single component, so you only need to build the Java targets if you implemented the business object in Java.

 The clientIN.dll and serverIN.dll files are stored in the working\NT\PRODUCTION directory.

 The libclientIN.so and libserverIN.so files are stored in the working/AIX/PRODUCTION directory.

If you have a Java business object, the clientIN.jar and serverIN.jar files are stored in the working\platform\PRODUCTION directory.

If you had a Java client application, regardless of the language your component is implemented in, you would also select **Build > Out-of-Date Targets > Java Client Bindings** to generate Java client bindings (working\platform\<build style>\JCB\JCBclientIN.jar). <build style> is one of the configuration directories: NOOPT, PRODUCTION, TRACE, TRACE_DEBUG. See Platform-specific information for more information.

For this exercise, you will be using QuickTest to run and test your application, which has its own build process that includes generation of client bindings.



Defining an application family and application

Define the application family (DataPersistenceApplication). An application family groups a set of applications within System Management.

1. From the pop-up menu of the Application Configuration folder, click **Add Application Family** to open the Application Family wizard.
2. Name the application family.
3. Click **Finish**.

The DataPersistenceApplication application family appears under the Application Configuration folder.

Define the server application (DataPersistenceObjects). An application defines a set of components that will operate together on the server. The application name you provide here is the name that will be used by System Management, when the application and its components are deployed.

1. From the pop-up menu of the application family, click **Add Application** to open the Application wizard.
2. Name the application.
3. Click **Next** to turn to the Additional Executables page.
4. Select the platform you are configuring the application for (for example, **NT Files**).
5. Add the file Person.sql:
 - a. Click **Add Another**.
 - b. Click the **Browse** button to open the Executables to Include dialog.
 - c. Locate your Object Builder working directory.
 - d. Select Person_PFINH.sql
 - e.  Click the **Open** button.
 - f.  Click the **OK** button.
6. Add the files Person_PFINH.bnd, Beneficiary_ADINH.sql, and Beneficiary_ADINH.bnd in the same manner.
7. Click **Finish**.

The DataPersistenceObjects application appears under the DataPersistenceApplication application family.

Configuring the components with the application

Configure Person's managed object (PersonFile PersonModule::PersonMO)

with the application (DataPersistenceObjects). Create a new container instance (DataPersistenceContainer) that will **Throw an exception** when a method is called outside a transaction.

1. From the pop-up menu of the application, click **Add Managed Object** to open the Managed Object Configuration wizard.
2. In the **Managed Objectlist**, select the component's managed object. The other lists should be automatically populated with the appropriate selections.
3. Click **Next** and select the data object implementation. Click **Add Another**. The appropriate data object implementation and DLL will be selected. If there were more than one data object implementation for the component, or it was being built into more than one DLL, you would click **Add Another** again to add the extra information.
4. Click **Next** to turn to the Container page.
5. Check the **Create a new container** option. An extra page is inserted into the wizard, to allow you to name the new container and specify its policies.
6. Click **Next** to turn to the Create New Container page.
7. Name the container, and set its behavior for methods called outside a transaction.

Most of the container's behavior is defined for you, based on the managed object you are configuring, and the data object implementation you selected.

8. Click **Next** and review the home. The appropriate home is already selected, and the default options are acceptable. If there were more than one appropriate home, you could choose the home to use.
9. Click **Finish**.

The PersonMO managed object configuration appears under the DataPersistenceObjects application, and the new container DataPersistenceContainer appears in the Container Definition folder. You can review the properties of the container, including the service and data access patterns that have been selected for you, by clicking **Properties** from the container's pop-up menu.

Configure Beneficiary's managed object in the same way:

1. From the pop-up menu of the application, click **Configure Managed Object** to open the Managed Object Configuration wizard.
2. In the **Managed Objectlist**, select AttributeDupFile AttributeDupModule::BeneficiaryMO.
3. Click the page title and turn to the Container page.

4. Select `DataPersistenceContainer`. Because `Person` and `Beneficiary` are similar components (with the same type of datastore and the same policies), they can use the same container.
5. Click **Finish**.

Generate the application installation information:

1. From the pop-up menu of the `Application Configuration` folder, click **Generate**.

The DDL that defines the applications for System Management is generated into `Working\platform\PRODUCTION\DataPersistenceApplication`.

Close Object Builder:

1. Click **File > Save** to save your work.
2. Click **File > Exit** to close Object Builder.

Summary

You have created a component named `Beneficiary` that inherits from `Person`, and which provides persistence in a database table both for its own attributes, and for the attributes it inherits from `Person`. You have implemented the attributes duplication pattern for inheritance with persistence.

You can find information on installing the component on the server in the *System Administration Guide*, “Configure a New Application Environment” chapter, or online in the topic *Installing and configuring a new application*.

You can find information on testing the installed component with a `QuickTest` client application in “Chapter 13. Testing applications with `QuickTest`” on page 611 .

Inheritance with key duplication

If your datastores are divided in a way that mirrors your component hierarchy, then inheritance works the same for persistent data as it does for data on the object level. In other words, a child has its own datastore for its own attributes, and uses its parent’s datastore for inherited attributes. The only exception is for key attributes: in this pattern, the child typically uses the same key as the parent, and the parent’s key is duplicated in the child’s datastore.

This is the default inheritance pattern in Object Builder. If you create new parent and child components (starting from the business object interface and working down to persistent objects and DB schemas), then each schema holds

only the definitions for data defined in its component. The child component uses its own persistent object for its own data, and its parent's persistent object for inherited data.

Advantages

The advantage to this approach is its precision, and efficient use of space. It is almost as space-efficient as the attribute duplication pattern. It is very efficient for type-constrained (non-polymorphic) queries.

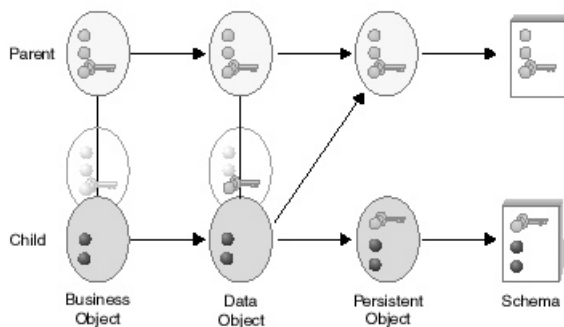
Disadvantages

Because data access can span multiple datastores, access time may be slower than with other patterns. Also, this approach is problematic if your parent and child use different keys. Because part of the child's data is stored in the parent's datastore, the parent datastore needs to support both keys (the child's and the parent's), to ensure data for the right object type is returned. Generally, you should only use this pattern when the parent and the child use the same key.

For this pattern, the parent's table and the child's table must be in the same database.

Note: This pattern is not supported for polymorphic homes.

Key Duplication Pattern



In this pattern:

- The parent's data object attributes and special framework methods are mapped to the parent's persistent object.
- The child's data object attributes, and its inherited key, are mapped to the child's persistent object. This creates a duplicate of the parent's key in the child's persistent object, which allows it to locate the parent's persistent object when it needs to retrieve inherited attributes.

- The child's data object special framework methods are mapped in one of two ways, depending on the type of creation and deletion scenarios you want to support.

Mapping for separate creation and deletion

If you want to support creation of a child with an existing parent entry, and deletion of a child without deletion of its parent entry, map as follows:

- insert and update map first to the parent's, and then to the child's persistent objects, with the **Always complete calling sequence** check box selected. (For example, insert() maps to iPersonPO.insert() and iBeneficiaryPO.insert().)

Because they map to both, and the calling sequence ignores errors, you can successfully create a Beneficiary that already exists as a Person: the parent insert fails, but still proceeds to the child insert(), which is successful. You will not be able to set values for the attributes of an existing parent during creation of the child. If you create the child using a copy helper, any values you set for inherited attributes are ignored, because they are applied to the parent's existing records using insert() instead of update(). You can change the inherited attributes in a separate update call after you create the child.
- retrieve and setConnection map to first the child's and then the parent's persistent objects, with the **Always complete calling sequence** check box not selected. (For example, retrieve() maps to iBeneficiaryPO.retrieve() and iPersonPO.retrieve().)

Because Beneficiary stores its inherited attributes in Person's datastore, it must be able to retrieve the parent's data. If an error occurs on the parent's retrieve(), it abandons the calling sequence and returns an error.
- delete() maps to the child's persistent object. (For example, iBeneficiaryPO.delete().)

Because the delete() method maps only to the child's persistent object, when a child object is deleted, its record as a parent object remains. (For example, when you delete a Beneficiary, you retain an entry for the Person, even though the Person is no longer a Beneficiary.)

Mapping for unified creation and deletion

You create only new parents and children, and delete the child and its parent in the same step, map as follows:

- insert and update map first to the parent's, and then to the child's persistent objects, with the **Always complete calling sequence** check box **not** selected. (For example, insert() maps to iPersonPO.insert() and iBeneficiaryPO.insert().)

This always creates a new parent along with the child.

If you wanted to create a new child from an existing parent, you could still find the existing parent, create a copy of its attribute values, delete the parent, and then create the child as a new object with the values of the deleted parent.

- `retrieve()` and `setConnection()` map first to the child's, and then to the parent's persistent objects, with the **Always complete calling sequence** check box **not** selected. (For example, `retrieve()` maps to `iBeneficiaryPO.retrieve()` and `iPersonPO.retrieve()`.)

Because Beneficiary stores its inherited attributes in Person's datastore, it **must** be able to retrieve the parent's data. If an error occurs on the parent's retrieve, it abandons the calling sequence and returns an error.

- `delete` maps to first the child's and then the parent's persistent objects, with the **Always complete calling sequence** check box **not** selected. (For example, `iBeneficiaryPO.delete` and `iPersonPO.delete`.)

This deletes the parent along with the child.

If you wanted to delete the child and leave the parent entry, you could still copy the existing parent values, continue with the deletion of the child, and then recreate the parent with the copied values.

RELATED CONCEPTS

"Chapter 6. Inheritance" on page 299

"Choosing an inheritance pattern for persistence" on page 304

RELATED TASKS

"Creating a child component" on page 306

"Defining a child with key duplication"

"Tutorial: Inheritance with key duplication" on page 327

Defining a child with key duplication

This task covers the main steps necessary to create a component that inherits from another component already defined in Object Builder, and duplicates its parent's key in the child's datastore, so it can look up its parent and use its parent's persistence for other inherited attributes. It does not cover every step; you should first be familiar with the tasks necessary to create a component without inheritance.

To use the key duplication pattern, the child must have the same key attributes as the parent. If the child has a different key, use the attributes duplication pattern. Also, if persistence is provided in a database, both the parent and child must use tables in the same database.

To create a child component in Object Builder, follow these steps:

1. Create the business object file.

2. Add the business object interface, and select the parent's business object interface on the Interface Inheritance page.
3. Add the copy helper. You can include attributes of the parent's copy helper either by selecting specific attributes on the Name and Attributes page, or include all the parent's attributes by selecting the parent copy helper on the Implementation Inheritance page. Do **not** do both.
4. Add the business object implementation:
 - a. Under Data Object Interface, click **Add or select one later**. This allows you to add the data object interface in a separate step, and define its parent.
 - b. Select the parent's business object implementation on the Implementation Inheritance page.
 - c. Select the parent's key on the Key and Copy Helper page.
 - d. Do **not** override any attributes on the Attributes to Override page.
 - e. Do **not** override any relationships on the Relationships to Override page.
 - f. Select any methods you want to override on the Methods to Override page.
5. Add the managed object, and select the parent's managed object on the Implementation Inheritance page.
6. Add the data object interface:
 - a. From the business object implementation's pop-up menu, click **Add New Data Object Interface**.
 - b. Select the attributes and methods of the business object you want represented in the data object.
 - c. You should select the parent data object interface on the Interface Inheritance page.
7. Add the data object implementation, and select the parent data object implementation on the Implementation Inheritance page.
8. Add a persistent object and schema, and on the Attributes Mapping page, click **Key Duplication** to map the child's data object attributes, and its inherited key, to the child's persistent object.
9. Open the data object implementation's properties, and map the special framework methods as follows:
 - On the Methods Mapping page:
 - delete maps to the child's persistent object. (For example, iBeneficiary.delete.)
 - insert and update map to first the parent's and then the child's persistent objects, with the **Always complete calling sequence** option checked. (For example, insert maps to iPersonPO.insert and iBeneficiaryPO.insert.)

- retrieve and setConnection map to first the child’s and then the parent’s persistent objects, with the **Always complete calling sequence** option **not** checked. (For example, retrieve maps to iBeneficiaryPO.retrieve and iPersonPO.retrieve.)

These mappings support creation of a child when its entry as a parent already exists (for example, creation of a Beneficiary when a Person with the same key value already exists). If you wanted to restrict creation to entirely new objects, you could uncheck the **Always complete calling sequence** option on the insert and update mappings. This would mean that new children are always created with new parents.

These mappings also support deletion of the child without deletion of its parent, leaving the parent entry behind (for example, deletion of a Beneficiary does not affect its Person values). If you wanted to have deletion remove the parent along with the child, you could map the delete method to the parent’s persistent object as well.

RELATED CONCEPTS

“Chapter 6. Inheritance” on page 299

“Choosing an inheritance pattern for persistence” on page 304

“Inheritance with key duplication” on page 322

Components (*Programming Guide*)

RELATED TASKS

“Creating a child component” on page 306

“Building a child component” on page 380

“Tutorial: Inheritance with key duplication”

Tutorial: Inheritance with key duplication

Objectives

To export a component from an existing sample

To import the component into a new project

To create a child component that uses its parent’s persistence for inherited attributes



To generate the code for the component

To build the DLLs for the component

Before you begin

You need the following installed on your system:

- A CB Server
- A CB Client
- CB tools, including Samples

- DB2 Universal Database
- DB2 SDK
-  VisualAge for C++ and (for Java applications) VisualAge for Java
-  IBM C and C++ Compilers for AIX V3.6.4 and (for Java applications) VisualAge for Java

For a complete list of prerequisites that you must install on your system, see the *Getting Started with Component Broker* documentation for the appropriate platform.

You should be familiar with the Component Broker programming model, as described in the *IBM Component Broker Programming Guide*. At minimum, you should have read chapter 1, the introduction to the programming model.

You should be familiar with the different models for inheritance and persistence, as described in “Choosing an inheritance pattern for persistence” on page 304.

You should be familiar with the key duplication pattern for inheritance and persistence, as described in “Inheritance with key duplication” on page 322.

Sample files

There is an equivalent sample for this exercise. The sample contains the results of all the inheritance tutorials. The sample includes:

- A BusinessObjects project that contains the finished Object Builder model
- A zipped Rational Rose model that contains definitions for the business object, key, copy helper, and data object interface
- Documentation for the sample, including instructions on testing it with QuickTest

The sample files are in the following directories (under <CBroker>, the install directory):

For C++:

```
samples\Tutorial\Fundamentals\Inherit\BusinessObjects
samples\Tutorial\Fundamentals\Inherit\Docs\Inherit.html
```

No Java sample is available. However, the only difference is in the business object implementations, where you can change the implementation language to **Java** to turn the C++ sample into a Java sample.

There are some differences between the sample project and the one you create in this exercise.

- The sample contains four different versions of the Beneficiary component, demonstrating the different patterns for inheritance with persistence. This exercise results in only one version of the Beneficiary component.
- The sample contains persistent objects with short names that match the package names. This exercise uses the default names generated by Object Builder.

Description

In this exercise you define a child component that uses its parent's persistence for inherited attributes (so that each component in the object hierarchy provides persistence for its own attributes, plus its parent's key, which is used to look up the parent and find the value of inherited attributes). This inheritance pattern makes the most sense when parent and child share the same key. For a scenario where parent and child have different keys, see "Tutorial: Inheritance with attributes duplication" on page 310.

After you complete this tutorial, you will have a component named Beneficiary that inherits from Person, and provides persistence in a database table for its own attributes (plus Person's key), and uses Person's database table to access inherited attributes. For example, a query on Beneficiary.name (an attribute inherited from Person) results in a lookup on Beneficiary's table to find the parent Person's key (which is duplicated in Beneficiary's table), and then a lookup on Person's table to find the value of the name attribute.

Note: For this pattern, the parent's table and the child's table must be in the same database.

The following tasks do not give explicit instructions for every step, but should at least get you into the right wizards. If you are experiencing problems, click the Help button within a wizard, or go to the Help pulldown in Object Builder.

For this exercise, you will complete the following tasks:

1. Exporting the parent component from the sample
2. Creating the new project
3. Importing the parent component
4. Creating a business object interface
5. Adding a key and copy helper
6. Adding a business object implementation
7. Adding a data object implementation
8. Adding a persistent object and schema
9. Adding a managed object
10. Generating the code

11. Configuring the database
12. Defining a client DLL and server DLL

Exporting the parent component

Open the existing sample:

1. Start Object Builder.
2. In the Open Project wizard, type the path for the inheritance sample model
(<CBroker>\samples\Tutorial\Fundamentals\Inherit\BusinessObjects)
3. Click **Finish**.

Export the Person component objects:

1. Under the User-Defined Business Objects folder, locate the PersonFile business object file.
2. From the pop-up menu of PersonFile, click **Export**. The Export XML wizard opens to the XML Export Directory page.
3. Click **Finish**.
udbo.PersonFile.xml is exported to the specified directory. The XML file defines the Person business object, copy helper, key, and managed object (Person, PersonBO, PersonKey, PersonCopy, PersonMO).
4. Locate the PersonFileDO data object file in the User-Defined Data Objects folder, and export it in the same way.
uddo.PersonFileDO.xml is exported to the specified directory. The XML file defines the Person data object (PersonDO and PersonDOImpl).
5. Locate the PersonFileGroup schema group in the DBA-Defined Schemas folder, and export it in the same way.
uddbschema.PersonFileGroup.xml is exported to the specified directory. The XML file defines the PersonFileGroup schema group, the schema it contains, and the schema's associated persistent object (PersonFileGroup, CBSampDB.P_PFINH, PPOPFINH).
6. Close Object Builder.

You are ready to create a new project, and import the XML files.

Creating the project

Create a sample project to hold your work. For example, e:\tutorials\inhkdup

1. Start Object Builder.
2. In the Open Project wizard, type a name and path for the project directory.
3. Click **Finish**.
4. When asked whether you want to create a new project, click **Yes**.

Import the parent XML files

Import the definition of the Person component:

1. Click **File > Import Model** to open the Import Model wizard to the XML File Selection page.
2. Select `udbo.PersonFile.xml`:
 - a. Click **Add Another**.
 - b. Type the path and name for the file, or click **Browse** to locate and select the file.
 - c. Click **Refresh**.
3. Select `uddo.PersonFileDO.xml` and `uddbschema.PersonFileGroup.xml` in the same way.
4. Click **Finish**.

The component objects for Person appear in the Tasks and Objects pane. You are ready to define a child component that inherits from Person.

Creating the business object interface

Define a business object file (`KeyDupFile`):

1. From the User-Defined Business Objects folder's pop-up menu, click **Add File** to open the Business Object File wizard.
2. Name the file.
3. Click **Finish**. The file now appears under the folder.

Add a module (`AttributeDupModule`):

1. From the pop-up menu of the file, click **Add Module** to open the Business Object Module wizard.
2. Name the module.
3. Click **Finish**. The module now appears under the file.

Add an interface (`Beneficiary`) to the `AttributeDupModule`.

1. From the module's pop-up menu, click **Add Interface** to open the Business Object Interface wizard.
2. Name the interface `Beneficiary`.
3. Click the page title and turn to the Interface Inheritance page.
4. Add `Person` as a parent (replacing the default inheritance).
5. Click the page title and turn to the Attributes page.
6. Add the following attribute:
 - `float claimPayments`
7. Click **Finish**. The interface now appears under the module.

Adding the copy helper

Because the child component inherits its key attribute from its parent, you do not need to define a separate key. The child component can re-use its parent's key helper.

You still need to define a separate copy helper for the child.

Add BeneficiaryCopy:

1. From the interface's pop-up menu, click **Add Copy Helper** to open the Copy Helper wizard.
2. Add all attributes to the copy helper (both parent's and child's).
3. Click **Finish**. The copy helper now appears under the interface.

Adding the business object implementation

Add BeneficiaryBO:

1. From the interface's pop-up menu, click **Add Implementation** to open the Business Object Implementation wizard.
2. Set **Data Object Interface > Add or select one later** (you will create a new data object as a separate step).
3. Click the page title and turn to the Implementation Inheritance page.
4. Add PersonBO as a parent.
5. Click the page title and turn to the Key and Copy Helper page.
6. Select PersonKey and BeneficiaryCopy.
7. Click **Finish**. The business object implementation appears under the business object interface.

Adding the data object interface

Add BeneficiaryDO:

1. From the business object implementation's pop-up menu, click **Add New Data Object Interface** to open the Data Object Interface wizard.
2. Select all the business object attributes as state data (to be preserved in the data object).
3. Click the page title and turn to the Interface Inheritance page.
4. Add PersonDO as a parent.
5. Click **Finish**. The data object interface appears under the business object implementation.

Adding the data object implementation

Add BeneficiaryDOImpl:

1. From the data object interface's pop-up menu, click **Add Implementation** to open the Data Object Implementation wizard.
2. Click **Next** to turn to the Behavior page.
3. Set the following patterns:
 - **Environment - BOIM with any key**
 - **Type of Persistence - Embedded SQL**
 - **Data Access Pattern - Delegating**
4. Click **Next** to turn to the Implementation Inheritance page.

5. Add PersonDOImpl as a parent.
6. Click the page title and turn to the Key and Copy Helper page.
7. Select PersonKey and BeneficiaryCopy.
8. Click **Finish**. The data object implementation appears under the data object interface.

Adding the persistent object and schema

Add BeneficiaryPO and its associated schema:

1. From the data object implementation's pop-up menu, click **Add Persistent Object and Schema** to open the Add Persistent Object and Schema wizard.
2. Type a name for the schema group (KeyDupGroup).
3. Type a name for the database (for example, CBSampDB).
4. Type a name for the table (B_KDINH)
5. Type a name for the schema file (Beneficiary_KDINH)
6. Click **Next** to turn to the Attributes Mapping page. Both Beneficiary's attributes and Person's attributes are displayed.
7. Click **Key Duplication**. This maps the child's attributes and the parent's key to the child's persistent object, creating a duplicate entry for the key in the child's persistent object.

Because Beneficiary now has a record of the parent's key, a call to Beneficiary for an inherited attribute (such as town) can be delegated to the parent table. Beneficiary receives the call, then uses the parent's key to find the right row in the parent's table, and retrieve the called attribute. Contrast this with the situation in "Tutorial: Inheritance with attributes duplication" on page 310, in which all of the parent's data is persisted in the child's table.

8. Click **Finish**. The persistent object and schema appear under the data object implementation.

Mapping the special framework methods

Map the way in which the data object's special framework methods will call the persistent objects' special framework methods:

1. From BeneficiaryDOImpl's pop-up menu, click **Properties** to open the Data Object Implementation wizard.
2. Click the page title and turn to the Methods Mapping page.

Because Beneficiary has its own data in one persistent object and inherited data in a separate persistent object, the special framework methods need to access both persistent objects in order to ensure all the right data is retrieved.

3. Map each method as follows:

- insert and update map to first iPersonPO's methods and then iBeneficiaryPO's methods, with the **Always complete calling sequence** option checked.

Because they map to both, and the calling sequence will ignore errors, you can successfully create a Beneficiary that already exists as a Person: the parent insert will fail, but still proceed to the child insert, which is successful.

You will not be able to set values for the attributes of an existing parent during creation of the child. If you create the Beneficiary using a copy helper, any values you set for inherited attributes of Person are ignored, since they are applied to Person's existing records using insert, when they need to use update. You can change the inherited attributes in a separate update call after you create the child.

- retrieve and setConnection map to first iBeneficiaryPO's methods and then iPersonPO's methods, with the **Always complete calling sequence** option **not** checked.

Because Beneficiary stores its inherited attributes in Person's datastore, it **must** be able to retrieve the parent's data. If an error occurs on the parent's retrieve, it abandons the calling sequence and returns an error.

By mapping to both the parent and the child persistent object, you allow a call to Beneficiary for its parent's data (for example, Beneficiary.town) to resolve as follows:

- a. Person's key is retrieved from iBeneficiaryPO
 - b. The appropriate Person is located, and Person.town is retrieved from iPersonPO.
- delete maps to iBeneficiaryPO.delete.

Because the delete method maps only to the child's persistent object, when a Beneficiary is deleted, its record as a Person remains. (So you retain an entry for the Person, even though the Person is no longer a Beneficiary.)

This scenario supports creation of a Beneficiary when its entry as a Person already exists.

If you wanted to restrict creation to entirely new objects, you could clear the **Always complete calling sequence** option on the insert and update mappings. This would mean that new children are always created with new parents.

This scenario also supports deletion of the Beneficiary without deletion of its parent Person, leaving the Person entry behind. If you wanted to have deletion remove the parent along with the child, you could map the delete method to the parent's persistent object as well.

Recommendation:It is recommended that only advanced users follow this

method of mapping. The normal method of mapping is to restrict creation to entirely new objects, and have deletion remove the parent along with the child.

4. Click **Finish**.

Adding the managed object

Add BeneficiaryMO:

1. From the business object implementation's pop-up menu, click **Add Managed Object** to open the Managed Object wizard to the Name and Service page.
2. Accept all the defaults and click **Finish**.
3. Click **File > Save** to save your work before continuing to the next step.

Generating the code

You are now ready to generate the code for the objects you have defined. Generated code will appear in your project's `\Working\platform` directory (for example, `e:\tutorials\inhkdup\Working\NT`).

1. From the pop-up menu of the User-Defined Business Objects folder, click **Generate > All**.

The code generation can take several minutes.

2. Review the contents of the `Working\platform` directory. All the source files for the component have been generated, and you can now define how to build them.

While you can view the generated source files for a particular object (by selecting **View Source** from its pop-up menu), you cannot edit the source files through Object Builder. If you edit them outside of Object Builder, you should restrict your changes to method implementations, and import your changes back into Object Builder before re-generating.

Configuring the database

You need to define (in DB2) the CBSampDB database and its tables (Person and Beneficiary) that your component will access. You should have a database administrator perform this procedure.

To configure the database and table, you need to enter the following commands from a DB2 command prompt.

```
create database CBSampDB
connect to CBSampDB
```

```
db2 -t -f P_PFINH.sql
```

```
db2 -t -f B_KDINH.sql
```

If you look in the listed SQL files, you can see that the table for Beneficiary duplicates the key column defined in Person's table, as the name of the inheritance pattern (key duplication) suggests. Person's table definition is otherwise different from Beneficiary's table definition, and does not repeat the definitions for any other inherited attributes. Person's table will be used to persist Person components, and will also be used to persist the inherited attributes of Beneficiary components.

Defining a client DLL and server DLL

Configure the build process that will create the client and server DLLs.

While "DLL" is the generic term used in Object Builder, the configuration for the build actually results in whatever the appropriate targets are for the selected platforms. For example, on AIX the build process creates shared library files (lib*.so). If you chose to create a Java business object, then in addition to DLLs there will also be JAR files. The names for these files are derived from the name you provide for the DLL file, within the DLL configuration node.

Start by defining the client DLL configuration and client DLL or library file (for this exercise, name them both clientIN). The client DLL provides client applications with access to the component on the server, using the key and copy helper. You must also include the business object interface, which defines the methods and attributes of the component that the client can access.

1. From the pop-up menu of the Build Configuration folder, click **Add Client DLL** to open the Client DLL wizard to the Name and Options page.
2. Name the configuration. This is the name that uniquely identifies the configuration node.
3. Name the library as well. This is the name for the makefile and for the resulting DLL file. You can create multiple configuration nodes that produce different versions of the same DLL. For example, you could set different compile and link options to produce a development version and a deployment version of the same DLL.
4. For this exercise, the default configuration options are sufficient.
5. Click the title and turn to the Client Source Files page.
6. Select all the client files and add them to the **Items Chosen** list.
7. Click **Finish**.

The clientIN DLL configuration appears under the Build Configuration folder.

Define the server DLL configuration and server DLL file (for this exercise, name them both serverIN). The server DLL is installed on the server to deploy the component, making it available for access by client applications and other components.

1. From the pop-up menu of the Build Configuration folder, click **Add Server DLL** to open the Server DLL wizard.
2. Name the configuration and target DLL file.
3. Click **Next** to turn to the Libraries to Link With page.
4. Add the client DLL file. The server DLL needs access to the client DLL because the client DLL defines the component's interface, and the component implementation inherits from it.
5. Click **Next** to turn to the Server Source Files page.
6. Select all the server source files for the component, and add them to the **Items Chosen** list.
7. Click **Finish**.

The serverIN DLL configuration appears under the Build Configuration folder.

Building the DLLs

To generate the makefiles, based on the configuration choices you made in the DLL wizards:

1. From the pop-up menu of the Build Configuration folder, click **Generate > All > All Targets**. This generates makefiles for all the DLL files defined in the folder and generates an all.mak file that calls the individual DLL makefiles. By choosing **All Targets**, you set the default behavior of the makefile, when it is built. You can still override this default when you build (for example, you can choose to build Java targets only, and override the default of all targets).

To build the DLL and (optionally) JAR files:

1. From the Build Configuration folder's pop-up menu, select **Build > Out-of-Date Targets > C++**.

You always need to generate C++ targets, even if you selected **Java** as your business object implementation language. The other objects of the component (such as the data object implementation and managed object) are still implemented in C++.

2. If you selected **Java** as your business object implementation language, then you also need to build the Java targets. From the Build Configuration folder's pop-up menu, select **Build > Out-of-Date Targets > Java**.

When you have a mixed-language application (some business objects are implemented in C++, some in Java) you must generate both C++ and Java targets for *all* components, even for those implemented in C++. The .jar files for C++ components allow Java components to interact with the C++ components on the server.

For this exercise, your application contains just a single component, so you only need to build the Java targets if you implemented the business object in Java.

► **WIN** The clientIN.dll and serverIN.dll files are stored in the working\NT\PRODUCTION directory.

► **AIX** The libclientIN.so and libserverIN.so files are stored in the working/AIX/PRODUCTION directory.

If you have a Java business object, the clientIN.jar and serverIN.jar files are stored in the working\platform\PRODUCTION directory.

If you had a Java client application, regardless of the language your component is implemented in, you would also select **Build > Out-of-Date Targets > Java Client Bindings** to generate Java client bindings (working\platform\\JCB\JCBclientIN.jar). <build style> is one of the configuration directories: NOOPT, PRODUCTION, TRACE, TRACE_DEBUG. See Platform-specific information for more information.

For this exercise, you will be using QuickTest to run and test your application, which has its own build process that includes generation of client bindings.

Defining an application family and application



Define the application family (DataPersistenceApplication). An application family groups a set of applications within System Management.

1. From the pop-up menu of the Application Configuration folder, click **Add Application Family** to open the Application Family wizard.
2. Name the application family.
3. Click **Finish**.

The DataPersistenceApplication application family appears under the Application Configuration folder.

Define the server application (DataPersistenceObjects). An application defines a set of components that will operate together on the server. The application name you provide here is the name that will be used by System Management, when the application and its components are deployed.

1. From the pop-up menu of the application family, click **Add Application** to open the Application wizard.
2. Name the application.
3. Click **Next** to turn to the Additional Executables page.
4. Select the platform you are configuring the application for (for example, **NT Files**).
5. Add the file Person.sql:
 - a. Click **Add Another**.
 - b. Click the **Browse** button to open the Executables to Include dialog.

- c. Locate your Object Builder working directory.
 - d. Select Person_PFINH.sql
 - e.  Click the **Open** button.
 - f.  Click the **OK** button.
6. Add the files Person_PFINH.bnd, Beneficiary_KDINH.sql, and Beneficiary_KDINH.bnd in the same manner.
 7. Click **Finish**.

The DataPersistenceObjects application appears under the DataPersistenceApplication application family.

Configuring the components with the application

Configure Person's managed object (PersonFile PersonModule::PersonMO) with the application (DataPersistenceObjects). Create a new container instance (DataPersistenceContainer) that will **Throw an exception** when a method is called outside a transaction.

1. From the pop-up menu of the application, click **Add Managed Object** to open the Managed Object Configuration wizard.
2. In the **Managed Objectlist**, select the component's managed object. The other lists should be automatically populated with the appropriate selections.
3. Click **Next** and select the data object implementation. Click **Add Another**. The appropriate data object implementation and DLL will be selected. If there were more than one data object implementation for the component, or it was being built into more than one DLL, you would click **Add Another** again to add the extra information.
4. Click **Next** to turn to the Container page.
5. Check the **Create a new container** option. An extra page is inserted into the wizard, to allow you to name the new container and specify its policies.
6. Click **Next** to turn to the Create New Container page.
7. Name the container, and set its behavior for methods called outside a transaction.
Most of the container's behavior is defined for you, based on the managed object you are configuring, and the data object implementation you selected.
8. Click **Next** and review the home. The appropriate home is already selected, and the default options are acceptable. If there were more than one appropriate home, you could choose the home to use.
9. Click **Finish**.

The PersonMO managed object configuration appears under the DataPersistenceObjects application, and the new container DataPersistenceContainer appears in the Container Definition folder. You can review the properties of the container, including the service and data access patterns that have been selected for you, by clicking **Properties** from the container's pop-up menu.

Configure Beneficiary's managed object in the same way:

1. From the pop-up menu of the application, click **Configure Managed Object** to open the Managed Object Configuration wizard.
2. In the **Managed Objectlist**, select KeyDupFile KeyDupModule::BeneficiaryMO.
3. Click the page title and turn to the Container page.
4. Select DataPersistenceContainer. Because Person and Beneficiary are similar components (with the same datastore and the same policies), they can use the same container.
5. Click **Finish**.

Generate the application installation information:

1. From the pop-up menu of the Application Configuration folder, click **Generate**.

The DDL that defines the applications for System Management is generated into Working\platform\PRODUCTION\DataPersistenceApplication.

Close Object Builder:

1. Click **File > Save** to save your work.
2. Click **File > Exit** to close Object Builder.

Summary

You have created a component named Beneficiary that inherits from Person, and which provides persistence in a database table for its own attributes, as well as for the key attributes it shares with its parent, and uses its parent's table for other inherited attributes. You have implemented the key duplication pattern for inheritance with persistence.

You can find information on installing the component on the server in the System Administration Guide, "Configure a New Application Environment" chapter, or online in the topic Installing and configuring a new application.

You can find information on testing the installed component with a QuickTest client application in "Chapter 13. Testing applications with QuickTest" on page 611.

Inheritance with a single datastore

►CHANGED

If all the components in an inheritance hierarchy share a single datastore (for example, both `Person` and its child `Beneficiary` store their data in the same database table), then you can represent the datastore with a single persistent object. You can then map the data object attributes of each component to selected persistent object attributes.

Essentially, this approach flattens the object hierarchy into a single datastore. There is only one entry for each unique attribute, and only one persistent object for both parent and child components.

Advantages

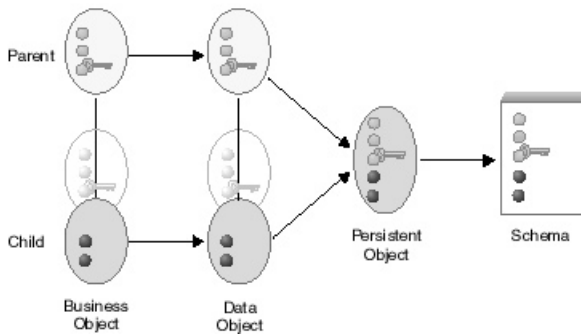
The biggest advantage of this inheritance pattern is that you can configure polymorphic behavior into your application, and do complex 'findBy' operations and queries. It is the most query-efficient pattern, because there are no joins (as in the key duplication pattern), or sequential table queries (as in the attributes duplication pattern).

Besides, this approach has faster access to the datastore, because both local and inherited attributes are in the same place.

Disadvantages

- This approach is problematic if you need to store pure parent objects (for example, a `Person` component that is not a `Beneficiary`). If you need to store both parent and child objects in the table, you must provide discriminator predicates for the data object implementations and configure the managed objects into either specialized or default polymorphic homes. (**Note:** It is recommended that you use this pattern instead of the inheritance with views pattern.)
- This approach is not very efficient in its use of space: each component accesses only a small part of the datastore, leaving most of the persistent object and schema unused for any one specific task.
- Growth in the hierarchy in object space requires the table to be redefined for the addition of new attributes from new data object implementations joining the hierarchy.
- This approach is problematic if your parent and child use different keys. Because both the parent's data and child's data is stored in a single datastore, the datastore needs to support both keys (the child's and the parent's), to ensure data for the right object type is returned. Generally, you should only use this pattern when the parent and the child use the same key.

Single Datastore Pattern



In this pattern:

- The parent's data object attributes and special framework methods are mapped to the shared persistent object.
- The child's data object attributes and special framework methods are also mapped to the shared persistent object.

RELATED CONCEPTS

"Chapter 6. Inheritance" on page 299

"Choosing an inheritance pattern for persistence" on page 304

RELATED TASKS

"Creating a child component" on page 306

"Defining a child with a single datastore"

"Tutorial: Inheritance with a single datastore" on page 344

Defining a child with a single datastore

This task covers the main steps necessary to create a component that inherits from another component already defined in Object Builder, and shares a single datastore with its parent. It does not cover every step; you should first be familiar with the tasks necessary to create a component without inheritance.

To use the single datastore pattern, the child must have the same key attributes as the parent, and the parent must be used for inheritance only (in other words, the only parent data in the datastore is for the child's inherited attributes). If the child has a different key, use the attributes duplication pattern. If the child has the same key but the parent is used as a real object (not just for inheritance), use the views pattern. The views pattern uses views of the datastore to select parent data of pure parent objects from parent data that is inherited by a child.

In the single datastore pattern, the parent's persistent object and schema include the attributes of the child component. Typically you would use this pattern after importing the data in a single large datastore into Object Builder, as part of a strategy to break up the data among several components in a class hierarchy.

To create the child component in Object Builder, follow these steps:

1. Create the business object file.
2. Add the business object interface, and select the parent's business object interface on the Interface Inheritance page.
3. Add the copy helper. You can include attributes of the parent's copy helper either by selecting specific attributes on the Name and Attributes page, or include all the parent's attributes by selecting the parent copy helper on the Implementation Inheritance page. Do **not** do both.
4. Add the business object implementation:
 - a. Under Data Object Interface, click **Add or select one later**. This allows you to add the data object interface in a separate step, and define its parent.
 - b. Select the parent's business object implementation on the Implementation Inheritance page.
 - c. Select the parent's key on the Key and Copy Helper page.
 - d. Do **not** override any attributes on the Attributes to Override page.
 - e. Do **not** override any relationships on the Relationships to Override page.
 - f. Select any methods you want to override on the Methods to Override page.
5. Add the managed object, and select the parent's managed object on the Implementation Inheritance page.
6. Add the data object interface:
 - a. From the business object implementation's pop-up menu, click **Add New Data Object Interface**.
 - b. Select the attributes and methods of the business object you want represented in the data object.
 - c. You should select the parent data object interface on the Interface Inheritance page.
7. Add the data object implementation:
 - a. From the data object interface's pop-up menu, click **Add Implementation**.
 - b. Select the parent data object implementation on the Implementation Inheritance page.

- c. Select the parent's key and the child's copy helper on the Key and Copy Helper page.
 - d. Map the child's attributes to the parent's persistent object on the Attributes Mapping page.
8. Map the child's data object attributes and special framework methods to the parent's (now shared) persistent object.

RELATED CONCEPTS

"Chapter 6. Inheritance" on page 299
"Choosing an inheritance pattern for persistence" on page 304
"Inheritance with a single datastore" on page 341
Components (*Programming Guide*)

RELATED TASKS

"Creating a child component" on page 306
"Building a child component" on page 380
"Tutorial: Inheritance with a single datastore"



Tutorial: Inheritance with a single datastore

Objectives

To export a component from an existing sample
To import the component into a new project
To define a single DB table that provides persistence for both parent and child attributes
To create a child component based on the table
To remap the parent component to the table
To generate the code for the component
To build the DLLs for the component

Before you begin

You need the following installed on your system:

- A CB Server
- A CB Client
- CB tools, including Samples
- DB2 Universal Database
- DB2 SDK
-  VisualAge for C++ and (for Java applications) VisualAge for Java
-  IBM C and C++ Compilers for AIX V3.6.4 and (for Java applications) VisualAge for Java

For a complete list of prerequisites that you must install on your system, see the *Getting Started with Component Broker* documentation for the appropriate platform.

You should be familiar with the Component Broker programming model, as described in the *IBM Component Broker Programming Guide*. At minimum, you should have read chapter 1, the introduction to the programming model.

You should be familiar with the different models for inheritance and persistence, as described in “Choosing an inheritance pattern for persistence” on page 304.

You should be familiar with the single datastore pattern for inheritance and persistence, as described in “Inheritance with a single datastore” on page 341.

Sample files

There is an equivalent sample for this exercise. The sample contains the results of all the inheritance tutorials. The sample includes:

- A BusinessObjects project that contains the finished Object Builder model
- A zipped Rational Rose model that contains definitions for the business object, key, copy helper, and data object interface
- Documentation for the sample, including instructions on testing it with QuickTest

The sample files are in the following directories (under <CBroker>, the install directory):

For C++:

```
samples\Tutorial\Fundamentals\Inherit\BusinessObjects  
samples\Tutorial\Fundamentals\Inherit\Docs\Inherit.html
```

No Java sample is available. However, the only difference is in the business object implementations, where you can change the implementation language to **Java** to turn the C++ sample into a Java sample.

There are some differences between the sample project and the one you create in this exercise.

- The sample contains four different versions of the Beneficiary component, demonstrating the different patterns for inheritance with persistence. This exercise results in only one version of the Beneficiary component.
- The sample contains a persistent object with a short name. This exercise uses a longer name to make the persistent object easier to distinguish in text.

- The sample has the parent and child in the same module. This exercise has the parent and child in separate modules, so that you can reuse the parent component more easily.

Description

In this exercise you define a parent component and child component that share a single datastore.

This inheritance pattern makes the most sense when parent and child share the same key. For a scenario where parent and child have different keys, see “Tutorial: Inheritance with attributes duplication” on page 310.

This scenario also does not use views, which means there is no easy way to determine when data is for a pure parent component, or part of the inherited data for a child component. While this is acceptable when there are no pure parent components to take into consideration (in other words, there are no pure parent instances to be persisted), it does not work well for datastores that contain a mix of objects. To manage distinct row types in a single table, you can configure polymorphic behavior into your application. See the task: Creating a specialized polymorphic home.

After you complete this exercise, you will have a component named Beneficiary that inherits from Person, and a single database table that provides persistence for both Beneficiary’s attributes and Person’s attributes.

The following tasks do not give explicit instructions for every step, but should at least get you into the right wizards. If you are experiencing problems, click the Help button within a wizard, or go to the Help pulldown in Object Builder.

For this exercise, you will complete the following tasks:

1. Exporting the parent component from the sample
2. Creating the new project
3. Importing the parent component
4. Creating a shared table definition (schema) and persistent object
5. Mapping the parent component to the shared persistent object
6. Adding a business object interface
7. Adding a key and copy helper
8. Adding a business object implementation
9. Adding a data object implementation and mapping it to the shared persistent object
10. Adding a managed object
11. Generating the code

12. Configuring the database
13. Defining a client DLL and server DLL

Exporting the parent component

Open the existing sample:

1. Start Object Builder.
2. In the Open Project wizard, type the path for the inheritance sample model
(<CBroker>\samples\Tutorial\Fundamentals\Inherit\BusinessObjects)
3. Click **Finish**.

Export the Person component objects:

1. Under the User-Defined Business Objects folder, locate the PersonFile business object file.
2. From the pop-up menu of PersonFile, click **Export**. The Export XML wizard opens to the XML Export Directory page.
3. Click **Finish**.
udbo.PersonFile.xml is exported to the specified directory. The XML file defines the Person business object, copy helper, key, and managed object (Person, PersonBO, PersonKey, PersonCopy, PersonMO).
4. Locate the PersonFileDO data object file in the User-Defined Data Objects folder, and export it in the same way.
uddo.PersonFileDO.xml is exported to the specified directory. The XML file defines the Person data object (PersonDO and PersonDOImpl).
5. Locate the PersonFileGroup schema group in the DBA-Defined Schemas folder, and export it in the same way.
uddbschema.PersonFileGroup.xml is exported to the specified directory. The XML file defines the PersonFileGroup schema group, the schema it contains, and the schema's associated persistent object (PersonFileGroup, CBSampDB.P_PFINH, PPOPFINH).
6. Close Object Builder.

You are ready to create a new project, and import the XML files.

Creating the project

Create a sample project to hold your work. For example, e:\tutorials\inhsngl

1. Start Object Builder.
2. In the Open Project wizard, type a name and path for the project directory.
3. Click **Finish**.
4. When asked whether you want to create a new project, click **Yes**.

Import the parent XML files

Import the definition of the Person component:

1. Click **File > Import Model** to open the Import Model wizard to the XML File Selection page.
2. Select udbo.PersonFile.xml:
 - a. Click **Add Another**.
 - b. Type the path and name for the file, or click **Browse** to locate and select the file.
 - c. Click **Refresh**.
3. Select uddo.PersonFileDO.xml and uddbschema.PersonFileGroup.xml in the same way.
4. Click **Finish**.

The component objects for Person appear in the Tasks and Objects pane. You are ready to define a child component that inherits from Person.

Creating the shared table and persistent object

You need to create a schema that contains columns for all of Person's and Beneficiary's attributes.

1. Create a file with the following contents. If you are viewing this online, you can cut and paste these lines directly into an editor:

```
CREATE TABLE S_DINH
(
  "ssNo" VARCHAR(20) NOT NULL ,
  "name" VARCHAR(100) NOT NULL ,
  "street" LONG VARCHAR ,
  "town" LONG VARCHAR ,
  "claimPayments" DOUBLE
  , PRIMARY KEY
  ( "ssNo", "name" )
);
```

2. From the pop-up menu of the DBA-Defined Schemas folder in Object Builder, click **Import > SQL**.
3. Select the file you created.
4. Name the database CBSampDB (or provide the name of your own database).
5. Name the group ShareDataGroup.
6. Click **Finish**.

The schema appears in the folder, under the schema group.

7. From the pop-up menu of the schema, click **Add Persistent Object**.
8. Name the persistent object SharedPO, and name its package file S_DINH.
9. Set its type of persistence to **Embedded SQL**, to match the type of persistence set in PersonDOImpl.
10. Click **Finish**.

The persistent object appears under the schema.

Mapping the shared persistent object to the parent

Replace PPOPFINH (the persistent object defined in the imported XML) with SharedPO.

Delete the old persistent object:

1. Delete Person's old persistent object (PPOPFINH) from under PersonDOImpl.
2. Delete Person's old schema (CBSampDB) from the DBA-Defined Schemas folder.
3. Delete Persons' old schema group (PersonFileGroup) from the DBA-Defined Schemas folder.

Map to the shared persistent object:

1. From the pop-up menu of PersonDOImpl, click **Properties** to open the Data Object Implementation wizard.
2. Click the page title and turn to the Associated Persistent Objects page.
3. Add SharedPO as an associated persistent object, with the instance name iPersonPO.
4. Click **Next** to turn to the Attributes Mapping page.
5. Map Person's attributes to their equivalents in iPersonPO.
6. Click **Next** to turn to the Methods Mapping page.
7. Map Person's methods to their equivalents in iPersonPO.
8. Click **Finish**.

SharedPO and its schema now appear under PersonDOImpl.

You can now define the child component.

Creating the business object interface

Define a business object file (SingleDataFile):

1. From the User-Defined Business Objects folder's pop-up menu, click **Add File** to open the Business Object File wizard.
2. Name the file.
3. Click **Finish**. The file now appears under the folder.

Add a module (SingleDataModule):

1. From the pop-up menu of the file, click **Add Module** to open the Business Object Module wizard.
2. Name the module.
3. Click **Finish**. The module now appears under the file.

Add an interface (Beneficiary) to SingleDataModule.

1. From the module's pop-up menu, click **Add Interface** to open the Business Object Interface wizard.
2. Name the interface Beneficiary.
3. Click the page title and turn to the Interface Inheritance page.
4. Add Person as a parent (replacing the default inheritance).
5. Click the page title and turn to the Attributes page.
6. Add the following attributes:
 - readonly long id
 - float claimPayments
7. Click **Finish**. The interface now appears under the module.

Adding the copy helper

Add BeneficiaryCopy:

1. From the interface's pop-up menu, click **Add Copy Helper** to open the Copy Helper wizard.
2. Add all attributes to the copy helper (both parent's and child's).
3. Click **Finish**. The copy helper now appears under the interface.

You do not need to add a key, because Beneficiary can re-use Person's key.

Adding the business object implementation

Add BeneficiaryBO:

1. From the interface's pop-up menu, click **Add Implementation** to open the Business Object Implementation wizard.
2. Set **Data Object Interface > Add or select one later** (you will create a new data object as a separate step).
3. Click the page title and turn to the Implementation Inheritance page.
4. Add PersonBO as a parent.
5. Click the page title and turn to the Key and Copy Helper page.
6. Select PersonKey and BeneficiaryCopy.
7. Click **Finish**. The business object implementation appears under the business object interface.

Adding the data object interface

Add BeneficiaryDO:

1. From the business object implementation's pop-up menu, click **Add New Data Object Interface** to open the Data Object Interface wizard.
2. Select all the business object attributes as state data (to be preserved in the data object).
3. Click the page title and turn to the Interface Inheritance page.
4. Add PersonDO as a parent.

5. Click **Finish**. The data object interface appears under the business object implementation.

Adding the data object implementation, and mapping it to the shared persistent object

Add BeneficiaryDOImpl, and map it to SharedPO:

1. From the data object interface's pop-up menu, click **Add Implementation** to open the Data Object Implementation wizard.
2. Click **Next** to turn to the Behavior page.
3. Set the following patterns:
 - **Environment - BOIM with any key**
 - **Type of Persistence - Embedded SQL**
 - **Data Access Pattern - Delegating**
4. Click **Next** to turn to the Implementation Inheritance page.
5. Add PersonDOImpl as a parent.
6. Click the page title and turn to the Key and Copy Helper page.
7. Select PersonKey and BeneficiaryCopy.
8. Click the page title and turn to the Attributes Mapping page.
9. Map claimPayments to SharedPO.claimPayments. SharedPO is available for selection because it is associated with Beneficiary's parent. It is recommended that you inspect the method mappings, and make sure that they map to SharedPO.
10. Click **Finish**. The data object implementation appears under the data object interface.

Adding the managed object

Add BeneficiaryMO:

1. From the business object implementation's pop-up menu, click **Add Managed Object** to open the Managed Object wizard to the Name and Service page.
2. Accept all the defaults and click **Finish**.
3. Click **File > Save** to save your work before continuing to the next step.

Generating the code

You are now ready to generate the code for the objects you have defined. Generated code will appear in your project's `\Working\platform` directory (for example, `e:\tutorials\inhsngl\Working\NT`).

1. From the pop-up menu of the User-Defined Business Objects folder, click **Generate > All**.

The code generation can take several minutes.
2. Review the contents of the `Working\platform` directory. All the source files for the component have been generated, and you can now define how to build them.

While you can view the generated source files for a particular object (by selecting **View Source** from its pop-up menu), you cannot edit the source files through Object Builder. If you edit them outside of Object Builder, you should restrict your changes to method implementations, and import your changes back into Object Builder before re-generating.

Configuring the database

You need to define (in DB2) the CBSampDB database and its table (S_DINH) that your components will access. You should have a database administrator perform this procedure.

To configure the database and table, you need to enter the following commands from a DB2 command prompt.

```
create database CBSampDB
connect to CBSampDB
```

```
db2 -t -f S_DINH.sql
```

As the name of the inheritance pattern (single datastore) suggests, there is only one table definition for both Person and Beneficiary. This is appropriate for cases where there will be no pure Person components, and the datastore does not need to distinguish between Person components and Beneficiary components.

Defining a client DLL and server DLL

Configure the build process that will create the client and server DLLs.

While “DLL” is the generic term used in Object Builder, the configuration for the build actually results in whatever the appropriate targets are for the selected platforms. For example, on AIX the build process creates shared library files (lib*.so). If you chose to create a Java business object, then in addition to DLLs there will also be JAR files. The names for these files are derived from the name you provide for the DLL file, within the DLL configuration node.

Start by defining the client DLL configuration and client DLL or library file (for this exercise, name them both clientIN). The client DLL provides client applications with access to the component on the server, using the key and copy helper. You must also include the business object interface, which defines the methods and attributes of the component that the client can access.

1. From the pop-up menu of the Build Configuration folder, click **Add Client DLL** to open the Client DLL wizard to the Name and Options page.
2. Name the configuration. This is the name that uniquely identifies the configuration node.

3. Name the library as well. This is the name for the makefile and for the resulting DLL file. You can create multiple configuration nodes that produce different versions of the same DLL. For example, you could set different compile and link options to produce a development version and a deployment version of the same DLL.
4. For this exercise, the default configuration options are sufficient.
5. Click the title and turn to the Client Source Files page.
6. Select all the client files and add them to the **Items Chosen** list.
7. Click **Finish**.

The clientIN DLL configuration appears under the Build Configuration folder.

Define the server DLL configuration and server DLL file (for this exercise, name them both serverIN). The server DLL is installed on the server to deploy the component, making it available for access by client applications and other components.

1. From the pop-up menu of the Build Configuration folder, click **Add Server DLL** to open the Server DLL wizard.
2. Name the configuration and target DLL file.
3. Click **Next** to turn to the Libraries to Link With page.
4. Add the client DLL file. The server DLL needs access to the client DLL because the client DLL defines the component's interface, and the component implementation inherits from it.
5. Click **Next** to turn to the Server Source Files page.
6. Select all the server source files for the component, and add them to the **Items Chosen** list.
7. Click **Finish**.

The serverIN DLL configuration appears under the Build Configuration folder.

Building the DLLs

To generate the makefiles, based on the configuration choices you made in the DLL wizards:

1. From the pop-up menu of the Build Configuration folder, click **Generate > All > All Targets**. This generates makefiles for all the DLL files defined in the folder and generates an all.mak file that calls the individual DLL makefiles. By choosing **All Targets**, you set the default behavior of the makefile, when it is built. You can still override this default when you build (for example, you can choose to build Java targets only, and override the default of all targets).

To build the DLL and (optionally) JAR files:


1. From the Build Configuration folder's pop-up menu, select **Build > Out-of-Date Targets > C++**.

You always need to generate C++ targets, even if you selected **Java** as your business object implementation language. The other objects of the component (such as the data object implementation and managed object) are still implemented in C++.

2. If you selected **Java** as your business object implementation language, then you also need to build the Java targets. From the Build Configuration folder's pop-up menu, select **Build > Out-of-Date Targets > Java**.

When you have a mixed-language application (some business objects are implemented in C++, some in Java) you must generate both C++ and Java targets for *all* components, even for those implemented in C++. The .jar files for C++ components allow Java components to interact with the C++ components on the server.

For this exercise, your application contains just a single component, so you only need to build the Java targets if you implemented the business object in Java.

 The clientIN.dll and serverIN.dll files are stored in the working\NT\PRODUCTION directory.

 The libclientIN.so and libserverIN.so files are stored in the working/AIX/PRODUCTION directory.

If you have a Java business object, the clientIN.jar and serverIN.jar files are stored in the Working\platform\PRODUCTION directory.

If you had a Java client application, regardless of the language your component is implemented in, you would also select **Build > Out-of-Date Targets > Java Client Bindings** to generate Java client bindings (working\platform\`<build style>`\JCB\JCBclientIN.jar). `<build style>` is one of the configuration directories: NOOPT, PRODUCTION, TRACE, TRACE_DEBUG. See Platform-specific information for more information.

For this exercise, you will be using QuickTest to run and test your application, which has its own build process that includes generation of client bindings.



Defining an application family and application

Define the application family (DataPersistenceApplication). An application family groups a set of applications within System Management.

1. From the pop-up menu of the Application Configuration folder, click **Add Application Family** to open the Application Family wizard.
2. Name the application family.
3. Click **Finish**.

The DataPersistenceApplication application family appears under the Application Configuration folder.

Define the server application (DataPersistenceObjects). An application defines a set of components that will operate together on the server. The application name you provide here is the name that will be used by System Management, when the application and its components are deployed.

1. From the pop-up menu of the application family, click **Add Application** to open the Application wizard.
2. Name the application.
3. Click **Next** to turn to the Additional Executables page.
4. Select the platform you are configuring the application for (for example, **NT Files**).
5. Add the file S_DINH.sql:
 - a. Click **Add Another**.
 - b. Click the **Browse** button to open the Executables to Include dialog.
 - c. Locate your Object Builder working directory.
 - d. Select S_DINH.sql
 - e.  Click the **Open** button.
 - f.  Click the **OK** button.
6. Add the file S_DINH.bnd in the same manner.
7. Click **Finish**.

The DataPersistenceObjects application appears under the DataPersistenceApplication application family.

Configuring the components with the application

Configure Person's managed object (PersonFile PersonModule::PersonMO) with the application (DataPersistenceObjects). Create a new container instance (DataPersistenceContainer) that will **Throw an exception** when a method is called outside a transaction.

1. From the pop-up menu of the application, click **Add Managed Object** to open the Managed Object Configuration wizard.
2. In the **Managed Object** list, select the component's managed object. The other lists should be automatically populated with the appropriate selections.
3. Click **Next** and select the data object implementation. Click **Add Another**. The appropriate data object implementation and DLL will be selected. If there were more than one data object implementation for the component, or it was being built into more than one DLL, you would click **Add Another** again to add the extra information.

4. Click **Next** to turn to the Container page.
5. Check the **Create a new container** option. An extra page is inserted into the wizard, to allow you to name the new container and specify its policies.
6. Click **Next** to turn to the Create New Container page.
7. Name the container, and set its behavior for methods called outside a transaction.

Most of the container's behavior is defined for you, based on the managed object you are configuring, and the data object implementation you selected.

8. Click **Next** and review the home. The appropriate home is already selected, and the default options are acceptable. If there were more than one appropriate home, you could choose the home to use.
9. Click **Finish**.

The PersonMO managed object configuration appears under the DataPersistenceObjects application, and the new container DataPersistenceContainer appears in the Container Definition folder. You can review the properties of the container, including the service and data access patterns that have been selected for you, by clicking **Properties** from the container's pop-up menu.

Configure Beneficiary's managed object in the same way:

1. From the pop-up menu of the application, click **Configure Managed Object** to open the Managed Object Configuration wizard.
2. In the **Managed Objectlist**, select SingleDataFile SingleDataModule::BeneficiaryMO.
3. Click the page title and turn to the Container page.
4. Select DataPersistenceContainer. Because Person and Beneficiary are similar components (with the same type of datastore and the same policies), they can use the same container.
5. Click **Finish**.

Generate the application installation information:

1. From the pop-up menu of the Application Configuration folder, click **Generate**.

The DDL that defines the applications for System Management is generated into Working\platform\PRODUCTION\DataPersistenceApplication.

Close Object Builder:

1. Click **File > Save** to save your work.
2. Click **File > Exit** to close Object Builder.

Summary

You have created a component named Beneficiary that inherits from Person, and defined a shared database table that provides persistence for attributes of both components. You have implemented the shared datastore pattern for inheritance with persistence.

You can find information on installing the component on the server in the System Administration Guide, “Configure a New Application Environment” chapter, or online in the topic Installing and configuring a new application.

You can find information on testing the installed component with a QuickTest client application in “Chapter 13. Testing applications with QuickTest” on page 611.

Inheritance with views

►CHANGED

Important: The functionality of the inheritance with views pattern is being replaced with the polymorphic homes function. When starting new work, use polymorphic homes. If you have applications that use inheritance with views, you must migrate them to polymorphic homes. See “Polymorphic homes” on page 581 for more information.

If your persistence is provided by a single datastore that stores both pure parent objects and child objects, you can use views to select out the appropriate data from the datastore. This combines an attributes duplication approach (one persistent object per component, with the parent’s attributes duplicated in the child’s persistent object) with a single datastore approach (a single datastore for all components in the hierarchy). The attributes duplication approach is used for retrieving data (allowing greater precision in selection of data), and the single datastore approach is used for changing (creating, updating, or deleting) data.

To accomplish this, the table must include a mechanism for identifying which data belongs to the parent, and which data belongs to the child. Typically, this can be accomplished by identifying a unique attribute for each component. For example, Beneficiary has an attribute claimPayment, and Person does not. So if the claimPayments column contains a value, then the component must be a Beneficiary.

Using the identifying attribute, you can create selective views of the table for each component type, and then create a persistent object for each view. These are the persistent objects that will be used to retrieve data, following the same pattern as the object-oriented approach. For example, Beneficiary could have a persistent object BeneficiaryPO, which represents a view of the table where

claimPayments=notNull, and Person could have PersonPO, with a view of the table where claimPayments=Null. These persistent objects are referred to as *retrieving* persistent objects.

You also need to create persistent objects that map all the rows in the table, but only the relevant columns. These are the persistent objects that components will use to create, update, or delete data. Children at the bottom of a hierarchy can use persistent objects that map directly to the table, because all columns in the table are relevant; all data is inherited. (Note that sibling subclasses of a single base class do not know about each other's attributes.) All other components in the hierarchy require an additional view for this purpose. For example, in the hierarchy Person -> Beneficiary -> PrimaryBeneficiary, Person and Beneficiary would require their own views, but PrimaryBeneficiary could use the full table, and would not require its own view. These persistent objects are referred to as *updating* persistent objects.

The need for special updating persistent objects is based on the way null columns are handled. Because the concept of a null value does not exist in the object-oriented world of the components, a new parent component that maps directly to the shared table could accidentally initialize the unused columns to a not-null value, when the persistent object state was applied to the new database row. Later, when a retrieving persistent object performed a notNull check on the data, the row would be incorrectly identified as belonging to a child component, and would be inaccessible. The special updating persistent object, and its selective view, insulate the component from any contact with columns that are not relevant to it, and prevent accidental initialization of child columns.

Note: You can use the null-tolerance features to prevent this type of mistake from happening. See Null value tolerance with sentinel values for more information.

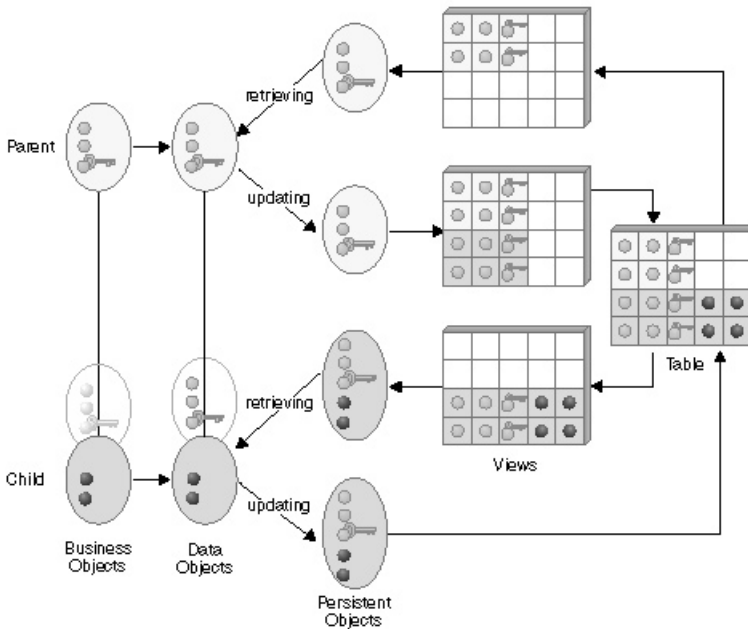
Advantages

The advantage of this approach is that it takes up less space than the pure attributes duplication approach (because there is only one table for all attributes), with more precision than the single datastore approach (which cannot easily distinguish between pure parent data and inherited parent data).

Disadvantages

It is neither as efficient as the pure attributes duplication pattern, nor as fast as the single datastore pattern. Also, like the single datastore pattern, the views pattern is problematic if your parent and child use different keys, because then the shared table would need to have two primary keys at once. Generally, you should only use this pattern when the parent and the child use the same key.

Single Datastore with Views Pattern



In this pattern:

- The parent's data object attributes are mapped first to its updating persistent object, and then to its retrieving persistent object.
- The parent's retrieve method is mapped to its retrieving persistent object.
- The parent's insert, update, and delete methods are mapped to its updating persistent object.
- The parent's setConnection method is mapped first to its updating persistent object, and then to its retrieving persistent object.
- The child's data object attributes and its inherited attributes are mapped first to its updating persistent object, and then to its retrieving persistent object.
- The child's retrieve method maps first to its retrieving persistent object, and then to its updating persistent object, with the **Always complete calling sequence** check box **not** selected.
- The child's insert, update, and delete methods are mapped to its updating persistent object only.
- The child's setConnection method is mapped first to its updating persistent object, and then to its retrieving persistent object.

This mapping creates a new parent along with the child, and deletes the parent along with the child.

If you wanted to create a new child from an existing parent, you could find the existing parent, create a copy of its attribute values, delete the parent, and then create the child as a new object with the values of the deleted parent.

If you wanted to delete the child and leave the parent entry, you could copy the existing parent values, continue with the deletion of the child, and then re-create the parent with the copied values.

RELATED CONCEPTS

“Chapter 6. Inheritance” on page 299

“Choosing an inheritance pattern for persistence” on page 304

“Polymorphic homes” on page 581

“Null value tolerance with sentinel values” on page 155

RELATED TASKS

“Creating a child component” on page 306

“Defining a child with views”

“Tutorial: Inheritance with views” on page 362

Defining a child with views

This task covers the main steps necessary to create a component that inherits from another component already defined in Object Builder, shares the same database table as its parent, and uses views to select the appropriate data out of the table. It does not cover every step; you should first be familiar with the tasks necessary to create a component without inheritance.

This approach is a variant of the single datastore pattern. To use this pattern, the child must have the same key attributes as the parent. If the child has a different key, use the attributes duplication pattern.

Typically you would use this pattern after importing the data in a single large datastore into Object Builder, as part of a strategy to break up the data among several components in a class hierarchy.

The parent maps to its data as follows:

- A single database table stores all the attributes for both parent and child.
- A view of the table selects the columns that apply to the parent. The view represents all rows, but only some columns. Its persistent object is used when *updating* parent attributes.
- A view of the table selects out those rows in which a unique child attribute is null (that is, the rows that do not contain data for a child component). The view represents only relevant rows. Its persistent object is used when *retrieving* parent attributes.

- The parent's data object attributes, and its retrieve method, map to the *retrieving* persistent object.
- The parent's include, update, delete, and setConnection methods map to the *updating* persistent object.

To create the child component in Object Builder, follow these steps:

1. Create a view of the shared table, selecting out those rows in which a unique child attribute is not null (that is, the rows that contain data for a child component).
2. Create a persistent object based on that view. This is the child's equivalent of the parent's retrieving persistent object.
3. Create a persistent object based on the original table. This is the child's equivalent of the parent's updating persistent object. The child does not need a separate view, because it does not need to exclude columns.

In an inheritance case with deeper nesting, all ancestors would require views. Only components at the absolute bottom of the hierarchy, with access to all the inherited data of the components above it, would map directly to the table.

For example, in the hierarchy Person -> Beneficiary -> PrimaryBeneficiary, you would need to create views for Person and Beneficiary, but could map directly to the shared table with PrimaryBeneficiary.

4. Create the business object file.
5. Add the business object interface, and select the parent's business object interface on the Interface Inheritance page.
6. Add the copy helper. You can include attributes of the parent's copy helper either by selecting specific attributes on the Name and Attributes page, or include all the parent's attributes by selecting the parent copy helper on the Implementation Inheritance page. Do **not** do both.
7. Add the business object implementation:
 - a. Under Data Object Interface, click **Add or select one later**. This allows you to add the data object interface in a separate step, and define its parent.
 - b. Select the parent's business object implementation on the Implementation Inheritance page.
 - c. Select the parent's key on the Key and Copy Helper page.
 - d. Do **not** override any attributes on the Attributes to Override page.
 - e. Do **not** override any relationships on the Relationships to Override page.
 - f. Select any methods you want to override on the Methods to Override page.
8. Add the managed object, and select the parent's managed object on the Implementation Inheritance page.

9. Add the data object interface:
 - a. From the business object implementation's pop-up menu, click **Add New Data Object Interface**.
 - b. Select the attributes and methods of the business object you want represented in the data object.
 - c. You should select the parent data object interface on the Interface Inheritance page.
10. Add the data object implementation, and select the parent data object implementation on the Implementation Inheritance page.
11. Map the data object implementation to the table-based persistent object (the *updating* persistent object) and the view-based persistent object (the *retrieving* persistent object, as follows:
 - a. The child's data object attributes and its inherited attributes are mapped to its retrieving persistent object.
 - b. The child's retrieve method maps to first its retrieving persistent object and then its parent's retrieving persistent object, with the **Always complete calling sequence** option **not** checked.
 - c. The child's insert, update, delete, and setConnection methods are mapped to its updating persistent object.

This always creates a new parent along with the child, and deletes the parent along with the child.

If you wanted to create a new child from an existing parent, you could still find the existing parent, create a copy of its attribute values, delete the parent, and then create the child as a new object with the values of the deleted parent.

If you wanted to delete the child and leave the parent entry, you could still copy the existing parent values, continue with the deletion of the child, and then re-create the parent with the copied values.

RELATED CONCEPTS

"Chapter 6. Inheritance" on page 299

"Choosing an inheritance pattern for persistence" on page 304

"Inheritance with views" on page 357

Components (*Programming Guide*)

RELATED TASKS

"Creating a child component" on page 306

"Building a child component" on page 380

"Tutorial: Inheritance with views"

Tutorial: Inheritance with views



Important: The functionality of the inheritance with views pattern is being replaced with the polymorphic homes function. When starting new

work, use polymorphic homes. If you have applications that use inheritance with views, it is recommended that you adjust your model to use polymorphic homes. See *Polymorphic homes* for more information.

Objectives

To export a component from an existing sample

To import the component into a new project

To define a single DB table that provides persistence for both parent and child attributes

To define views that separate out child and parent attributes from the shared table

To create a child component based on the child view



To remap the parent component to the parent view

To generate the code for the component

To build the DLLs for the component

Before you begin

You need the following installed on your system:

- A CB Server
- A CB Client
- CB tools, including Samples
- DB2 Universal Database
- DB2 SDK
-  VisualAge for C++ and (for Java applications) VisualAge for Java
-  IBM C and C++ Compilers for AIX V3.6.4 and (for Java applications) VisualAge for Java

For a complete list of prerequisites that you must install on your system, see the *Getting Started with Component Broker* documentation for the appropriate platform.

You should be familiar with the Component Broker programming model, as described in the *IBM Component Broker Programming Guide*. At minimum, you should have read chapter 1, the introduction to the programming model.

You should be familiar with the different models for inheritance and persistence, as described in “Choosing an inheritance pattern for persistence” on page 304.

You should be familiar with the single datastore pattern for inheritance and persistence, as described in “Inheritance with views” on page 357.

Sample files

There is an equivalent sample for this exercise. The sample contains the results of all the inheritance tutorials. The sample includes:

- A BusinessObjects project that contains the finished Object Builder model
- A zipped Rational Rose model that contains definitions for the business object, key, copy helper, and data object interface
- Documentation for the sample, including instructions on testing it with QuickTest

The sample files are in the following directories (under <CBroker>, the install directory):

For C++:

```
samples\Tutorial\Fundamentals\Inherit\BusinessObjects  
samples\Tutorial\Fundamentals\Inherit\Docs\Inherit.html
```

No Java sample is available. However, the only difference is in the business object implementations, where you can change the implementation language to **Java** to turn the C++ sample into a Java sample.

There are some differences between the sample project and the one you create in this exercise.

- The sample contains four different versions of the Beneficiary component, demonstrating the different patterns for inheritance with persistence. This exercise results in only one version of the Beneficiary component.
- The sample contains some persistent objects with short names. This exercise uses longer names throughout, to make persistent objects easier to distinguish in the text.
- The sample has the parent and child in the same module. This exercise has the parent and child in separate modules, so that you can re-use the parent component more easily.

Description

In this tutorial you define a parent component and child component that share the same database table, but map to it selectively using component-specific views. This inheritance pattern makes the most sense when parent and child share the same key. For a scenario where parent and child have different keys, see “Tutorial: Inheritance with attributes duplication” on page 310.

After you complete this scenario, you will have a component named Beneficiary that inherits from Person, a single database table that provides persistence for both Beneficiary’s attributes and Person’s attributes, and views of the table that provide component-specific schemas.

The following tasks do not give explicit instructions for every step, but should at least get you into the right wizard. If you are experiencing problems, click the Help button within a wizard, or go to the Help menu in Object Builder.

For this exercise, you will complete the following tasks:

1. Exporting the parent component from the sample
2. Creating the new project
3. Importing the parent component
4. Creating a shared table definition (schema) and persistent object
5. Creating component-specific views and associated persistent objects
6. Mapping the parent component to persistent objects
7. Adding a business object interface
8. Adding a key and copy helper
9. Adding a business object implementation
10. Adding a data object implementation and mapping it to persistent objects
11. Adding a managed object
12. Generating the code
13. Configuring the database
14. Defining a client DLL and server DLL

Exporting the parent component

Open the existing sample:

1. Start Object Builder.
2. In the Open Project wizard, type the path for the inheritance sample model
(<CBroker>\samples\Tutorial\Fundamentals\Inherit\BusinessObjects)
3. Click **Finish**.

Export the Person component objects:

1. Under the User-Defined Business Objects folder, locate the PersonFile business object file.
2. From the pop-up menu of PersonFile, click **Export**. The Export XML wizard opens to the XML Export Directory page.
3. Click **Finish**.
udbo.PersonFile.xml is exported to the specified directory. The XML file defines the Person business object, copy helper, key, and managed object (Person, PersonBO, PersonKey, PersonCopy, PersonMO).
4. Locate the PersonFileDO data object file in the User-Defined Data Objects folder, and export it in the same way.
uddo.PersonFileDO.xml is exported to the specified directory. The XML file defines the Person data object (PersonDO and PersonDOImpl).

5. Locate the PersonFileGroup schema group in the DBA-Defined Schemas folder, and export it in the same way.

uddbschema.PersonFileGroup.xml is exported to the specified directory. The XML file defines the PersonFileGroup schema group, the schema it contains, and the schema's associated persistent object (PersonFileGroup, CBSampDB.P_PFINH, PPOPFINH).

6. Close Object Builder.

You are ready to create a new project, and import the XML files.

Creating the project

Create a sample project to hold your work. For example, e:\tutorials\inhview

1. Start Object Builder.
2. In the Open Project wizard, type a name and path for the project directory.
3. Click **Finish**.
4. When asked whether you want to create a new project, click **Yes**.

Import the parent XML files

Import the definition of the Person component:

1. Click **File > Import Model** to open the Import Model wizard to the XML File Selection page.
2. Select udbo.PersonFile.xml:
 - a. Click **Add Another**.
 - b. Type the path and name for the file, or click **Browse** to locate and select the file.
 - c. Click **Refresh**.
3. Select uddo.PersonFileDO.xml and uddbschema.PersonFileGroup.xml in the same way.
4. Click **Finish**.

The component objects for Person appear in the Tasks and Objects pane. You are ready to define a child component that inherits from Person.

Creating the shared table

You need to create a schema that contains columns for all of Person's and Beneficiary's attributes.

1. Create a file with the following contents. If you are viewing this online, you can cut and paste these lines directly into an editor:

```
CREATE TABLE S_DINH
(
  "ssNo" VARCHAR(20) NOT NULL ,
  "name" VARCHAR(100) NOT NULL ,
  "street" LONG VARCHAR ,
  "town" LONG VARCHAR ,
```

```
"claimPayments" DOUBLE
, PRIMARY KEY
( "ssNo", "name" )
);
```

2. From the pop-up menu of the DBA-Defined Schemas folder in Object Builder, click **Import > SQL**.
3. Select the file you created.
4. Name the database CBSampDB (or provide the name of your own database).
5. Name the group ShareDataGroup.
6. Click **Finish**.

The schema appears in the folder, under the schema group.

Creating views for the parent

The parent requires two views: one for retrieval, and one for updates. Both these views are based on the shared table defined in the previous task.

The retrieval view needs to differentiate between pure Person components and Beneficiary components with inherited Person attributes. The update view merely needs to hide child-specific attributes from the parent.

Create the retrieval view CBSampDB.PView:

1. From the pop-up menu of ShareDataGroup, click **Add SQL View**. The SQL View Editor opens.
2. Name the view PView.
3. Click on the View Work Area tab.
4. Click on the S_DINH table in the Schemas pane.
5. In the Clauses pane, click the Selected Columns tab.
6. In the Columns pane, click on the columns you want represented in the view:
 - ssNo
 - name
 - street
 - town

Their data appears in the fields of the Selected Columns page.

7. In the Clauses pane, click the Where tab.
8. In the Columns pane, click on claimPayments. Its data appears in the fields of the Where page.

This is the column you are using to test whether the row contains data for the parent component, and to exclude rows that are for child components.

9. Click the list button of the Conditions field on the Where page, and select **Is NULL**.

This ensures that only rows without claimPayments information appear in the view. Because the claimPayments column only contains information for Beneficiary components, this excludes child data from the view.

If Person had additional child components, you could add additional **Is NULL** conditions, based on their unique attributes, to exclude them from the parent's view.

10. Click on the View Summary tab.
11. Review the SQL clauses that define the view, based on your selections on the previous page.
12. Click **Finish**.

The view appears under ShareDataGroup, in the DBA-Defined Schemas folder.

Create the update view CBSampDB.VP_DINH:

1. From the pop-up menu of ShareDataGroup, click **Add SQL View**. The SQL View Editor opens.
2. Name the view VP_DINH.
3. Click on the View Work Area tab.
4. Click on the S_DINH table in the Schemas pane.
5. In the Clauses pane, click the Selected Columns tab.
6. In the Columns pane, click on the columns you want represented in the view:
 - ssNo
 - name
 - street
 - town

Their data appears in the fields of the Selected Columns page.

7. Click **Finish**.

Creating the parent's persistent objects

Replace PPOPFINH (the persistent object defined in the imported XML) with PViewPO and ViewPPO, based on the new views of the shared table.

Delete the old persistent object:

1. Delete Person's old persistent object (PPOPFINH) from under PersonDOImpl.

2. Delete Person's old schema (CBSampDB) from the DBA-Defined Schemas folder.
3. Delete Persons' old schema group (PersonFileGroup) from the DBA-Defined Schemas folder.

Add the persistent object for retrieval:

1. From the pop-up menu of CBSampDB.PView, click **Add Persistent Object**.
2. Name the persistent object PViewPO, and name its package file PView.
3. Set its type of persistence to **Embedded SQL**, to match the type of persistence set in PersonDOImpl.
4. Click **Finish**.

Add the persistent object for updates:

1. From the pop-up menu of CBSampDB.VP_DINH, click **Add Persistent Object**.
2. Name the persistent object ViewPPO, and name its package file VP_DINH.
3. Set its type of persistence to **Embedded SQL**, to match the type of persistence set in PersonDOImpl.
4. Click **Finish**.

The persistent objects appear under their schemas.

Mapping the parent's data object to persistent objects

Map PersonDOImpl to PViewPO and ViewPPO:

1. From the pop-up menu of PersonDOImpl, click **Properties** to open the Data Object Implementation wizard.
2. Click the page title and turn to the Associated Persistent Objects page.
3. Add ViewPPO as an associated persistent object, with the instance name iViewPPO.
4. Add PViewPO as an associated persistent object, with the instance name iPViewPO.
5. Click **Next** to turn to the Attributes Mapping page.
6. Map each attribute of Person to first its equivalent in iViewPPO, and then its equivalent in iPViewPO.
7. Click **Next** to turn to the Methods Mapping page.
8. Map Person's retrieve method to iPViewPO.retrieve.
9. Map Person's insert, update, and delete methods to iViewPPO.insert, iViewPPO.update, and iViewPPO.delete.
10. Map Person's setConnection method to first iViewPPO.setConnection, and then iPViewPO.setConnection.
11. Click **Finish**.

PViewPO and ViewPPO now appear under PersonDOImpl.

You can now define the child component.

Creating the view for the child

The child component also needs a retrieval view, which can differentiate between pure Person components and Beneficiary components with inherited Person attributes. It does not need a special update view, because the child maps to all the columns in the shared table: it does not need to exclude any columns.

Create the retrieval view CBSampDB.BView:

1. From the pop-up menu of ShareDataGroup, click **Add SQL View**. The View Editor opens.
2. Name the view BView.
3. Click on the View Work Area tab.
4. Click on the S_DINH table in the Schemas pane.
5. In the Clauses pane, click the Selected Columns tab.
6. In the Columns pane, click on the columns you want represented in the view:
 - ssNo
 - name
 - street
 - town
 - claimPayments

Their data appears in the fields of the Selected Columns page.

7. In the Clauses pane, click the Where tab.
8. In the Columns pane, click on claimPayments. Its data appears in the fields of the Where page.

This is the column you are using to test whether the row contains data for the child component, and to exclude rows that are for parent components.
9. Click the list button of the Conditions field on the Where page, and select **Is Not NULL**.

This ensures that only rows with claimPayments information appear in the view. Because the claimPayments column only contains information for Beneficiary components, this excludes data of pure parent components from the view.
10. Click on the View Summary tab.
11. Review the SQL clauses that define the view, based on your selections on the previous page.
12. Click **OK**.

The view appears under ShareDataGroup, in the DBA-Defined Schemas folder.

Creating the child's persistent objects

Create BViewPO, based on the new view of the shared table:

1. From the pop-up menu of CBSampDB.BView, click **Add Persistent Object**.
2. Name the persistent object BViewPO, and name its package file BView.
3. Set its type of persistence to **Embedded SQL**.
4. Click **Finish**.

Create SharedPO, based on the actual shared table:

1. From the pop-up menu of the schema, click **Add Persistent Object**.
2. Name the persistent object SharedPO, and name its package file S_DINH.
3. Set its type of persistence to **Embedded SQL**, to match the type of persistence set in PersonDOImpl.
4. Click **Finish**.

The persistent objects appear under their schemas.

You can now define the child component.

Creating the business object interface

Define a business object file (ViewDataFile):

1. From the User-Defined Business Objects folder's pop-up menu, click **Add File** to open the Business Object File wizard.
2. Name the file.
3. Click **Finish**. The file now appears under the folder.

Add a module (ViewDataModule):

1. From the pop-up menu of the file, click **Add Module** to open the Business Object Module wizard.
2. Name the module.
3. Click **Finish**. The module now appears under the file.

Add an interface (Beneficiary) to SingleDataModule.

1. From the module's pop-up menu, click **Add Interface** to open the Business Object Interface wizard.
2. Name the interface Beneficiary.
3. Click the page title and turn to the Interface Inheritance page.
4. Add Person as a parent (replacing the default inheritance).
5. Click the page title and turn to the Attributes page.
6. Add the following attributes:
 - readonly long id

- float claimPayments

7. Click **Finish**. The interface now appears under the module.

Adding the copy helper

Add BeneficiaryCopy:

1. From the interface's pop-up menu, click **Add Copy Helper** to open the Copy Helper wizard.
2. Add all attributes to the copy helper (both parent's and child's).
3. Click **Finish**. The copy helper now appears under the interface.

You do not need to add a key, because Beneficiary can re-use Person's key.

Adding the business object implementation

Add BeneficiaryBO:

1. From the interface's pop-up menu, click **Add Implementation** to open the Business Object Implementation wizard.
2. Set **Data Object Interface > Add or select one later** (you will create a new data object as a separate step).
3. Click the page title and turn to the Implementation Inheritance page.
4. Add PersonBO as a parent.
5. Click the page title and turn to the Key and Copy Helper page.
6. Select PersonKey and BeneficiaryCopy.
7. Click **Finish**. The business object implementation appears under the business object interface.

Adding the data object interface

Add BeneficiaryDO:

1. From the business object implementation's pop-up menu, click **Add New Data Object Interface** to open the Data Object Interface wizard.
2. Select all the business object attributes as state data (to be preserved in the data object).
3. Click the page title and turn to the Interface Inheritance page.
4. Add PersonDO as a parent.
5. Click **Finish**. The data object interface appears under the business object implementation.

Adding the data object implementation, and mapping it to persistent objects

Add BeneficiaryDOImpl, and map it to SharedPO:

1. From the data object interface's pop-up menu, click **Add Implementation** to open the Data Object Implementation wizard.
2. Click **Next** to turn to the Behavior page.
3. Set the following patterns:

- **Environment > BOIM with any key**
 - **Type of Persistence > Embedded SQL**
 - **Data Access Pattern > Delegating**
4. Click **Next** to turn to the Implementation Inheritance page.
 5. Add PersonDOImpl as a parent.
 6. Click the page title and turn to the Key and Copy Helper page.
 7. Select PersonKey and BeneficiaryCopy.
 8. Click the page title and turn to the Associated Persistent Objects page.
The parent's persistent objects appear by default.
 9. Add BViewPO as an associated persistent object, with the instance name iBViewPO.
 10. Add SharedPO as an associated persistent object, with the instance name iSharedPO.
 11. Click **Next** to turn to the Attributes Mapping page.
 12. Map Beneficiary's claimPayments attribute to first iSharedPO.claimPayments, and then iBViewPO.claimPayments.
 13. Map Beneficiary's inherited attributes to first iSharedPO and then iBViewPO.
For example, Person.ssNo maps to first iSharedPO.ssNo and then iBViewPO.ssNo
 14. Click **Finish**.

Mapping the special framework methods

Map the way in which the data object's special framework methods will call the persistent objects' special framework methods:

1. From BeneficiaryDOImpl's pop-up menu, click **Properties** to open the Data Object Implementation wizard.
2. Click the page title and turn to the Methods Mapping page.
3. Map Beneficiary's retrieve method first to iBViewPO.retrieve, and then to PViewPO.retrieve, and make sure the **Always complete calling sequence** option is **not** checked.
4. Map Beneficiary's insert, update, and delete methods to iSharedPO.insert, iSharedPO.update, iSharedPO.delete.
5. Map Beneficiary's setConnection method to first iSharedPO.setConnection and then iBViewPO.setConnection.

This creates a new parent along with the child, and deletes the parent along with the child.

If you wanted to create a new child from an existing parent, you could find the existing parent, create a copy of its attribute values, delete the parent, and then create the child as a new object with the values of the deleted parent.

If you wanted to delete the child and leave the parent entry, you could copy the existing parent values, continue with the deletion of the child, and then re-create the parent with the copied values.

6. Click **Finish**.

Adding the managed object

Add BeneficiaryMO:

1. From the business object implementation's pop-up menu, click **Add Managed Object** to open the Managed Object wizard to the Name and Service page.
2. Accept all the defaults and click **Finish**.
3. Click **File > Save** to save your work before continuing to the next step.

Generating the code

You are now ready to generate the code for the objects you have defined.

Generated code will appear in your project's `\Working\platform` directory (for example, `e:\tutorials\inhview\Working\NT`).

1. From the pop-up menu of the User-Defined Business Objects folder, click **Generate > All**.

The code generation can take several minutes.

2. Review the contents of the `Working\platform` directory. All the source files for the component have been generated, and you can now define how to build them.

While you can view the generated source files for a particular object (by selecting **View Source** from its pop-up menu), you cannot edit the source files through Object Builder. If you edit them outside of Object Builder, you should restrict your changes to method implementations, and import your changes back into Object Builder before re-generating.

Configuring the database

You need to define (in DB2) the CBSampDB database, its table (S_DINH), and its views (PView, and BView), which your components will access. You should have a database administrator perform this procedure.

To configure the database, table, and views, you need to enter the following commands from a DB2 command prompt.

```
create database CBSampDB
```

```
connect to CBSampDB
```

```
db2 -t -f S_DINH.sql
```

```
db2 -t -f PView.sql
```

```
db2 -t -f BView.sql
```

```
db2 -t -f VP_DINH.sql
```

As the name of the inheritance pattern (inheritance with views) suggests, there is a single datastore with distinct views for each component (Person and Beneficiary).

Defining a client DLL and server DLL

Configure the build process that will create the client and server DLLs.

While “DLL” is the generic term used in Object Builder, the configuration for the build actually results in whatever the appropriate targets are for the selected platforms. For example, on AIX the build process creates shared library files (lib*.so). If you chose to create a Java business object, then in addition to DLLs there will also be JAR files. The names for these files are derived from the name you provide for the DLL file, within the DLL configuration node.

Start by defining the client DLL configuration and client DLL or library file (for this exercise, name them both clientIN). The client DLL provides client applications with access to the component on the server, using the key and copy helper. You must also include the business object interface, which defines the methods and attributes of the component that the client can access.

1. From the pop-up menu of the Build Configuration folder, click **Add Client DLL** to open the Client DLL wizard to the Name and Options page.
2. Name the configuration. This is the name that uniquely identifies the configuration node.
3. Name the library as well. This is the name for the makefile and for the resulting DLL file. You can create multiple configuration nodes that produce different versions of the same DLL. For example, you could set different compile and link options to produce a development version and a deployment version of the same DLL.
4. For this exercise, the default configuration options are sufficient.
5. Click the title and turn to the Client Source Files page.
6. Select all the client files and add them to the **Items Chosen** list.
7. Click **Finish**.

The clientIN DLL configuration appears under the Build Configuration folder.

Define the server DLL configuration and server DLL file (for this exercise, name them both serverIN). The server DLL is installed on the server to deploy the component, making it available for access by client applications and other components.

1. From the pop-up menu of the Build Configuration folder, click **Add Server DLL** to open the Server DLL wizard.

2. Name the configuration and target DLL file.
3. Click **Next** to turn to the Libraries to Link With page.
4. Add the client DLL file. The server DLL needs access to the client DLL because the client DLL defines the component's interface, and the component implementation inherits from it.
5. Click **Next** to turn to the Server Source Files page.
6. Select all the server source files for the component, and add them to the **Items Chosen** list.
7. Click **Finish**.

The serverIN DLL configuration appears under the Build Configuration folder.

Building the DLLs

To generate the makefiles, based on the configuration choices you made in the DLL wizards:

1. From the pop-up menu of the Build Configuration folder, click **Generate > All > All Targets**. This generates makefiles for all the DLL files defined in the folder and generates an all.mak file that calls the individual DLL makefiles. By choosing **All Targets**, you set the default behavior of the makefile, when it is built. You can still override this default when you build (for example, you can choose to build Java targets only, and override the default of all targets).

To build the DLL and (optionally) JAR files:


1. From the Build Configuration folder's pop-up menu, select **Build > Out-of-Date Targets > C++**.


You always need to generate C++ targets, even if you selected **Java** as your business object implementation language. The other objects of the component (such as the data object implementation and managed object) are still implemented in C++.

2. If you selected **Java** as your business object implementation language, then you also need to build the Java targets. From the Build Configuration folder's pop-up menu, select **Build > Out-of-Date Targets > Java**.

When you have a mixed-language application (some business objects are implemented in C++, some in Java) you must generate both C++ and Java targets for *all* components, even for those implemented in C++. The .jar files for C++ components allow Java components to interact with the C++ components on the server.

For this exercise, your application contains just a single component, so you only need to build the Java targets if you implemented the business object in Java.

 The clientIN.dll and serverIN.dll files are stored in the working\NT\PRODUCTION directory.

 The libclientIN.so and libserverIN.so files are stored in the working/AIX/PRODUCTION directory.

If you have a Java business object, the clientIN.jar and serverIN.jar files are stored in the Working\platform\PRODUCTION directory.

If you had a Java client application, regardless of the language your component is implemented in, you would also select **Build > Out-of-Date Targets > Java Client Bindings** to generate Java client bindings (working\platform\`<build style>\JCB\JCBclientIN.jar`). `<build style>` is one of the configuration directories: NOOPT, PRODUCTION, TRACE, TRACE_DEBUG. See Platform-specific information for more information.

For this exercise, you will be using QuickTest to run and test your application, which has its own build process that includes generation of client bindings.

Defining an application family and application



Define the application family (DataPersistenceApplication). An application family groups a set of applications within System Management.

1. From the pop-up menu of the Application Configuration folder, click **Add Application Family** to open the Application Family wizard.
2. Name the application family.
3. Click **Finish**.

The DataPersistenceApplication application family appears under the Application Configuration folder.

Define the server application (DataPersistenceObjects). An application defines a set of components that will operate together on the server. The application name you provide here is the name that will be used by System Management, when the application and its components are deployed.

1. From the pop-up menu of the application family, click **Add Application** to open the Application wizard.
2. Name the application.
3. Click **Next** to turn to the Additional Executables page.
4. Select the platform you are configuring the application for (for example, **NT Files**).
5. Add the file S_DINH.sql:
 - a. Click **Add Another**.
 - b. Click the **Browse** button to open the Executables to Include dialog.
 - c. Locate your Object Builder working directory.
 - d. Select S_DINH.sql

- e.  Click the **Open** button.
- f.  Click the **OK** button.
6. Add the files `S_DINH.bnd`, `VP_DINH.sql`, `VP_DINH.bnd`, `PView.sql`, `PView.bnd`, `BView.sql`, and `BView.bnd` in the same manner.
7. Click **Finish**.

The `DataPersistenceObjects` application appears under the `DataPersistenceApplication` application family.

Configuring the components with the application

Configure Person's managed object (`PersonFile PersonModule::PersonMO`) with the application (`DataPersistenceObjects`). Create a new container instance (`DataPersistenceContainer`) that will **Throw an exception** when a method is called outside a transaction.

1. From the pop-up menu of the application, click **Add Managed Object** to open the Managed Object Configuration wizard.
2. In the **Managed Objectlist**, select the component's managed object. The other lists should be automatically populated with the appropriate selections.
3. Click **Next** and select the data object implementation. Click **Add Another**. The appropriate data object implementation and DLL will be selected. If there were more than one data object implementation for the component, or it was being built into more than one DLL, you would click **Add Another** again to add the extra information.
4. Click **Next** to turn to the Container page.
5. Check the **Create a new container** option. An extra page is inserted into the wizard, to allow you to name the new container and specify its policies.
6. Click **Next** to turn to the Create New Container page.
7. Name the container, and set its behavior for methods called outside a transaction.

Most of the container's behavior is defined for you, based on the managed object you are configuring, and the data object implementation you selected.
8. Click **Next** and review the home. The appropriate home is already selected, and the default options are acceptable. If there were more than one appropriate home, you could choose the home to use.
9. Click **Finish**.

The `PersonMO` managed object configuration appears under the `DataPersistenceObjects` application, and the new container `DataPersistenceContainer` appears in the Container Definition folder. You can

review the properties of the container, including the service and data access patterns that have been selected for you, by clicking **Properties** from the container's pop-up menu.

Configure Beneficiary's managed object in the same way:

1. From the pop-up menu of the application, click **Configure Managed Object** to open the Managed Object Configuration wizard.
2. In the **Managed Objectlist**, select ViewDataFile
ViewDataModule::BeneficiaryMO.
3. Click the page title and turn to the Container page.
4. Select DataPersistenceContainer. Because Person and Beneficiary are similar components (with the same type of datastore and the same policies), they can use the same container.
5. Click **Finish**.

Generate the application installation information:

1. From the pop-up menu of the Application Configuration folder, click **Generate**.

The DDL that defines the applications for System Management is generated into
Working\platform\PRODUCTION\DataPersistenceApplication.

Close Object Builder:

1. Click **File > Save** to save your work.
2. Click **File > Exit** to close Object Builder.

Summary

You have created a component named Beneficiary that inherits from Person, and defined a shared database table that provides persistence for attributes of both components. You have defined selective views that allow the two components to treat the shared table as separate datastores. You have implemented the views pattern for inheritance with persistence.

You can find information on installing the component on the server in the System Administration Guide, "Configure a New Application Environment" chapter, or online in the topic Installing and configuring a new application.

You can find information on testing the installed component with a QuickTest client application in "Chapter 13. Testing applications with QuickTest" on page 611.

Building a child component

This task covers the main steps necessary to build a component that inherits from another component, already defined in a separate DLL. It does not cover every step; you should first be familiar with the tasks necessary to build a component without inheritance.

To build a child component in Object Builder, follow these steps:

1. Generate the code for the child component.
2. Define a client DLL for the component, or open the properties wizard for an existing client.
3. In the Client DLL wizard, on the Libraries to Link With page, select the import library file for the parent component's client DLL.
4. In the Client DLL wizard, on the Client Source Files page, add the component's client source files.
5. Define a server DLL for the component, or open the properties wizard for an existing server DLL.
6. In the Server DLL wizard, on the Libraries to Link With page, select the import library file for the parent component's server DLL.
7. In the Server DLL wizard, on the Server Source Files page, add the component's server source files.
8. Generate the makefiles for the DLLs.
9. Build the DLLs.

RELATED CONCEPTS

"Chapter 6. Inheritance" on page 299
Components (*Programming Guide*)

RELATED TASKS

"Creating a child component" on page 306
"Configuring builds" on page 549
"Packaging a child component" on page 381

RELATED REFERENCES

"Naming objects" on page 128
"Internationalization of data" on page 132

Packaging a child component

This task covers the main steps necessary to package a component that inherits from another component already defined in Object Builder. It does not cover every step; you should first be familiar with the tasks necessary to package a component without inheritance.

To package a child component in Object Builder, follow these steps:

1. Create the application family.
2. Add the client application. On the Additional Executables page, select the client executable, the component's client DLL, and the parent component's client DLL.
3. Add the server application.
4. Configure the component's managed object with the server application.
5. Generate the DDL.

RELATED CONCEPTS

"Chapter 6. Inheritance" on page 299
Components (*Programming Guide*)

RELATED TASKS

"Creating a child component" on page 306
"Packaging applications" on page 574

Chapter 7. Working with external files

External files for method bodies

Most of the editing you do in Object Builder is of method bodies. You can either create a method body in Object Builder using the Source pane editor, or define the method body in an external file that will get pulled into the generated code for the object.

You can select to use an external file for a particular method in its Method Implementation wizard. First click on the business object implementation to display its methods in the Methods pane. Then from the pop-up menu of the method in the Methods pane, click **Properties** to display the wizard. By default, new external files will be placed in the current project's \Model directory. The advantage of using external files is that you can do more work outside of Object Builder. You can edit the external files with your preferred editor, and then use the obgen command (with the -change option), outside of Object Builder, to generate the code for the relevant objects and pull together the code from the external file.

An alternative to the use of external files is direct editing of method bodies in the generated source files. You can enter your modifications of method bodies between the lines:

```
// <GeneratedMethodBody>
// <Body origin="..." xmi.uuid="...">
// Insert method modifications here
...
// End method modifications here
// </Body>
// </GeneratedMethodBody>
```

within a generated source file that includes method bodies. If you edit a framework method, you must also change the Body origin value to "user".

While this removes the need to use obgen to pull in your changes, you do need to remember to import the changes back into Object Builder before you generate code again, or your changes will get over-written.

Template Files

Another advantage of external files is that you can use the same external file for multiple methods, by putting macros in the file and identifying the file as a template. Macros are identified within the file by the delimiter \$.

For example, given the following template file:

```
GenericMethodBody.template
char* str = "This is a method of $classname$";
cout << str << endl;
return ::CORBA::string_dup (str);
```

In a method's wizard, for example the deny() method of a ClaimBO, specify the external file GenericMethodBody.template, specify that it is a template file, and on the next page, add a Template File Macro with the name classname and the substitution value ClaimBO. When you generate the code for ClaimBO, it contains a method body something like this:

```
::CORBA::Void ClaimBO_Impl::deny()
{
//Version identifier DCE:F3F30755-6F47-11d2-AF4E-000629B3CFEE:1
// Insert Method modifications here
char* str = "This is a method of ClaimBO";
cout << str << endl;
return ::CORBA::string_dup (str);
// End Method modifications here
}
```

You could then access the same method body from another method, for example SpecialClaimBO::deny, and substitute a different class name in the macro (for example SpecialClaimBO).

The macro is defined as a simple string, and Object Builder will not recognize it as a reference to any existing elements. In the above example, if you changed the name of ClaimBO (for example, to EGClaimBO), you would need to manually update the macro string to match.

RELATED CONCEPTS

- "User-defined methods" on page 751
- "Get and set methods" on page 755
- "Framework methods" on page 757
- "Special framework methods" on page 758
- "Push-down methods" on page 759
- "External files for method bodies" on page 383

RELATED TASKS

- "Importing edited source files" on page 385
- "Generating code from the command line" on page 684
- "Implementing methods" on page 752

Importing edited source files

If you make changes to method implementations in the generated source code, you need to import the changes back into Object Builder, or the changes will be overwritten the next time you generate code.

When you import edited code, only changes to method implementations are applied. The import process recognizes method implementations by the comment block that delimits them. The comment block varies depending on whether it is a user-defined method or a framework method.

User-defined method

```
// <GeneratedMethodBody>
// <Body origin="user" xmi.uuid="...">
// Insert method modifications here
...
// End method modifications here
// </Body>
// </GeneratedMethodBody>
```

Framework method

```
// <GeneratedMethodBody>
// <Body origin="ob" xmi.uuid="...">
// Insert method modifications here
...
// End method modifications here
// </Body>
// </GeneratedMethodBody>
```

If you edit a framework method, you must also change the Body origin attribute to “user”. Otherwise your changes will be ignored.

When you import the framework method, the existing body will be compared with the imported body. If the two are identical, there is no change. If the two are different, then the imported body replaces the existing one, and the framework method’s property is set to **Use the implementation defined in the Source pane**(as defined in the method’s Method Implementation wizard in Object Builder).

The comment block is inserted by the code generation process. Any changes you make outside of these generated comment blocks, or inside a comment block with <Body origin=“ob”>, are ignored.

To import code you have edited, follow these steps:

1. From the pop-up menu of either the User-Defined Business Objects folder or the User-Defined Data Objects folder, click **Import > Changes** to open the Import Changes wizard.
2. From the **Available files** list, select those that contain changes to method bodies that you want to import.
3. Click >> to move the files to the **Files to be imported** list.
4. Click **Finish**. The method body changes are applied.
5. Make any other changes necessary (for example, if the interface of a method has changed, you need to change its definition in the Business Object Interface wizard).

RELATED TASKS

“Importing edited source files from the command line” on page 682

“Editing a business object implementation” on page 792

“Generating code” on page 551

“Implementing methods” on page 752

RELATED REFERENCES

“importimpl” on page 683

Importing C++ or Java classes



If you have an existing C++ or Java class that you want your component to use, and you do not want to create an IDL interface for it, you can import the class into Object Builder as a non-IDL type.

Once the class is imported, you can select it as an interface type within Object Builder (that is, as a method return type, method parameter, or attribute type).

Note: Because this is not an IDL type, it cannot be accessed through the distributed environment. The component’s managed object will not expose methods or attributes that use this type. Methods or attributes that use the type should be added to the business object implementation, not the business object interface.

To import a non-IDL type, follow these steps:

1. In the Tasks and Objects pane, find the Non-IDL Type Objects folder.
2. From the folder’s pop-up menu, click **Import Non-IDL Type**. The Import Non-IDL Type wizard opens to the Name and Language page.
3. Type the name of the class you want to import.

For a Java class, if the class is defined in a package, you must specify the full class name including the package name. When the code is generated,

the names of non-IDL types are fully qualified in the body (there is no import statement for them).

For example:

► C++ MyCppClass

► JAVA java.util.Hashtable (where java.util is the name of the package, and Hashtable is the class name)

4. Select whether the class is implemented in C++ or Java.
5. If you selected C++, then click **Next**, otherwise click **Finish**.
6. Provide implementation details for the class.

For a C++ class, provide the name of the header file that defines the class and the name of the library file that contains its object code. Include the file extensions.

7. Click **Finish**.

The class now appears in the Non-IDL Type Objects folder, and you can select it as the type of a method parameter, a method return type, or an attribute type.

RELATED CONCEPTS

Programming languages and conventions (*Programming Guide*)

RELATED TASKS

“Adding a business object interface” on page 777

Exporting XML

You can export the data in a project’s model in XML format. The exported files are named according to the structure of the Tasks and Objects pane. Within each folder in the pane, you can export XML for the top-level elements. The exported file names are based on the folder name plus element name: for example, udbo.ClaimFile.xml defines the Claim business object layer in the User-Defined Business Objects folder. You can export either from within Object Builder, or from a command line. Once you have exported, you can import the data into another project’s model.

The files you can export are described in “XML interchange files” on page 493.

To export the XML for a project:

1. Open the project.
2. Click **File > Export Model**.
3. Select whether to export to a single file or to multiple files.

Generally you should export to multiple files. For the purposes of change control or changing project divisions, you should always export to multiple files.

4. Click **Finish**. Files are exported to the \Export subdirectory.

To export the XML for a folder (any folder except for Framework Interfaces or Default Homes):

1. From the folder's pop-up menu, click **Export** to open the Export XML wizard.
2. Click **Finish**. The appropriate XML files are exported to the \Export subdirectory.

To export the XML for a component layer:

1. From the pop-up menu of a file or top-level object within a folder (any folder except for Framework Interfaces or Default Homes), click **Export** to open the Export XML wizard.
2. Click **Finish**. The appropriate XML file is exported to the \Export directory, as described in "XML interchange files" on page 493.

The exported XML files are placed in the project's \Export directory, and named according to the source folder for the exported items. For example, if you exported the contents of the Application Configuration folder, the exported files all start with udaf. If you exported the business object layer for a component with the business object file name ClaimFile, the exported file is udbo.ClaimFile.xml.

Once you have exported the file, you can import it into another project. The file conforms to the DTD (document type definition) eom.dtd for Object Builder models.

RELATED CONCEPTS

"Model interchange with XML" on page 492

RELATED TASKS

"Maintaining a team environment " on page 490

"Exporting XML from the command line" on page 660

"Importing XML" on page 389

RELATED REFERENCES

"XML interchange files" on page 493

"obexport" on page 661

Importing XML

You can import XML that has been exported from Object Builder. This allows you to transfer information from one project model to another.

XML can be imported from the File menu. Click **File > Import Model**, and select XML files to import.

The exported XML conforms to a DTD (document type definition) for Component Broker models: eom.dtd. When you import an XML file, it is parsed and checked against the DTD. Only XML that conforms to the DTD can be imported.

The current project must contain the information necessary to resolve any references in the XML file, or the references will not be imported (the rest of the XML will be).

For example:

- If you are importing XML that defines a child component, the child's parent component must already be defined in the current project.
- If you are importing XML that defines application configuration, the managed objects configured with the application must already be defined in the current project.
- If you are importing XML that defines a component Customer that references a component Account, the referenced Account component must already be defined in the current project.

To import XML, follow these steps:

1. Click **File > Import Model**. The Import XML wizard opens to the File Selection page.
2. Specify the files you want to import. Unless you have moved them, they are in the \Export subdirectory of the project you exported them from.
3. Click **Finish**.
The data in the XML files is loaded into the current project.
4. Select **File > Save**. The newly imported data is saved to the project's model.

You can also import XML from the command line, with additional options. If you are importing multiple files with circular references, or importing into multiple projects, you should import from the command line with the `-X` option.

RELATED CONCEPTS

"Model interchange with XML" on page 492

RELATED TASKS

"Maintaining a team environment " on page 490

"Exporting XML" on page 387

"Importing XML from the command line" on page 663

RELATED REFERENCES

"XML interchange files" on page 493

"obimport" on page 664


Chapter 8. Working with enterprise beans

An enterprise bean is a Java component that can be combined with other enterprise beans, and other Java components to create a distributed, three-tiered application. You can create enterprise beans using VisualAge for Java, and then import them into Object Builder for deployment by importing the EJB JAR file that encapsulates them.

The following tasks deal with enterprise beans:

- “Importing enterprise beans into Object Builder” on page 392
- “Deploying enterprise beans” on page 408

Note: VisualAge for Java supports composers and converters. Composers are not supported for enterprise beans that are deployed using Object Builder and the Component Broker MOFW (CORBA) infrastructure. However, some converters are supported - not physically, but in concept (in the form of its various attribute mapping patterns, which include the specification of arbitrary, user-defined mappings). This infrastructure is used for deploying enterprise beans only in this version of WebSphere. See the reference topic ‘Java to Object Builder type mappings’, which lists the default type conversions that take place during enterprise bean deployment.

 390 Enterprise bean support is available on the Windows NT, AIX, Solaris, and HP-UX deployment platforms. It is also available for OS/390. However, Component Broker and WebSphere EJB clients on platforms other than CB OS/390 will not be able to exchange information with CB OS/390 enterprise beans. Neither will CB and WebSphere EJB clients that are on CB OS/390 be able to exchange information with enterprise beans on other platforms.

RELATED CONCEPTS

The EJB JAR file (*Writing Enterprise Beans in WebSphere*)

An introduction to enterprise beans (*Writing Enterprise Beans in WebSphere*)

RELATED REFERENCES

“Java to Object Builder type mapping” on page 394

Importing enterprise beans into Object Builder

▶CHANGED

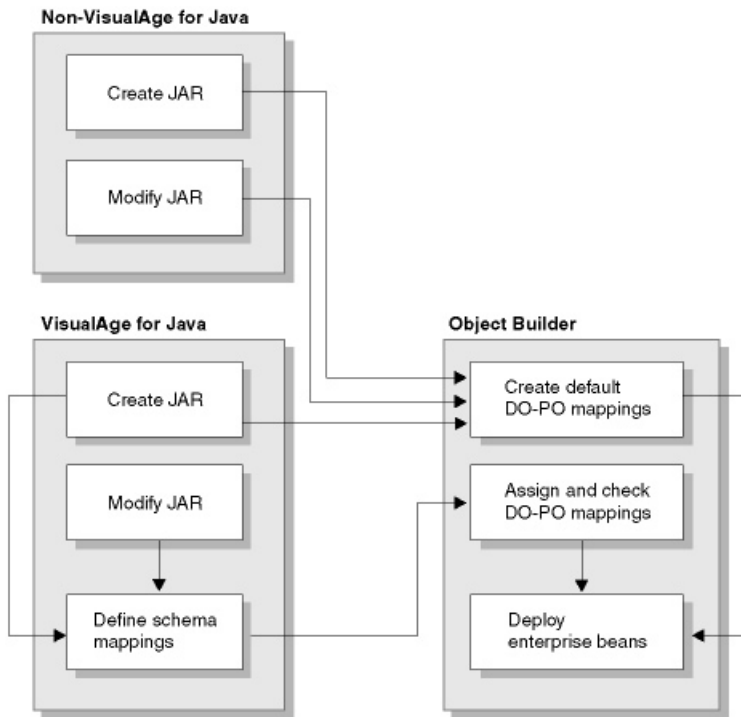
Enterprise beans encapsulate the business logic and data used and shared by EJB clients. Enterprise beans are contained in EJB JAR files, which you can import into Object Builder.

You can import enterprise beans into Object Builder in one of the following ways:

- “Importing enterprise beans using Object Builder” on page 401
- “Importing enterprise beans from the command line” on page 670
- “Importing enterprise beans from VisualAge for Java” on page 405
From VisualAge for Java, you can export an EJB JAR file that contains any type of enterprise bean, directly into Object Builder. You can also define the mapping between the data object and the persistent object in VisualAge for Java.

▶390 Enterprise bean support is available on the Windows NT, AIX, Solaris, and HP-UX deployment platforms. It is also available for OS/390. However, Component Broker and WebSphere EJB clients will not be able to exchange information with CB OS/390 enterprise beans.

Whichever method you use to import the beans, it follows the deployment flow as in the diagram:



For enterprise beans that are created using a tool other than VisualAge for Java:

If you have either create the JAR file, or modify an existing one, and then import it into Object Builder, it undergoes the following processes:

1. Object Builder creates default data object to persistent object mappings for the beans that you choose to deploy
2. It then deploys those beans

For enterprise beans that are created using VisualAge for Java:

If you create a JAR file for which you define schema mappings, and then import it into Object Builder, it undergoes the following processes:

1. Object Builder checks these mappings, and translates them into corresponding data object to persistent object mappings
2. It then deploys the beans

If you create a JAR file for which you do not define schema mappings, and then import it into Object Builder, it undergoes the following processes:

1. Object Builder creates default data object to persistent object mappings for the beans that you choose to deploy

2. It then deploys those beans

If you modify a JAR file by redefining its schema mappings, and then import it into Object Builder, it undergoes the following processes:

1. Object Builder checks these mappings, and translates them into corresponding data object to persistent object mappings
2. It then deploys the beans

RELATED CONCEPTS

The EJB JAR file (*Writing Enterprise Beans in WebSphere*)

An introduction to enterprise beans (*Writing Enterprise Beans in WebSphere*)

RELATED TASKS

“Importing enterprise beans using Object Builder” on page 401

“Importing enterprise beans from the command line” on page 670

“Importing enterprise beans from VisualAge for Java” on page 405

Java to Object Builder type mapping



The existing VisualAge for Java tooling supports composers and converters. Composers are not supported for enterprise beans that are deployed using Object Builder and the CB MOFW (CORBA) infrastructure. However, some converters are supported in concept, and this infrastructure is used for deploying enterprise beans only in this version of WebSphere.

The physical VisualAge for Java or VisualAge Persistence (Persistence Builder, or VAP) converters will not be used when an enterprise bean is deployed using Object Builder. However, Object Builder does provide some type conversion capability, using a C++ data object. Some predefined mapping helpers exist, and ship with Object Builder, or you can define and use your own.

The following table shows the default type conversion (the supported converter mappings), with the types that will be created in Object Builder:

Java type	IDL type	SQL type
int	long	INTEGER
short	short	SMALLINT
long	long long (workstation) long (CB/390)	BIGINT (Note: not yet supported by Query. Fallback type is INTEGER)
char	char	CHAR(1)
String	string (wstring)	VARCHAR or CHAR (VARGRAPHIC or GRAPHIC)
byte	octet	SMALLINT

Java type	IDL type	SQL type
byte[]	bytestring (sequence<octet>)	VARCHAR FOR BITDATA
boolean	boolean	SMALLINT
float	float	REAL
double	double	DOUBLE

The following types do not have a direct mapping, but can be converted relatively easily. They will require this conversion in the tie bean, and in the data object.

Java type	IDL type	SQL type
Java.sql.Date	String	DATE
Java.sql.Time	String	TIME
Java.sql.Timestamp	String	TIMESTAMP
Java.math.BigInteger	String	DECIMAL
Java.math.BigDecimal	String	DECIMAL

Note: User-defined types, such as Java classes that you provide, and VAP composers and persisters, will not be automatically mapped in this version of Object Builder version. Any type other than those listed above will be serialized in the tie bean, and persisted as a binary VARCHAR FOR BIT DATA field in the database.

RELATED TASKS

“Chapter 8. Working with enterprise beans” on page 391

Keys for enterprise beans



In general, a primary key must have the least number of attributes (state data), which are used to uniquely identify an instance of a business object. All the attributes of the key must be set by the client to ensure uniqueness, but at least one attribute of the key must be set if the key is to be used on the server. Like a copy helper, a key is a local-only object, which means it is not accessible remotely. A key usually has a subset of the attributes defined for the copy helper associated with the same business object.

In the Managed Object Framework (MOFW), the key, if it is defined for a relational database, is made up of some subset of the fields that are in the externalized IDL interface.

For an enterprise bean, a key is made up of some subset of container-managed persistence (CMP) fields. However, there are cases where the keys are not related to CMP. The CMP fields usually represent the same fields that are in the remotable interface to the enterprise bean. However, very often, in addition to these field representations, there are additional CMP attributes that are part of the key. This is to support encapsulation.

Enterprise beans do not expose query to client programs, but instead enable query of all CMP fields. They encapsulate this logic in finders. This differs from the Component Broker MOFW, where the ability to write arbitrary predicates is exposed to clients and servers by means of homes and query evaluators.

Object Builder, in keeping with the MOFW, now supports two styles of primary keys (and copy helpers):

- Those consisting strictly of attributes from a business object client interface
- Those consisting of attributes from a single business object server implementation and, optionally, from the business object client interface

When the key includes attributes only from the business object interface, you can use it to map to the remote EJB interface. When it includes attributes from a business object implementation as well, you can use it to map to the EJB object.

Modeling EJB keys that have attributes that are not defined on the remote EJB interface

When you define keys and copy helpers in Object Builder, you can select a business object implementation so that its attributes can be used in the definition of these objects. When you select a business object implementation, your choice of attributes is not limited to those that are defined on the implementation only; you can also select the attributes that are defined on the related business object interface. However, you do not have to use attributes from the business object interface. This follows the model of the EJB keys that contain attributes that are not defined on the remote EJB interface.

RELATED TASKS

“Adding a key” on page 826

“Editing a key” on page 828

“Adding a copy helper” on page 830

“Chapter 8. Working with enterprise beans” on page 391

CMP Entity Bean-Specific Settings



This section has entries for container-managed beans that are not sessional. Details include the type of backend storage, whether the bean is queryable,

whether it is one that is created by importing a PA bean using procedural adaptors, and whether it uses *wstring* to map to the bean-specific Java *string* type, in its corresponding data object.

Backend Storage Type


The backend storage type for the enterprise bean's persistent data can be one of the following data stores:

- DB2 V5.2 Embedded SQL
- DB2 V6.1 Embedded SQL
- DB2 V5.2 Cache Service
- DB2 V6.1 Cache Service
- Oracle Cache Service
- Informix Cache Service
- HOD - Procedural Adaptors
- ECI - Procedural Adaptors
- LU6.2 - Procedural Adaptors
- EXCI - Procedural Adaptors
- OTMA - Procedural Adaptors

Select one from the list.

Note: If you do not specify a backend storage type, DB2 V5.2 Embedded SQL is taken as the default.

Restrictions:

 If the deployment platform is 390, you can select only from the following options:

- DB2 V5.2 Embedded SQL
- DB2 V6.1 Embedded SQL
- EXCI - Procedural Adaptors
- OTMA - Procedural Adaptors

Informix If you are using the Informix Cache Service, a given transaction will not be able to access more than one Informix database per CB server. To involve two Informix databases in a transaction, you must access each database from a different server.

Home Type

You can select the type of the home to be generated. You have the following choices:

- Regular home
- Queryable home

- Polymorphic home

These options can be used only for CMP entity beans that store their persistent data in a relational database. That is, you can select one of these home types only if you have selected one of the following backend data storage types:

- DB2 V5.2 Embedded SQL
- DB2 V6.1 Embedded SQL
- DB2 V5.2 Cache Service
- DB2 V6.1 Cache Service
- Oracle Cache Service
- Informix Cache Service

Regular home

This is the default. A regular home is always generated if you do not specify the type.

Queryable home

Select this option to direct the tool to generate a queryable Component Broker (CB) home object. This option must be used if the finder helper class, which is used to implement the finder methods in a CMP entity bean, uses the CB Query Service. This option must not be used if an entity bean uses CICS or IMS to store its persistent data. When you designate the home to be queryable, the generated business object interface is automatically marked as queryable.

Note: If you select either Oracle Cache Service, or Informix Cache Service, you must select the **Queryable home** option.

Polymorphic home

Select this option if you want to deploy the enterprise beans into a polymorphic home. It directs the tool to generate a polymorphic Component Broker (CB) home object. This option must be used if the finder helper class, which is used to implement the finder methods in a CMP entity bean, uses the CB Query Service. This option must not be used if an entity bean uses CICS or IMS to store its persistent data. When you designate the home to be polymorphic, the generated business object interface is automatically marked as polymorphic (that is, it inherits from `IManagedClient` `IManagedClient::IPolymorphicHome`).

Note: You must use either this option, or the **Queryable home** option if the finder helper class, which is used to implement the finder methods in a CMP entity bean, uses the CB Query Service.

Bottom-up PAA

This check box is selected, and cannot be edited if the enterprise bean that you selected for deployment was generated using the **PAOToEJB** tool. The enterprise bean in this case, is created as a wrapper for the Enterprise Access Builder business object, which is itself a wrapper for a procedural adaptor object (one that is usually imported into Object Builder as a PA bean).

Workload management

Select this check box if workload management is to be used in the deployment of the enterprise bean. When you select this option, Object Builder marks the CMP entity bean to be used in a workload manager-enabled container.

The workload management service improves the scalability of the EJB server environment by grouping multiple EJB servers into server groups. Clients can access these server groups as if they are a single EJB server, and the workload management service ensures that the workload is evenly distributed across the EJB servers in the server groups. An EJB server can belong to only one server group.

Use wstring in data object

Directs the tool to map the container-managed fields of an entity bean (which are of Java *string* type) to the *wstring* IDL type (rather than the *string* IDL type) on the data object. It is preferable to map to the *string* IDL type if the data source contains single-byte character data; it is preferable to map to the *wstring* IDL type if the data source contains double-byte or Unicode character data.

RELATED CONCEPTS

Entity beans (*Writing Enterprise Beans in WebSphere*)

The deployment descriptor (*Writing Enterprise Beans in WebSphere*)

The EJB JAR file (*Writing Enterprise Beans in WebSphere*)

Workload management (Using Object Builder) (*Advanced Programming Guide*)

RELATED TASKS

“Importing enterprise beans into Object Builder” on page 392

“Deploying enterprise beans” on page 408

Developing and deploying enterprise beans with EJB server (CB) tools
(*Writing Enterprise Beans in WebSphere*)

BMP Entity Bean-Specific Settings

This section has entries for bean-managed beans that are not sessional. These are the options:

Workload management

If this option is selected, Object Builder generates a home interface that is workload manager-enabled.

The workload management service improves the scalability of the EJB server environment by grouping multiple EJB servers into server groups. Clients can access these server groups as if they are a single EJB server, and the workload management service ensures that the workload is evenly distributed across the EJB servers in the server groups. An EJB server can belong to only one server group.

JDBCAA

Select this check box if the BMP entity beans require JDBC (Java Database Connectivity), along with the ability to carry out distributed transactions. The JDBC application adaptor handles distributed transactions by enabling the bean implementation to connect to the CB Transaction Service. If you do not select this option, the BMP beans handle persistence by themselves: they may or may not use JDBC.

RELATED CONCEPTS

Entity beans (*Writing Enterprise Beans in WebSphere*)

The deployment descriptor (*Writing Enterprise Beans in WebSphere*)

The EJB JAR file (*Writing Enterprise Beans in WebSphere*)

Workload management (Using Object Builder) (*Advanced Programming Guide*)

RELATED TASKS

“Importing enterprise beans into Object Builder” on page 392

“Deploying enterprise beans” on page 408

Developing and deploying enterprise beans with EJB server (CB) tools
(*Writing Enterprise Beans in WebSphere*)

Session Bean-Specific Settings

This section has information for session beans that are sessional. You can view the following settings:

Workload management

If this option is selected for a stateful session bean, Object Builder generates a home interface that is workload manager-enabled. If this option is selected for a stateless session bean, it indicates that the bean is to be used in a workload manager-enabled container.

RELATED CONCEPTS

Session beans (*Writing Enterprise Beans in WebSphere*)

The deployment descriptor (*Writing Enterprise Beans in WebSphere*)

The EJB JAR file (*Writing Enterprise Beans in WebSphere*)

Workload management (Using Object Builder) (*Advanced Programming Guide*)

RELATED TASKS

“Importing enterprise beans into Object Builder” on page 392

“Deploying enterprise beans” on page 408

Developing and deploying enterprise beans with EJB server (CB) tools

(*Writing Enterprise Beans in WebSphere*)

Importing enterprise beans using Object Builder

Enterprise beans are contained in EJB JAR files, which you can import into Object Builder. You can import both session and entity beans into Object Builder using this method.

To import enterprise beans, follow the steps in the task “Creating a deployed EJB JAR file” on page 414.

When you click **Finish** from the Import EJB JAR wizard, XML is imported into Object Builder. The EJB JAR file is represented as a node in the Enterprise Beans folder, with a class corresponding to each of the enterprise beans that you selected to be deployed, as a separate node beneath it.

The Tie class, and the IDL and Java files corresponding to the imported JAR file are created in the Working\<platform> directory.

The data object and its implementation are created in the User-Defined Data Objects folder. The IDL attributes of the data object interface correspond to the entity bean’s container-managed fields. The User-Defined Business Objects folder has the associated business object, the key, the copy helper, and the managed object.

CMP entity beans

For CMP entity beans the default mappings between the data object and the persistent object are created when you import the beans into Object Builder. You can override these mappings, and provide your own.

Session beans, or BMP entity beans

Normally, you do not have to make any additions to your model for these beans. But, in the case of session beans that are associated with an MQSeries application adaptor-backed business object, you must follow this step:

1. Complete the definition of the data object implementation for the business object that is associated with the session bean.

The business object referred to is either an existing one that you specified when you created the session bean using the **mqaajb** tool, or one that you created while running the tool. See the section: Creating an enterprise bean that communicates with MQSeries in *Writing Enterprise Beans in WebSphere*.

You can select the **Do All** option from the **File** menu to run the Consistency Checker on your model.

RELATED CONCEPTS

Entity beans (*Writing Enterprise Beans in WebSphere*)

Session beans (*Writing Enterprise Beans in WebSphere*)

The EJB JAR file (*Writing Enterprise Beans in WebSphere*)

An introduction to enterprise beans (*Writing Enterprise Beans in WebSphere*)

RELATED TASKS

“Chapter 8. Working with enterprise beans” on page 391

“Deploying enterprise beans” on page 408

Enabling transactions and security in enterprise beans (*Writing Enterprise Beans in WebSphere*)

Setting the transaction attribute (*Writing Enterprise Beans in WebSphere*)

Creating an enterprise bean that communicates with MQSeries (*Writing Enterprise Beans in WebSphere*)

The EJB Deployment Tool

The EJB Deployment Tool works with Object Builder to create and compile the files required by the EJB server (CB) to manage an enterprise bean. The EJB Deployment Tool introspects the EJB JAR file, paying attention to the EJB home, the EJB object classes and the deployment descriptors. The EJB Deployment Tool generates a model that Object Builder uses to create the necessary deployment library files. The output of this process is a set of server side and client side JAR and library files.

The EJB Deployment Tool deploys enterprise beans by generating extensible markup language (XML) files and importing those files into Object Builder. If the XML import fails, you can view any error messages generated by Object Builder in the `import_model.log` file located in the project directory.

When you deploy enterprise beans with the EJB Deployment Tool, a Component Broker data object IDL interface is created. The IDL attributes of this interface correspond to the entity bean’s container-managed fields. The EJB Deployment Tool maps the container-managed fields of entity beans to data object IDL attributes, and it also maps these IDL attributes to the entity bean’s data source. You can use an existing data source such as a DB2, Oracle or Informix database for meet-in-the-middle deployment, or you can define a new one for top-down deployment.

By default, Object Builder creates the persistent objects and schemas for deployed enterprise beans. If a user is deploying an enterprise bean from VisualAge for Java version 3.5, the schema mappings created in VisualAge for Java are preserved. For JAR files that are created using earlier versions of

VisualAge for Java, and for those created using other tools, Object Builder creates the default persistent objects and schemas. The default mappings that are created will be similar to those created when you do top-down development.

RELATED CONCEPTS

The EJB JAR file (*Writing Enterprise Beans in WebSphere*)

RELATED TASKS

“Importing enterprise beans from the command line” on page 670

“Deploying enterprise beans” on page 408

RELATED REFERENCES

“cbejb options” on page 673

Importing enterprise beans from the command line

After you have installed the EJB deployment tool, from a command prompt, invoke it using the *cbejb* command, specifying the EJB JAR file that contains the enterprise beans that you want to deploy, the project directory within Object Builder that you want to use for the beans, the names of the beans that you want to deploy, and whether you want to use the graphical user interface of Object Builder to make deployment-specific changes (*-guisg*).

Type:

```
cbejb ejb-jarFile [-ob projDir] [-bean beanNames] [-guisg]
```

For CMP entity beans, and MQSeries application adaptor-backed beans

If you specify *-guisg*

After the data object to persistent object mapping is done, (and if you had not specified *-noursraction* with *cbejb*), you are presented with a dialog box from which you can select different actions to be taken:

- launch Object Builder if you want to change the default mappings between the data object and the persistent object;
- continue with code generation and build; or
- stop the deployment without generating and building code.

If you do not specify *-guisg*

After the data object to persistent object mapping is done, (and if you had not specified *-noursraction* with *cbejb*), you are prompted for the next action (at the command line):

- *x*, to cancel out of the deployment process;
- *c*, to continue with model check, code generation and build; or
- *o*, to launch Object Builder (most often if you want to change the default mapping between the data object and the persistent object).

When you exit Object Builder, after you have deployed either CMP entity beans, or session beans that are associated with an MQSeries application adaptor-backed business object, the **Do All** dialog box appears. You can specify further actions to be taken: run the Model Consistency Checker, generate all code for model, build targets, or exit Object Builder. You also have the option of changing the default checks that the Consistency Checker will perform (click the **Checker Options** button, and select a different set of options, if you want to). Click **Start** to run the Model Consistency Checker, generate code, and compile it (if you have selected the corresponding check box options).

After the data object to persistent object mapping is done, (and if you had specified `-noursaction` with `cbejb`), the deployment flow (codegen and build, if specified previously using the respective `cbejb` options) will continue by means of the command line only.

For BMP entity beans, and session beans

If you had either specified `-guisg`, and used the graphical user interface of Object Builder, and completed the task of importing the beans using the wizards; or if you did not specify `-guisg`, but specified all other options (you chose to deploy the beans entirely using the command line); the deployment flow continues uninterrupted, with code generation and build using the command line.

Using Object Builder with the EJB Deployment Tool

If you use the `-guisg` option, Object Builder's graphical user interface is displayed, and you can use it to import the beans into Object Builder for deployment.

Follow these steps:

1. On the first page (the Enterprise Bean Selection page) of the Import EJB JAR wizard, specify the EJB JAR file that you want to import, and the project directory for the imported enterprise beans.
2. Indicate whether you want code to be generated, and then built for the objects that are created at the time of the import.
3. Select which of the imported enterprise beans in the JAR file have to be deployed.
4. Click **Finish**. The Import EJB JAR wizard opens to the EJB Browser page. You can use this page to view, or edit the settings of the beans to be deployed.
5. Click **Properties**. The Deployment Information page opens.
6. Except for the deployment platforms, you can edit all other information on this page. You can specify a finder helper class name (the name of the public interface class that contains the definitions and initializations of the

enterprise bean's finder methods, which are used to find entity bean objects); name an application family into which the beans are to be configured; specify the name of the DLL that is to be associated with the application family; and for CMP beans, specify a database to be associated with the beans. You can also add or delete client and server JAR file dependencies, and specify whether the bean is to be deployed into a regular home, a queryable home, or a polymorphic home. Besides, you can edit information specific to the bean type.

7. Click **Finish**.

During deployment, a deployed JAR file is generated from an EJB JAR file. The enterprise beans are deployed in the EJB server (CB) environment. The deployed JAR file (which is represented by a node in the EJB Folder in the Tasks and Objects pane of Object Builder) contains classes required by the EJB server.

The EJB deployment tool also generates the data definition language (DDL) file used during installation of the enterprise bean into the EJB server (CB).

RELATED CONCEPTS

The EJB JAR file (*Writing Enterprise Beans in WebSphere*)

An introduction to enterprise beans (*Writing Enterprise Beans in WebSphere*)

The deployment descriptor (*Writing Enterprise Beans in WebSphere*)

RELATED TASKS

"Chapter 8. Working with enterprise beans" on page 391

"Importing enterprise beans into Object Builder" on page 392

Deploying enterprise beans into a polymorphic home

RELATED REFERENCES

"cbejb options" on page 673

Importing enterprise beans from VisualAge for Java



You can export session, BMP, and CMP beans from VisualAge for Java, directly into Object Builder.

In VisualAge for Java, once you have finished creating and testing your enterprise beans, follow these steps:

1. In the **Enterprise Beans** pane of the **EJB** page, select the EJB group that contains the enterprise beans that you want to import.
2. From the **EJB** menu, select **Export**, and then **EJB JAR for CB**.
3. The **Export to an EJB JAR File for Component Broker** wizard opens.
4. Specify the name of the EJB JAR file and its contents, as well as other options such as debugging, compressing, or overwriting information.

5. Click **Finish**.

The Import EJB JAR wizard of Object Builder opens to the Enterprise Bean Selection page. Follow these steps:

1. Using the Enterprise Bean Selection page, specify the project directory for the imported enterprise beans. The default project directory is `x:\tmpCBDeploy`, where `x` is the directory from which you launched VisualAge for Java. You must provide a different directory name each time you import a JAR file, unless you are redeploying the beans. Otherwise, information will be overwritten. You can indicate whether you want code to be generated, and then built for the objects that are created at the time of the import. Select which of the imported enterprise beans in the JAR file have to be deployed.
2. Click **Next**. The EJB Browser page opens.
3. You can use this page to view, or edit the settings of the beans to be deployed. Select the bean from the **Enterprise Beans to Be Deployed** list, and click the **Properties** button.
4. The Deployment Information page opens. Except for the deployment platforms, you can edit all other information on this page. You can provide a finder helper class name, a database name, and add or delete client and server JAR file dependencies.

Note: If you created the beans that you imported using VisualAge for Java, do not use the finder helper class that is generated by the tool. Instead, you must use the **FinderHelperGenerator** utility of the EJB server in CB to implement the finder helper class. For example, to generate a finder helper class for the AccountHome interface, use the command:

```
# ejbfhgen com.ibm.ejs.doc.account.AccountHome
```

This command generates the finder helper class named `com.ibm.ejs.doc.account.AccountHome`.

For more information on finder helper classes, see *Defining finder methods*, and in particular, *Creating finder logic in the EJB server (CB) in Writing Enterprise Beans in WebSphere*.

5. Click **Finish**.

A window is launched.

The batch file and the response file

During import, two files: a batch file (`.bat` extension), and a response file (`.rsp` extension) are created in your project directory. They take their names from the name of the JAR file that you are importing. For example if the name of the EJB JAR file is `EJBHotel.jar`, the names of these files will be `EJBHotel.bat`, and `EJBHotel.rsp`.

Both these files contain the options that you select as you import the beans for deployment.

The batch file sets the local class path, and adds all the client and server JAR file dependencies to your class path. It has the cbejb command used along with the process options (options used with the cbejb command to generate and compile code).

The response file contains the deployment options that you specify for the bean that you select to import and deploy (these include any options that you selected as you imported the bean, except for the process options). When you import enterprise beans subsequently from the same JAR file, the batch and the response files are regenerated, and they overwrite the existing files.

The window shows the class path and the cbejb command that is used for deployment. You also see the response file used with the command. You will notice that the cbejb command is invoked with -nouseraction. You are not prompted for any action after the data object to persistent object mapping is done. Object Builder is not launched. The deployment flow (codegen and build, if specified as process options) will continue by means of the command line only. However, from the command line, you can bring up Object Builder if you want to either view or modify your model.

This window is non-interactive, but you can save the output in a file for later reference.

Recommendations:

- Do not close this button using the Close (X) button in the title bar. Instead, close it by terminating and removing your program from VisualAge for Java's Console window.
- Before you deploy another JAR file, refresh VisualAge for Java's Console window: select **Programs > Terminate and Remove**.
- If you prefer to comment on the output of deployment as it progresses, rather than later, instead of saving the output from the non-interactive window, use VisualAge for Java's Console window itself: as the output scrolls in the **Output** panel, type your comments in the **Standard In** panel. You can then save this documented output of deployment using **File > Save As**.

Note: When you deploy beans using the EJB Deployment Tool (cbejb), you can make use of this same response file that is generated (specifying it with the -rsp option for the cbejb command), if you are specifying different process options. You can also use these files when you are redeploying the same JAR file, and prefer not to use VisualAge for Java (in which case, you can run the generated batch file, if you are using the same process options).

A deployed JAR file is generated from an EJB JAR file. The deployed JAR file (represented by a node in the Enterprise Beans folder in the Tasks and Objects

pane of Object Builder) contains the EJB classes (represented by nodes beneath the deployed JAR file) that are required by the EJB server (CB) environment.

The ImportEJB directory, which is created in the same directory as the batch and response files, and which is at the same level as the Model and Working directories contains the XML file for your model. For example, for the imported EJBHotel.jar file, it will be EJBHotel.xml. You can later import this XML file into Object Builder.

When you exit Object Builder, after you have deployed either CMP entity beans, or session beans that are associated with an MQSeries application adaptor-backed business object, the **Do All** dialog box appears. You can specify further actions to be taken: run the Model Consistency Checker, generate all code for model, build targets, or exit Object Builder. You also have the option of changing the default checks that the Consistency Checker will perform (click the **Checker Options** button, and select a different set of options, if you want to). Click **Start** to run the Model Consistency Checker, generate code, and compile it (if you have selected the corresponding check box options).

Additional information on exporting EJB JAR files is found in the VisualAge for Java EJB Development Environment online help topic 'Exporting enterprise beans to EJB or deployed JAR files'. For points to consider when you deploy CMP beans that are created in VisualAge for Java to Component Broker, refer to the VisualAge for Java topic 'Default type mappings for deployment to Component Broker'.

RELATED CONCEPTS

The EJB JAR file (*Writing Enterprise Beans in WebSphere*)

An introduction to enterprise beans (*Writing Enterprise Beans in WebSphere*)

The deployment descriptor (*Writing Enterprise Beans in WebSphere*)

RELATED TASKS

"Chapter 8. Working with enterprise beans" on page 391

"Deploying enterprise beans"

Importing XML

Creating finder logic in the EJB server (CB) (*Writing Enterprise Beans in WebSphere*)

Deploying enterprise beans



Enterprise beans encapsulate the business logic and data used and shared by EJB clients.

The C++ code that is required to deploy an enterprise bean onto CB must be compiled on the target platform.

There are two ways to deploy enterprise beans onto Component Broker:

- “Deploying enterprise beans using Object Builder” on page 410
- “Deploying enterprise beans using the EJB Deployment Tool” on page 411 (cbejb)

Aspects of the enterprise bean persistence model that are not transferred to Object Builder

VisualAge for Java supports both composers and converters. However, composers are not supported for enterprise beans that are deployed using Object Builder and the Component Broker MOFW (CORBA) infrastructure. Some converters are supported in concept (in the form of its various attribute mapping patterns, which include the specification of arbitrary, user-defined mappings). This infrastructure is used for deploying enterprise beans only in this version of WebSphere. See the reference topic ‘Java to Object Builder type mappings’, which lists the default type conversions that take place during enterprise bean deployment.

Deploying from Windows to AIX

If you developed an EJB JAR file in VisualAge for Java on Windows, and you want to deploy it onto CB on AIX, export it using the export function in the VisualAge for Java integrated development environment (IDE). Copy the exported JAR file, and any other JAR files the beans may need to use, to your AIX machine. See the VisualAge for Java IDE help topics, ‘Exporting code’ and ‘Deploying code’ for more details.

Redeploying CMP entity beans

Certain restrictions apply when you redeploy CMP entity beans into the same Component Broker model. See ‘Restrictions for R3.5’.

RELATED CONCEPTS

Restrictions for R3.5

The EJB JAR file (*Writing Enterprise Beans in WebSphere*)

An introduction to enterprise beans (*Writing Enterprise Beans in WebSphere*)

RELATED TASKS

“Chapter 8. Working with enterprise beans” on page 391

Developing and deploying enterprise beans with EJB server (CB) tools (*Writing Enterprise Beans in WebSphere*)

RELATED REFERENCES

“Java to Object Builder type mapping” on page 394

Deploying enterprise beans using Object Builder

► CHANGED

► 390 Enterprise bean support is available on the Windows NT, AIX, Solaris, and HP-UX deployment platforms. It is also available for OS/390. However, Component Broker and WebSphere EJB clients on platforms other than CB OS/390 will not be able to exchange information with CB OS/390 enterprise beans. Neither will CB and WebSphere EJB clients that are on CB OS/390 be able to exchange information with enterprise beans on other platforms.

Enterprise beans encapsulate the business logic and data used and shared by EJB clients. Using Object Builder, you can deploy any type of enterprise bean.

Once you have imported the EJB JAR file into Object Builder, the deployed enterprise beans are created. For enterprise beans that are created using VisualAge for Java version 3.5, and for which you have defined schema and mapping information in that tool, Object Builder translates that information into data object to persistent object mappings. For enterprise beans that are created using versions of VisualAge for Java prior to 3.5, or for those beans created using any other tool, Object Builder creates default mappings between the data object and the persistent object. Of course, you can override the default mappings and provide your own.

If you select to generate and build code (by using either the pop-up menu items from the Enterprise beans folder, or from the File menu), code for the required objects is generated into the appropriate directories within your project directory.

Object Builder creates the following objects for each enterprise bean that you select for deployment:

In the User-Defined Data Objects folder:

- the data object file
- the data object interface
- the data object implementation

Note: The IDL attributes of the data object interface correspond to the entity bean’s container-managed fields.

In the User-Defined Business Objects folder:

- the associated business object
- the key

- the copy helper
- the managed object

Warning:It is recommended that you do not change the business object's implementation language, which is C++.

The following files that correspond to the imported JAR file are created in the Working\<platform> directory:

- the Tie class
- the IDL files
- the Java files

You can now build and configure your application.

Restriction:Composers are not supported for enterprise beans that are deployed using Object Builder and the Component Broker MOFW (CORBA) infrastructure even though VisualAge for Java supports both composers and converters.

For a list of the default type conversions (the supported converter mappings) that take place during enterprise bean deployment, see the reference topic 'Java to Object Builder type mappings'.

RELATED CONCEPTS

The EJB JAR file (*Writing Enterprise Beans in WebSphere*)

An introduction to enterprise beans (*Writing Enterprise Beans in WebSphere*)

RELATED TASKS

"Deploying enterprise beans" on page 408

"Importing enterprise beans into Object Builder" on page 392

"Mapping a data object to a persistent object" on page 703

"Configuring builds" on page 549

"Packaging applications" on page 574

RELATED REFERENCES

"Java to Object Builder type mapping" on page 394

Deploying enterprise beans using the EJB Deployment Tool

You can use the EJB deployment tool, which has a command-line interface, to deploy BMP entity beans, session beans, and CMP entity beans. See the task Importing enterprise beans using the EJB Deployment Tool.

RELATED CONCEPTS

The EJB JAR file (*Writing Enterprise Beans in WebSphere*)

An introduction to enterprise beans (*Writing Enterprise Beans in WebSphere*)

RELATED TASKS

“Chapter 8. Working with enterprise beans” on page 391
“Importing enterprise beans from the command line” on page 670
Developing and deploying enterprise beans with EJB server (CB) tools
(*Writing Enterprise Beans in WebSphere*)

RELATED REFERENCES

“cbejb options” on page 673

Deploying enterprise beans into a polymorphic home

► NEW

The process of deploying an enterprise bean into a polymorphic home is different for beans that have been mapped (by means of the mapping between the associated data object and persistent object), and for those that have not been mapped.

Note: You can use polymorphic homes for deployment only if the inheritance follows the single table pattern: you cannot use polymorphic homes and have inheritance with either the attributes duplication pattern, or the key duplication pattern (that is, any root-leaf patterns).

Polymorphic deployment for unmapped enterprise beans

To deploy an unmapped enterprise bean, follow these steps:

1. Import the enterprise beans into Object Builder using either the interface (see “Importing enterprise beans using Object Builder” on page 401), or the command line (see “Importing enterprise beans from the command line” on page 670)
2. On the Deployment Information page, specify the type of home for the enterprise beans to be polymorphic. This instructs the tool to create a polymorphic specialized home instead of a queryable, iterable, or a regular home.

Object Builder automatically creates a single table mapping (inheritance with a single datastore pattern) from a hierarchy of enterprise beans. It follows these steps:

1. Defines a table that contains columns for the container-managed (CM) fields of the enterprise beans in the hierarchy.
2. Defines a column to hold a discriminator value.
3. Defines a unique discriminator value for each pair of objects that consists of an enterprise bean and its data object implementation. This unique value is the EJB class name.
4. Defines a discriminator expression on each data object implementation. This expression contains both the discriminator column and the unique value it is set to.

Note: The homes of enterprise beans reside in the Object Builder model as specialized homes. Each enterprise bean that is deployed into Object Builder is supported by two complete managed object assemblies:

- a regular managed object assembly for the EJBObject, and
- a specialized home managed object assembly for the enterprise bean's EJBHome.

Polymorphic deployment for mapped enterprise beans

Given a hierarchy of enterprise beans that are mapped (in the Managed Object Framework (MOF)) to a single table, an equivalent mapping is automatically created in Object Builder.

In VisualAge Persistence (VAP, which is also called the Persistence Builder), you can define a discriminator column (which is unmapped to any CM field of the enterprise beans) for the hierarchy, and a discriminator value for each enterprise bean in the hierarchy.

Object Builder follows these steps:

1. Creates an unmapped column in the single table
2. Creates a discriminator expression on each data object implementation that contains the equal to (=) sign, and equates the discriminator column to the discriminator value for the data object implementation's enterprise bean.

Note: If you provide your own data object to persistent object mappings, and it does not satisfy all the criteria for a polymorphic model, the Consistency Checker will warn you.

RELATED REFERENCES

"cbejb options" on page 673

Working with deployed EJB JAR files

You can import an EJB JAR file that is created using either VisualAge for Java, or any other tool such as the jetace tool, into Object Builder.

The following tasks deal with deployed EJB JAR files:

- "Creating a deployed EJB JAR file" on page 414
- "Editing an EJB JAR file" on page 415
- "Deleting an EJB JAR file" on page 415

RELATED CONCEPTS

The EJB JAR file (*Writing Enterprise Beans in WebSphere*)

An introduction to enterprise beans (*Writing Enterprise Beans in WebSphere*)

RELATED TASKS

“Working with components” on page 697

Creating a deployed EJB JAR file

► CHANGED

An EJB JAR file contains one or more enterprise beans. When you import an EJB JAR file into Object Builder, it is converted to a deployed JAR file.

To create a deployed EJB JAR file in the Enterprise Beans folder, follow these steps:

1. Select the Enterprise Beans folder in the Tasks and Objects pane.
2. From its pop-up menu, select **Import EJB JAR**.
3. The Import EJB JAR wizard opens to the Enterprise Bean Selection page. Specify the JAR file that you want to import, and the project directory for the imported enterprise beans. You can indicate whether you want code to be generated, and then built for the objects that are created at the time of the import. Select which of the imported enterprise beans in the JAR file have to be deployed. You can also indicate whether you want to preserve the mapping between the data object and the persistent object if it has already been defined.
4. Click **Next**. The EJB Browser page opens.
5. You can use this page to view, or edit the settings of the beans to be deployed: Select the bean from the **Enterprise Beans to Be Deployed** list, and click the **Properties** button.
6. The Deployment Information page opens. Except for the deployment platforms, you can edit all other information on this page. You can provide a finder helper class name, a database name, and add or delete client and server JAR file dependencies.

Note: If you created the beans that you imported using VisualAge for Java, do not use the finder helper class that is generated by the tool. Instead, you must use the **FinderHelperGenerator** utility of the EJB server in CB to implement the finder helper class. For example, to generate a finder helper class for the AccountHome interface, use the command:

```
# ejbfhgen com.ibm.ejs.doc.account.AccountHome
```

This command generates the finder helper class named com.ibm.ejs.doc.account.AccountHome.

For more information on finder helper classes, see Defining finder methods, and in particular, Creating finder logic in the EJB server (CB) in *Writing Enterprise Beans in WebSphere*.

7. Click **Finish**.

RELATED CONCEPTS

The EJB JAR file (*Writing Enterprise Beans in WebSphere*)

An introduction to enterprise beans (*Writing Enterprise Beans in WebSphere*)

RELATED TASKS

“Working with deployed EJB JAR files” on page 413

RELATED REFERENCES

“Naming objects” on page 128

“Internationalization of data” on page 132

Editing an EJB JAR file

You cannot edit the properties of an EJB JAR file; you can only view its details. Follow these steps:

1. Select the EJB JAR file from the Enterprise Beans folder.
2. From the pop-up menu of the file, select **Properties**. The Import EJB JAR wizard opens to the EJB JAR File page.
3. This page shows you the project directory that is used for the deployment of the enterprise beans, and the deployed enterprise beans that belong to the EJB JAR file.

RELATED CONCEPTS

The EJB JAR file (*Writing Enterprise Beans in WebSphere*)

An introduction to enterprise beans (*Writing Enterprise Beans in WebSphere*)

RELATED TASKS

“Working with deployed EJB JAR files” on page 413

Deleting an EJB JAR file

To delete an EJB JAR file, follow these steps:

1. Select the EJB JAR file in the Enterprise Beans folder.
2. From the pop-up menu of the file, select **Delete**.

The EJB JAR file, and all the classes that belong to it are deleted from the folder.

RELATED CONCEPTS

The EJB JAR file (*Writing Enterprise Beans in WebSphere*)

An introduction to enterprise beans (*Writing Enterprise Beans in WebSphere*)

RELATED TASKS

“Working with deployed EJB JAR files” on page 413

Working with deployed enterprise beans

You can import an EJB JAR file that is created using either VisualAge for Java, or any other tool such as the jetace tool, into Object Builder. Deployed enterprise beans are generated in the process.

The following tasks deal with deployed enterprise beans:

- “Creating a deployed enterprise bean”
- “Editing an EJB class”
- “Deleting an EJB class” on page 417

RELATED CONCEPTS

The EJB JAR file (*Writing Enterprise Beans in WebSphere*)

An introduction to enterprise beans (*Writing Enterprise Beans in WebSphere*)

RELATED TASKS

“Working with components” on page 697

Creating a deployed enterprise bean

Enterprise beans are contained in EJB JAR files. When you import an EJB JAR file into Object Builder, it is converted to a deployed JAR file, with the enterprise beans being converted into deployed enterprise beans.

To create a deployed enterprise bean, follow the steps in the task “Creating a deployed EJB JAR file” on page 414.

When you click Finish from the Import EJB JAR wizard, the deployed enterprise beans are created in the Enterprise Beans folder, and are represented as EJB classes beneath the EJB JAR file.

RELATED CONCEPTS

The EJB JAR file (*Writing Enterprise Beans in WebSphere*)

An introduction to enterprise beans (*Writing Enterprise Beans in WebSphere*)

RELATED TASKS

“Working with deployed enterprise beans”

RELATED REFERENCES

“Naming objects” on page 128

“Internationalization of data” on page 132

Editing an EJB class



The EJB class represents a deployed enterprise bean in Object Builder, and is represented as a node beneath the deployed EJB JAR file in the Enterprise Beans folder.

You cannot edit the properties of an EJB class, but you can view them. Follow these steps:

1. Select the EJB class in the Enterprise Beans folder.
2. From the pop-up menu of the file, select **Properties**. The EJB Browser wizard opens to the Deployment Descriptor page. This page is read-only. View the deployment descriptor settings for the enterprise bean.
3. Click **Next**. The Deployment Information page opens.
4. View the information, and click **Finish**.

You can edit the EJB class with the tool that you used to create the enterprise bean that you imported into Object Builder. You can then reimport the bean into Object Builder, and redeploy it.

RELATED CONCEPTS

The EJB JAR file (*Writing Enterprise Beans in WebSphere*)

An introduction to enterprise beans (*Writing Enterprise Beans in WebSphere*)

RELATED TASKS

“Working with deployed enterprise beans” on page 416

Deleting an EJB class

To delete an EJB class, follow these steps:

1. Select the EJB class in the Enterprise Beans folder.
2. From the pop-up menu of the class, select **Delete**.

The EJB class is deleted from the folder.

RELATED CONCEPTS

The EJB JAR file (*Writing Enterprise Beans in WebSphere*)

An introduction to enterprise beans (*Writing Enterprise Beans in WebSphere*)

RELATED TASKS

“Working with deployed enterprise beans” on page 416

Chapter 9. Multi-platform development

Multi-platform development

► CHANGED

You can use Object Builder to develop components for deployment on Windows NT, AIX, OS/390, Solaris, or HP-UX servers, with some varying functionality from platform to platform.

Most development options are the same for all platforms: the main differences appear when you generate the code for your components. There are three mechanisms in place for dealing with these differences: platform-filtered views, platform-targeted code generation, and platform-specific development constraints. You can also implement different versions of your method implementations (and correspondingly, different versions of file and method adornments) for different platforms.

Views

You can select a platform view from the **Platform > View** menu in Object Builder. Inheritance options, framework methods, and framework method implementations will be filtered for the selected platform. The information for all views is stored in the same project model; you can switch between views at any time.

Code generation

You can select platforms to generate for from the **Platform > Generate** menu in Object Builder. For each platform you select, an equivalent subdirectory will be added to the project's \Working directory. For example, if you select AIX and 390, you will have code generated to the directories `<project>\Working\AIX` and `<project>\Working\390`. Every time you select a **Generate** option from within the Tasks and Objects pane, code will be generated for all selected platforms. The more platforms you select to generate for, the longer code generation will take.

Constraints

You can set constraints to ensure that the components you develop will be deployable on your target platforms. Select platform constraints on the **Platform > Constrain** menu in Object Builder. By default, any components you develop will be deployable on the platforms you selected. You can override these defaults on a particular object, reducing the number of platforms to which it can be deployed, to create a platform-specific version of the object. The more platforms an artifact is deployable to, the more restrictions there may be in its definition.

You can set object-specific platform constraints on business object interfaces, business object implementations, data object interfaces, data object implementations, local-only objects, managed objects, containers, application families, and DLLs. For containers, DLLs, and application families, you can set the constraints when you either create the object, or edit its properties (**Properties** from the pop-up menu of the object). For the other objects, you can set the object-specific platform constraints when you either create the object, or edit its file properties (**File Properties** from the pop-up menu of the object).

On the first page of the object's wizard (if you are creating the object, or even if you are editing containers and DLLs), or the object's File wizard (if you are editing the object's file), in the section **Deployment Platforms**, you can select a subset of the platform constraints to apply.

For example, if your platform constraints are set to AIX and 390, you can specify one of the following sets of deployment platforms:

- select only 390 to apply only 390 constraints (to develop a 390-specific version of the object)
- select only AIX to apply only AIX constraints (to develop an AIX-specific version of the object)
- select both 390 and AIX (to deploy the object on both platforms)

After you have set platform constraints, you can apply the constraints to your primary model (except to method bodies and adornments) by selecting **Platform > Constrain > Apply Constraints to Model**. All existing files, DLLs and containers in the model will be updated to reflect any changes. After the update, the Consistency Checker is automatically run on the model. This option is particularly useful if you want to apply a new platform constraint to your earlier models.

Methods

In the Properties wizard for a method, you can define whether its implementation in the Source pane is to be used for all platforms, or to be defined separately for each platform. Access the wizard from the Methods pane, by selecting **Properties** from a method's pop-up menu. Once you have set to use different versions, you can use the **Platform > View** menu option to choose which platform-specific implementation to display and edit in the Source pane.

Properties for file and method adornments also allow for platform tagging.

RELATED CONCEPTS

"Cross-platform development" on page 426

"File and method adornments" on page 239

RELATED TASKS

“Setting platform constraints”

“Generating code” on page 551

“Tutorial: Developing a multi-platform application” on page 429

“Adding file adornments” on page 240

“Adding method adornments” on page 242

RELATED REFERENCES

“Platform differences” on page 425

Setting platform constraints



In Object Builder, you can develop components that will work on Windows NT, AIX, OS/390, Solaris, and HP-UX. However, not all development options are available for every platform. To ensure that your components will run on the platforms you intend to deploy on, and to take advantage of all opportunities available for each platform, you can constrain your development options on both a project level and on an object level.

To ensure that objects you create will run on your deployment platforms, you can set project-wide constraints that allow access only to development options available on all your deployment platforms. To set project-wide platform constraints, follow these steps:

1. From the Object Builder menu bar, click **Platform > Constrain**.
2. From the cascade, select a platform constraint.
3. Add additional platform constraints in the same way.

Once you set these constraints, development options (such as framework inheritance and services) are filtered to ensure that the application you develop will be deployable to the platforms you select. This is a “least common denominator” approach; you may want to supplement it by developing some objects in multiple versions, to take advantage of some platform-specific options.

Within the project-wide constraints, you can develop multiple versions of an object for your different deployment platforms. For example, in a project to be deployed on AIX and OS/390, you could develop a data object implementation for AIX only, and another data object implementation for OS/390 only. This would allow you to use the Cache Service on AIX, which is unavailable for OS/390 (where the Embedded SQL pattern performs well enough to require no alternative).

You can set object-specific platform constraints on data object implementations, managed objects, application families, business object interfaces,

business object implementations, containers, and DLLs. You can set the constraints when you create the object, or when you edit its file properties.

To set object-specific constraints when you create the object, follow these steps:

1. Open the object's wizard.

On the first page of the wizard, the group box **Deployment Platforms** contains check boxes for NT, AIX, OS/390, Solaris, and HP-UX. Only platforms that are listed in your project-wide constraints are available for selection.

By default, all the platforms in the project-wide constraints are listed.

2. Clear the platform check boxes for any platforms that you are not deploying this object on. For example, if you are deploying the object for AIX only, make sure the NT and OS/390 check boxes are *not* checked.

The wizard will now allow access to all options available for the platforms you indicated. For example, if you are deploying the object for AIX only, all AIX-specific options will be available, even those not available on other platforms.

3. Complete your selections in the wizard, and click **Finish**.

To set object-specific platform constraints when you edit the object, follow these steps:

1. From the pop-up menu of the object, select **File Properties**. The object's wizard opens to the Name page.
2. In the "Deployment platforms" on page 423 section, select or clear the deployment platform check boxes of your choice.
3. Click **Finish**.

The platform constraint settings for the object take effect.

To change your project-wide platform constraints, follow these steps:

1. From the Object Builder menu bar, click **Platform > Constrain**.
2. From the cascade, select or clear a platform constraint.
3. Add or remove additional platform constraints in the same way.

The new constraints will affect the choices you have in developing new objects, but will not affect any existing objects created under different constraints. To check your application under the new constraints, run a consistency check on the project's model.

4. From the Object Builder menu bar, click **File > Check Model**.
5. Review the report, and save it if you want before closing it.
6. Edit objects as necessary to make your model consistent under the new constraints.

If you want to apply the current set of constraints to *all* artifacts in your primary model, select **Platform > Constrain > Apply Constraints to Model**. This will apply the constraints to all artifacts (except method bodies and adornments), and then run the consistency checker. You may find this useful when you are retrofitting a new platform (such as HP-UX) into earlier models; it will save you from opening the wizard for each artifact in the model.

RELATED CONCEPTS

“Multi-platform development” on page 419

RELATED TASKS

“Checking a model for consistency” on page 31

“Tutorial: Developing a multi-platform application” on page 429

RELATED REFERENCES

“Platform differences” on page 425

Deployment platforms

You can select a particular deployment platform or platforms for an object within a multi-platform application. By making the object specific to a particular platform or platforms, you can take advantage of development options available only on those platforms.










The deployment platforms available for selection are based on the choices made in the **Platform > Constrain** menu. For example, if you have constrained your entire project to only NT and AIX platforms, then you cannot select to deploy an object on OS/390, but you can still decide to make an object deployable only on NT.

The full range of deployment platforms are:




- NT
- AIX
- 390
- Solaris
- HP-UX


Note the following points:

- For version 3.5 (R3.5), the following platforms are supported in the CB run-time level of function:
 - CB/NT 3.5
 - CB/AIX 3.5
 - CB/Solaris 3.5

- CB/390 3.02 (A minimal subset, from a tooling standpoint, of the 3.5 net function on the workstation.)
- CB/HP-UX 3.0 (HP-UX support is at version 3.0 beta level.)
-    All R3.5 Object Builder features that exploit new run-time function in 3.5 will be enabled for models and artifacts that are deployed to the Windows NT, AIX and Solaris platforms.
-      All R3.5 Object Builder features, which are pure Toolkit features will be enabled for models and artifacts that are deployed to all platforms.
-  Several run-time functional enhancements were made in R3.0. Since the Solaris run time is stepping directly from an R2.0 level of function to an R3.5 level of function, Object Builder retroactively enables exploitation of these R3.0 run-time features on Solaris.

By default, all the platforms specified on the **Platform > Constrain** menu are selected. You can only select from these deployment platforms: a platform that has *not* been selected on the **Constrain** menu cannot be selected as a deployment platform.

   You can select only the Windows NT, Solaris and HP-UX platforms for deployment if you want to use the MQSeries application adaptor support (that is, if you want to create applications that send messages to, or receive messages from queues that are managed by MQSeries application adaptors).

 Enterprise bean support is available on the Windows NT, AIX, Solaris, and HP-UX deployment platforms. It is also available for OS/390. However, Component Broker and WebSphere EJB clients on platforms other than CB OS/390 will not be able to exchange information with CB OS/390 enterprise beans. Neither will CB and WebSphere EJB clients that are on CB OS/390 be able to exchange information with enterprise beans on other platforms.

RELATED CONCEPTS

“Multi-platform development” on page 419

RELATED TASKS

“Setting platform constraints” on page 421

RELATED REFERENCES

“Platform differences” on page 425

Platform differences

►CHANGED

Most development options are the same for all platforms. The main differences are between OS/390 and the workstation platforms, NT, AIX, Solaris, and HP-UX.

The following differences apply between OS/390 and the workstation platforms:

- **Inheritance**

Different framework inheritance may apply for the different platforms. When you generate code for multiple platforms, the right inheritance will automatically be used. When you view a specific platform, the inheritance that applies to that platform is shown. Not all inheritance options have cross-platform equivalents.

If you are developing an OS/390 component, you cannot select the parent:

IBOIMExtLocal IBOIMExtLocal::IUUIDCopyHelperBase

- **Framework methods**

Different framework methods may apply for the different platforms. When you generate code for multiple platforms, the right framework methods will automatically be implemented. When you view a specific platform, the framework methods that apply to that platform, and the appropriate method implementations, are shown.

- **Wide and long long types**

Wide and long long types are not available on OS/390. Do not use when you develop for OS/390. They are not available for selection if you have 390 listed as a platform constraint.

- **Services**

Cache Service is not available on OS/390.

- **Sessionable managed objects**

You cannot create sessionable managed objects for OS/390.

- **PA development**

Procedural Adaptor persistent objects on OS/390 and the workstation platforms have mutually exclusive connection types. You cannot create common PA persistent objects for both OS/390 and any other platform. You must create platform-specific versions of the data object implementations and persistent objects for PA components.

- **Container definition**

OS/390 servers do not use any of the container information you provide when you define a container, except for its name and description. If you are developing an OS/390-specific container, the additional pages are not available. If you are developing a container for multiple platforms including OS/390, the additional pages are available, but the information on them will be ignored by the OS/390 server.

RELATED CONCEPTS

“Multi-platform development” on page 419

RELATED TASKS

“Setting platform constraints” on page 421

“Generating code” on page 551

“Tutorial: Developing a multi-platform application” on page 429

Cross-platform development

► CHANGED

You can use Object Builder on one platform to generate code for another platform. For example, you can develop an application on Windows NT, and build the application on AIX and OS/390.

Transfer of files among platforms

When you transfer files between Windows NT and other platforms, or between OS/390 and other platforms, use an ASCII-aware mechanism (such as FTP in ASCII mode, or an ASCII NFS mount). Some programs, such as Systems Management, fail unless the file is a valid, native ASCII file. If you transfer code in binary form (for example, with binary FTP, a binary NFS mount, or inside a .tar archive), the following problems occur:

- **► WIN** Windows NT CR/LF characters will not be mapped to equivalent UNIX new line characters, and files that depend on line formatting (for example, .mak, .sql, .ddl) will not be usable.
- **► 390** The ASCII character codings on NT and Unix will not be mapped to equivalent EBCDIC character codings on OS/390. All files will be transferred in EBCDIC, and will be unusable.
- **► 390** **► HP-UX** **► SOLARIS** If a business object depends on other business objects, the paths to the directories that contain those models is written by Object Builder into the file prjdefs.mk. If you copy the generated source to OS/390, HP-UX, or Solaris, you must edit prjdefs.mk, and modify the path to be compliant with the Solaris machines. (It is an NT path that is generated into prjdefs.mk.)

In fact, the paths can differ whenever generated code is moved, including from one directory to another within a single AIX or NT system. Use the Build Location page in the Build Configuration wizard to override absolute path names for each target platform. These pathnames will be emitted instead of `.../project/Working/<platform>...` in prjdefs.mk, qt.bat, QTjar.txt and the DDL file.

It is recommended that you use the ASCII transfer option of the basic ftp client to transfer files between different platforms.

Transfer of single files among platforms

To transfer a single file in text mode without the addition of Ctrl-M's, use the `-ascii` option when you use ftp.

To transfer a zip file, use binary mode with ftp, and use the following command to unzip it:

```
unzip -a <zip filename>
```

► **AIX** To remove Ctrl-M's from files after you transfer them from another platform to AIX, follow these steps:

1. Create a file called `mydos2unix`, which has the following commands:

```
#!/bin/ksh
cat $1 | sed s/^M//g 2>&1 | tee $1 # Use CTRL-v-m to get the ^M
```

Note: You must use CTRL-v-m to replace the character '^M' with blanks.

2. Save and Exit
3. Move this file somewhere in your PATH environment.
4. Change its mode to executable:

```
chmod +x mydos2unix
```

5. Run it using the following command syntax, within the directory where the files are you want to change.

```
ls -l | grep -v ^drw | awk '{print $9}' | xargs -i mydos2unix {}
```

Note: `grep -v ^drw` excludes directories.

This will copy <DOS file> (which is the file that contains the Ctrl-M characters) back to itself to become the <UNIX file>, stripping it of the Ctrl-M's.

► **SOLARIS** To remove Ctrl-M's from files after you transfer them from another platform to Solaris, use the command:

```
dos2unix -ascii
```

► **HP-UX** To remove Ctrl-M's from files after you transfer them from another platform to HP-UX, use the following commands:

```
dos2ux file > tmpfile
mv tmpfile file
```

Note the following points:

- You must use a temporary file for the transfer: if you use the command `dos2ux file > file`, you will lose the contents of the file.

- The transferred file must be named following HP-UX standards.

Transfer of directories among platforms

Some ftp programs do not transfer directories correctly. For example, the NT ftp program will transfer directories as zero-byte files (even though the ftp client acknowledges that it cannot transfer directories). If the output directories NOOPT, PRODUCTION, TRACE and TRACE_DEBUG are present on the target system as zero-byte files, the makefiles will fail. So, if you have content in the output directories, or if you have QuickTest artifacts that are generated, you must either use an ftp program such as Hummingbird[®] Exceed ftp that is capable of transferring nested directories, or NFS (the standard file transfer facility on UNIX systems) for the transfer.

To easily transfer a group of files and subdirectories, follow these steps:

1. Archive them into a single file using either the tar or zip utility
2. Transfer the archive file
3. Extract the archive on the target machine.

For example, if you want to transfer the directory **Solaris**, follow these steps:

1. cd to its parent directory
2. Archive the files by running the command: `tar cvf Solaris.tar Solaris`
3. Transfer the files using the command: `ftp <target_machine>` (and transfer Solaris.tar)
4. Login to the target machine, and cd to the parent directory.
5. Extract the archive using the command: `tar xvf Solaris.tar`

You now have a Solaris directory that contains all the files and subdirectories from the original directory.

RELATED CONCEPTS

“Multi-platform development” on page 419

“Remote build” on page 570

RELATED TASKS

“Launching a remote OS/390 build” on page 572

RELATED REFERENCES

“Platform-specific information” on page 20



Tutorial: Developing a multi-platform application

Objectives

- To create a component for deployment on two different platforms.
- To add platform-specific method implementations.
- To create platform-specific versions of a data object implementation.
- To build DLLs for the different platforms.
- To define containers for use on each platform.
- To create application packages for each platform.

Before you begin

You need the following installed on your system:

- A CB Server
- A CB Client
- CB tools, including Samples
- DB2 Universal Database
- DB2 SDK
-  VisualAge for C++ and (for Java applications) VisualAge for Java
-  IBM C and C++ Compilers for AIX V3.6.4 and (for Java applications) VisualAge for Java

For a complete list of prerequisites that you must install on your system, see the *Getting Started with Component Broker* documentation for the appropriate platform.

You should be familiar with the Component Broker programming model, as described in the *IBM Component Broker Programming Guide*.

You should be familiar with the steps involved in defining, building, and packaging components in Object Builder. For most tasks in this tutorial, you will be given only general directions. For more specific instructions, you can refer to the referenced scenario or one of its sequels.

Description

This exercise defines the objects required to create a component named “Claim” for deployment on the AIX and OS/390 platforms. The component will have platform-specific versions of its data object implementation. For this exercise, you will:

1. Create the project
2. Create a business object interface
3. Add a key and copy helper
4. Add a business object implementation

5. Add a data object implementation for AIX
6. Define a data object implementation for OS/390
7. Define a persistent object and schema
8. Add a managed object
9. Generate the code
10. Define a client DLL and server DLL for AIX
11. Define a client DLL and server DLL for OS/390
12. Define an application family and application for AIX
13. Define an application family and application for OS/390
14. Configure the component with both applications

Creating the project

Create a sample project to hold your work.

1. Start Object Builder.
2. In the Open Project wizard, type a name and path for the project directory.
3. Click **Finish**.
4. When asked whether you want to create a new project, click **Yes**.

Ensure that platform constraints are set to include AIX and OS/390. Click **Platform > Constrain** and ensure that **AIX** and **390** are selected. By default, any objects that have platform-specific development options will only allow selection of options that exist on both AIX and OS/390. These constraints will be overridden when you create the data object implementation, to allow separate versions for each platform.

Set code generation for AIX and OS/390:

1. Click **Platform > Generate > AIX**
2. Click **Platform > Generate > 390**

Code will be generated for both platforms, into the `\Working\AIX` and `\Working\390` directories.

3. Click **Platform > View > AIX**

When there are differences in an object's inheritance or framework methods for different platforms, you will see the AIX version.

Creating the business object interface

Define a business object file (MPFile):

1. From the User-Defined Business Objects folder's pop-up menu, click **Add File** to open the Business Object File wizard.
2. Name the file.
3. Click **Finish**. The file now appears under the folder.

Add a module (MPModule):

1. From the pop-up menu of the file, click **Add Module** to open the Business Object Module wizard.
2. Name the module.
3. Click **Finish**. The module now appears under the file.

Add an interface (Agent):

1. From the pop-up menu of the module, click **Add Interface** to open the Business Object Interface wizard.
2. Name the interface csAgent.
3. Click the page title and turn to the Attributes page.
4. Add the following attributes:
 - readonly float commissions
 - float commPercent
 - float pendingPaycheck
 - string agentName
 - readonly long id
5. Set the size of agentName to 100. You should always provide a size for string attributes.
6. Click **Next** and turn to the Methods page.
7. Add the following method:
 - void payCommission (in float amount)
8. Click **Finish**.

Adding a key and copy helper

Add a key (AgentKey):

1. From the interface's pop-up menu, click **Add Key** to open the Key wizard.
2. Accept the default name; select the id attribute and add it to the **Key Attributes** list.
3. Click **Finish**.

Even though the id attribute of the business object interface is read-only, the id attribute of the key is both readable and writable. The client application can set the value of the id on the key, and use it either to initialize a new instance of the component, or to locate an existing instance of the component, on the server.

Add a copy helper (AgentCopy):

1. From the interface's pop-up menu, click **Add Copy Helper** to open the Copy Helper wizard.
2. Accept the default name; select all listed attributes and add them to the **Copy Helper Attributes** list.
3. Click **Finish**.

Adding a business object implementation and data object interface

Add a business object implementation (AgentBO) and data object interface (AgentDO):

1. From the interface's pop-up menu, click **Add Implementation** to open the Business Object Implementation wizard.
2. Change the **Pattern for Handling State Data** to **Delegating**. This is easier to debug than the **Caching** pattern.
3. Click the page title and turn to the Implementation Language page.
4. Select the language you want the business object to be implemented in (**C++** or **Java**).
5. Click the page title and turn to the Key and Copy Helper page. The appropriate key and copy helper are already selected.
6. Click the page title and turn to the Data Object Interface page.
7. Select all attributes and add them to the **State Data** list (to be preserved in the data object).
8. Click **Finish**.

Adding platform-specific method implementations

For each method, you can specify whether to use a different method implementation for each platform, or share the same implementation on all platforms. By default, method implementations are shared.

Make the payCommission() method implementation platform-specific:

1. Click on AgentBO in the Tasks and Objects pane. Its methods and attributes are listed in the Methods pane.
2. In the Methods pane, locate the payCommission() method.
3. From the pop-up method for the approve() method, click **Properties** to open the Method Implementation wizard.
4. Deselect the option **Method body is the same for all platforms**.
5. Click **Finish**.

Add an implementation for the payCommission() method on AIX:

1. Click on the payCommission() method in the Methods pane. The skeleton implementation appears in the Source pane.
2. Type the following implementation for the payCommission() method:

C++

```
float tmp;  
tmp = amount * iDataObject->commPercent();  
iDataObject->commissions(tmp);  
iDataObject->pendingPaycheck(iDataObject->pendingPaycheck() + tmp);  
// Applies to AIX
```

Java

```

float tmp;
tmp = amount * iDataObject.commPercent();
iDataObject.commissions(tmp);
iDataObject.pendingPaycheck(iDataObject.pendingPaycheck() + tmp);
// Applies to AIX

```

Add an implementation for the payCommission method on OS/390:

1. Click **Platform > View > 390**.
2. Click on the payCommission() method in the Methods pane. The method implementation you provided for AIX does not appear. Instead you see a skeleton implementation for the 390-specific version of the implementation.
3. Type the following implementation for the payCommission() method:

C++

```

float tmp;
tmp = amount * iDataObject->commPercent();
iDataObject->commissions(tmp);
iDataObject->pendingPaycheck(iDataObject->pendingPaycheck() + tmp);
// Applies to 390

```

Java

```

float tmp;
tmp = amount * iDataObject.commPercent();
iDataObject.commissions(tmp);
iDataObject.pendingPaycheck(iDataObject.pendingPaycheck() + tmp);
// Applies to 390

```

When you generate code for the business object implementation, the code in the Working\AIX directory will use the AIX-specific implementation, and the code in the Working\390 directory will use the OS/390-specific implementation.

In most cases, you should be able to use the same implementation for all platforms. This example is intended to show the procedure, but is not intended as a model for you to follow.

The method implementations for the attribute get and set methods apply to all platforms you generate code for, because you did not change the default settings in the methods' Method Implementation wizards.

Adding a data object implementation for AIX

1. From AgentDO's pop-up menu, click **Add Implementation** to open the Data Object Implementation wizard.

On the first page, the **Deployment Platforms** constraints are listed:

- NT is disabled, and cannot be selected, because the project-wide platform constraints exclude it. This prevents you from creating an NT-specific object within project constraints for AIX and 390.

- **AIX** is selected by default, based on the project-wide platform constraints.
 - **390** is selected by default. based on the project-wide platform constraints.
2. Deselect the **390** option. AIX-specific development options are now available.
 3. Name the object AgentAIXDOImpl, with the file name AgentFileAIXDOImpl.
 4. Click **Next** to turn to the Behavior page.
 5. Set the following behaviors:
 - **Environment: BOIM with any key**
 - **Type of Persistence: Cache Service**
 The Cache Service is not used on OS/390. By making this object AIX-specific, it can take advantage of the Cache Service, while the 390-specific version can use the delegating pattern.
 For more information on the Cache Service, see the *IBM Component Broker Advanced Programming Guide*.
 - **Data Access Pattern: Delegating**
 6. Click **Finish**. AgentAIXDOImpl appears under ClaimDO.

Adding a data object implementation for OS/390

1. From AgentDO's pop-up menu, click **Add Implementation** to open the Data Object Implementation wizard.
 On the first page, the **Deployment Platforms** constraints are listed:
 - **NT** is greyed out and cannot be selected, because the project-wide platform constraints exclude it. This prevents you from creating an NT-specific object within project constraints for AIX and 390.
 - **AIX** is selected by default, based on the project-wide platform constraints.
 - **390** is selected by default. based on the project-wide platform constraints.
2. Deselect the **AIX** option.
3. Name the object Agent390DOImpl, with the file name AgentFile390DOImpl.
4. Click **Next** to turn to the Behavior page.
5. Set the following behaviors:
 - **Environment: BOIM with any key**
 - **Type of Persistence: Embedded SQL**
 You cannot select the Cache Service option here, because it is not available on OS/390. The embedded SQL option on OS/390 is fast enough not to require an alternative.

- **Data Access Pattern: Delegating**

6. Click **Finish**. AgentAIXDOImpl appears under ClaimDO.

Defining a persistent object and schema

Each version of the data object requires its own persistent object (one with the **Cache Service** type of persistence and one with the **Embedded SQL** type of persistence), but they can share the same schema definition because they are both accessing the same data.

Add a persistent object for AIX, and a common schema for both platforms:

1. From the pop-up menu of AgentAIXDOImpl, click **Add Persistent Object and Schema** to open the Add Persistent Object and Schema wizard.
2. Type AgentDBGGroup in the **Group Name** field.
3. Type CBSampDB in the **Database** field.
4. Name the persistent object AgentAIXPO.
5. Click the **Finish** button.

The AgentDBGGroup schema group, CBSampDB.Agent schema, and AgentAIXPO persistent object appear in the DBA-Defined Schemas folder.

Add a persistent object for OS/390:

1. From the pop-up menu of CBSampDB.Agent in the DBA-Defined Schemas folder, click **Add Persistent Object** to open the Add Persistent Object wizard.
2. Make sure the type of persistence is set to **Embedded SQL**.
3. Review the mappings (from **SQL Type** INTEGER to **Attribute Type** long, and so on).
4. Name the persistent object Agent390PO.
5. Click **Finish**.

Agent390PO appears under CBSampDB.Agent in the DBA-Defined Schemas folder.

Map the OS/390 data object implementation and persistent object:

1. From the pop-up menu of Agent390DOImpl in the User-Defined Data Objects folder, click **Properties** to open the Data Object Implementation wizard.
2. Click the page title and turn to the Associated Persistent Objects page.
3. Add a persistent object instance with the default instance name (iPO), and with type Agent390PO.
4. Click **Next** to turn to the Attributes Mapping page.
5. Map the attributes (commissions to iPO.commissions, commPercent to iPO.commPercent, and so on).

6. Click **Next** to turn to the Methods Mapping page.
7. Map each method to its equivalent (insert to `iPO.insert`, retrieve to `iPO.retrieve`, and so on).
8. Click **Finish**.

Agent390PO appears under Agent390DOImpl, in the User-Defined Data Objects folder and User-Defined Business Objects folder.

Adding a managed object

While you can create separate managed objects for both platforms, there is no need in this case. Both versions of the component can use the same managed object.

1. From the business object implementation's pop-up menu, click **Add Managed Object** to open the Managed Object wizard to the Name and Service page.
2. Accept all the defaults and click **Finish**.
3. Click **File > Save** to save your work before continuing to the next step.

AgentMO appears under AgentBO, in the User-Defined Business Objects folder.

Generating the code

You can generate all the code for the components, including the separate method versions and data object implementations for each platform, in one step:

1. From the pop-up menu of AgentFile in the User-Defined Business Objects folder, click **Generate > All**.

This will take some time. When the code generation is complete, review the contents of the two directories (`Working\AIX` and `Working\390`).

Configuring the database

You need to define (in DB2) the CBSAMPDB database and `csAgent` table that your component will access. You should have a database administrator perform this procedure.

To configure the database and table, you need to enter the following commands from a DB2 command prompt.

```
create database CBSAMPDB
connect to CBSAMPDB
db2 -t -f csAgent.sql
```

Defining a common client DLL

Because the client interfaces are the same on both platforms, there is no need to specify separate client DLLs (known on AIX as shared library files). You

can define a single client DLL, using the same build configuration options. The appropriate makefile will be generated into both the Working\AIX\PRODUCTION and Working\390\PRODUCTION directories.

1. From the pop-up menu of the Build Configuration folder, click **Add Client DLL** to open the Client DLL wizard.
2. Name the DLL AgentC.
3. Set the deployment platforms to **AIX** and **390**.
4. Click the page title and turn to the Client Source Files page.
5. Select all the client source files for Agent and add them to the **Items Chosen** list.
6. Click **Finish**.

AgentC appears under the Build Configuration folder.

Defining a server DLL for AIX

Because you have different data object implementations for the two platforms, you need to define different server DLLs.

1. From the pop-up menu of the Build Configuration folder, click **Add Server DLL** to open the Server DLL wizard.
2. Name the DLL AgentAIXS.
3. Set the deployment platform to **AIX**.
4. Click **Next** to turn to the Libraries to Link With page.
5. Select ClaimC.
6. Click **Next** to turn to the Server Source Files page.
7. Select all the server source files for Claim except for Claim390DOImpl, and add them to the **Items Chosen** list.
8. Click **Finish**.

Defining a server DLL for OS/390

In addition to defining a separate server DLL for OS/390, you can also choose to run a remote build.

1. From the pop-up menu of the Build Configuration folder, click **Remote OS/390 Options** to open the Remote OS/390 Options wizard.
You can specify an OS/390 host on which to build the Claim DLLs for OS/390, using the generated source in the Working\390 subdirectory. When you build the DLLs, the OS/390 DLL will get built on the specified host.
2. Click **Finish** when you have completed the configuration. If you do not configure the remote build, then the DLLs will be built locally. You will still be able to debug the code, but you will not be able to run it.
3. From the pop-up menu of the Build Configuration folder, click **Add Server DLL** to open the Server DLL wizard.

4. Name the DLL Claim390S.
5. Set the deployment platform to **390**.
6. Click **Next** to turn to the Libraries to Link With page.
7. Select ClaimC.
8. Click **Next** to turn to the Server Source Files page.
9. Select all the server source files for Claim except for ClaimAIXDOImpl, and add them to the **Items Chosen** list.
10. Click **Finish**.

Building the DLLs

To generate the makefiles and build the DLL files:

1. From the pop-up menu of the Build Configuration folder, select **Generate > All > C++ Default Targets** to generate makefiles for all the DLL files defined in the folder and generate an all.mak file that calls the DLL makefiles.
2. From the same pop-up menu, select **Build > Out-of-Date Targets > C++** to call all.mak and display the progress of the build in a window.
3. Close this window after the build finishes.

For OS/390, the ClaimC.dll and ClaimS.dll files are stored in the specified directory on the specified host.

For AIX, the libClaimC.so and libClaimS.so files are stored in Working\AIX\PRODUCTION.

Defining a container

Define a container to hold the component on the server. You can use the same container definition for both application families.

1. From the pop-up menu of the Container Definition folder, click **Add Container Instance** to open the Add Container wizard.
2. Accept the deployment constraints of **AIX** and **390**.
3. Name the container ContainerOfClaims.

The name is the only information that will be used in the OS/390 installation. The rest of the information in the wizard will be ignored on OS/390, and can be AIX-specific.

4. Click the page title and turn to the Service page.
5. Click **Use RDB Transaction Service**.
6. Click the page title and turn to the Data Access Patterns page.
7. Set the following patterns:
 - **Business Object: Delegating**
 - **Data Object: Delegating**
 - **Cache Service**

These are based on the settings in the ClaimBO business object implementation, and the ClaimAIXDOIImpl data object implementation.

8. Click **Finish**.

ContainerOfClaims appears under the Container Definition folder.

Defining an application family and application for AIX

To define the application family and server application for AIX, follow these steps:

1. From the pop-up menu of the Application Configuration folder, click **Add Application Family** to open the Add Application Family wizard.
2. Name the family ClaimAppFamAIX.
3. Clear the **390** deployment option.
4. Click **Finish**. ClaimAppFamAIX appears under the Application Configuration folder.
5. From the pop-up menu of ClaimAppFamAIX, click **Add Application** to open the Add Application wizard.
6. Name the application ClaimAppAIX.
7. Set the initial state of the application to **stopped**.
8. Click **Next** to turn to the Additional Executables page.
9. Click the **Browse** button to open the Executables to Include dialog box.
10. Locate your Object Builder working directory.
11. From this directory, select:
 - Claim.sql
 - ClaimAIXPO.bnd
12. Click the **OK** button.
13. Click **Finish**. ClaimAppAIX appears under ClaimAppFamAIX.

Defining an application family and application for OS/390

To define the application family and server application for OS/390, follow these steps:

1. From the pop-up menu of the Application Configuration folder, click **Add Application Family** to open the Add Application Family wizard.
2. Name the family ClaimAppFam390.
3. Clear the **AIX** deployment option.
4. Click **Finish**. ClaimAppFam390 appears under the Application Configuration folder.
5. From the pop-up menu of ClaimAppFam390, click **Add Application** to open the Add Application wizard.
6. Name the application ClaimApp390.
7. Set the initial state of the application to **stopped**.

8. Click **Next** to turn to the Additional Executables page.
9. Click the **Browse** button to open the Executables to Include dialog box.
10. Locate your Object Builder working directory.
11. From this directory, select:
 - Claim.sql
 - Claim390PO.bnd
12. Click the **OK** button.
13. Click **Finish**. ClaimApp390 appears under ClaimAppFam390.

Configuring the component with both applications

Configure Claim for AIX:

1. From the pop-up menu of ClaimAppAIX in the Application Configuration folder, click **Add Managed Object** to open the Managed Object Configuration wizard.
2. Select ClaimFileMO ClaimMO. The rest of the fields should fill in with correct defaults.
3. Click **Next** to turn to the Data Object Implementations page.
4. Add ClaimAIXDOImpl.
5. Click **Next** to turn to the Container page.
6. Select ContainerOfClaims.
7. Click **Finish**.

ClaimMO appears under the ClaimAppAIX application.

Configure Claim for OS/390:

1. From the pop-up menu of ClaimApp390 in the Application Configuration folder, click **Add Managed Object** to open the Managed Object Configuration wizard.
2. Select ClaimFileMO ClaimMO. The rest of the fields should fill in with correct defaults.
3. Click **Next** to turn to the Data Object Implementations page.
4. Add Claim390DOImpl.
5. Click **Next** to turn to the Container page.
6. Select ContainerOfClaims.
7. Click **Finish**.

Generate the application installation information:

1. From the pop-up menu of the Application Configuration folder, click **Generate**.

The DDL that defines the applications for System Management is generated into the Working\AIX\ClaimAppFamAIX and Working\390\ClaimAppFam390.

Summary

You have created a component for deployment on either AIX or OS/390, with different versions of some component objects to take advantage of platform-specific development options. You have defined separate build and packaging processes, and have created two separate application packages targeted at two different platforms, based on a single project model.

You can find information on installing the component on the server in the System Administration Guide, “Configure a New Application Environment” chapter, or online in the topic *Installing and configuring a new application*.

You can find information on testing the installed component with a QuickTest client application in “Chapter 13. Testing applications with QuickTest” on page 611.

Chapter 10. Team development

When you develop a stand-alone project, the entire application is contained in a single project, and its development cycle, from component definition through build to packaging, is all handled through that single project. To develop this same application in a team environment, you simply distribute the application's components among a number of interdependent projects. Each project can then be worked on by a separate developer, with the code in the project built as required.

Typically, a team environment begins with a stand-alone project or Rose design, in which the basic structure of the application is defined. Then the stand-alone project is split out into multiple projects, that are accessed and edited through a change control system, and kept up-to-date with regular, automated builds.

If you are working in Rose, then your design can be split out by package, with selected packages in Rose corresponding to projects in the team environment.

If you are starting with a stand-alone Object Builder project, then your design can be split out by package (that is, conceptual groupings of related components) or by layer (that is, different layers of component objects: business objects, data objects, or persistent objects).

If you want to define a set of standard interfaces for which other projects can provide implementations, you can do so by defining the standard interfaces as simple business objects with minimal implementations, and inherit from them as if they were abstract base classes.

Generally, a team environment consists of:

- A number of interdependent projects, which contain the component objects that make up the application.
- An integration project, which defines the build configuration and application packaging options for the application.
- A change control system, which holds all the projects, and controls access to them. The change control system can also be set up to operate on a finer level of granularity, using XML files that represent component elements.
- An automated build process, which extracts all projects and XML files, generates and builds the code, on a daily or nightly basis.

- A project repository, which is the result of the automated build process, and that can be used to resolve inter-project dependencies when a team member checks out and updates a project.

RELATED CONCEPTS

- “Projects and models” on page 17
- “Change control” on page 463
- “XML-based change control” on page 469
- “Model interchange with XML” on page 492
- “Abstract base class inheritance” on page 303

RELATED TASKS

- “Setting up a team environment” on page 457
- “Working in a team environment” on page 480
- “Maintaining a team environment ” on page 490

Developing as part of a team

The basis of team development is the division of an application into multiple projects. Members of the team can then each have their own project, which lets them work with part of the application (contained in their project, accessed in read-write mode) while maintaining relationships with others parts of the application (contained in other projects, referenced in read-only mode).

The main development tasks in a team environment are as follows:

1. “Working with Rose in a team environment”
2. “Setting up a team environment” on page 457
3. “Working in a team environment” on page 480
4. “Maintaining a team environment ” on page 490

RELATED CONCEPTS

- “Chapter 10. Team development” on page 443

RELATED TASKS

- “Developing in Object Builder” on page 19

Working with Rose in a team environment

The following tasks discuss how to use Rose with a team environment, in which you have multiple developers working with multiple projects through a change control system:

- “Exporting a Rose design to a team environment” on page 445

- “Importing a Rose design from a team environment” on page 447

For an introduction to these capabilities, complete the following tutorial series:

- “Tutorial: Exporting from Rose” on page 81
- “Tutorial: Importing into Rose” on page 94
- “Tutorial: Team development with Rose” on page 449

RELATED CONCEPTS

“Rose” on page 64

“The Rose Bridge” on page 69

“Chapter 10. Team development” on page 443

RELATED TASKS

“Chapter 3. Using Rational Rose with Object Builder” on page 63

“Developing as part of a team” on page 444

“Working in a team environment” on page 480

Exporting a Rose design to a team environment



When you export a Rose design to Object Builder, it becomes a project or set of projects, depending on the way you have divided your design into .cat files.

You can specify a separate .cat file for any package in your design. Any package stored in a separate .cat file will be exported to a separate project. The division of your design into .cat files should reflect the project divisions you want to work with. The project is the primary unit of work in a team environment: each .cat file should be as self-contained as possible, and each resulting project should be worked on by only one person at a time.

The OBProjectDirectory property in the IDL page of the package specification notebook identifies the name of the Object Builder project that corresponds to the .cat file. If an absolute path is specified in this property, the Object Builder project will be created in this directory. If a relative path is entered in the property, the specified directory will be created relative to the target project directory you specify at export time. If the property is left blank, a default project name will be generated using the package name and this project will be created as a subdirectory of the target project directory.

You can also specify virtual symbols in the OBProjectDirectory property, or use a mix of virtual symbols and relative paths (for example, \$basedir\myprojects\project1). You can then define the virtual symbol as whatever base directory you want when you are ready to export. If you use the same virtual symbol and paths in both the OBProjectDirectory property and in the path specification for the .cat files (for example, \$basedir\mycats\cat1, \$basedir\myprojs\proj1), you can move the entire

directory at once and only have to modify one virtual symbol definition. Using virtual symbols makes your model more portable.

To specify a separate .cat file for a package, follow these steps:

1. Select the package in a Class Diagram in the Logical View.
2. Click **File > Units > Control** *package* (where *package* is the name of the package you selected).
3. Specify a path and file name for the .cat file that will contain the package.
4. Click **Save**.
5. From the package's pop-up menu, click **Open Specification** to open its Specification notebook.
6. Turn to the IDL page, and set the property `OBProjectDirectory` to the project directory path. The project directory can be defined as:
 - An absolute directory path. For example, `e:\projects\package1`.
 - A relative directory path. The project will be a subdirectory of the target project you specify during export. For example, the value `package1` creates the project directory `e:\myproject\package1` (if you specified `e:\myproject` as the main export directory when you exported).
Generally speaking, you should not use relative paths. The result (nested project directories) can be hard to manage.
 - A virtual path mapping. You can also set the property to a virtual path mapping, as defined in Rose. Click **File > Edit Path Map** to define a virtual path mapping in Rose. For example, you can specify `PACKAGE1_PATH` in the Specification notebook, and set it to resolve to `e:\projects\package1` in the Virtual Path Map window.
 - A mix of virtual path map and relative path For example, you can specify `BASE_DIR\Package1` in the Specification notebook, and set `BASE_DIR` to resolve to `e:\myprojects`, resulting in `e:\myprojects\Package1` when you export.
7. Click **OK** to apply your changes and close the notebook.

The package and its contents will now be stored in the .cat file, rather than in the main .mdl file, and its contents will be exported to the project directory you specified.

You can exclude portions of your design from the export (packages or classes) by setting the property `BridgeToOB=FALSE` in the package or class specification notebook. Set `BridgeToOB=FALSE` for a package to prevent all packages and classes contained within the package from being mapped into Object Builder. By default, your entire design is exported.

To export, follow these steps:

1. Start Rose, and open the design you want to export.

2. Select **File > Export to Object Builder**. The Rose Bridge wizard opens to the Export from Rose to Object Builder page.
In the **Source Model** field, the current Rose model file is selected by default.
3. Provide any virtual path mapping information needed to locate any separate .cat files. If you used Component Broker frameworks in your design, you will need to provide the mapping for BOSS_PATH. You can check the value for any virtual path maps in Rose by clicking **File > Edit Path Map**.
4. In the **Target Project** field, specify a main project directory. Any portions of your design which are *not* stored in separate .cat files will be exported to this project. The portions of your design that *are* stored in .cat files will be exported to the directories specified in the package OBProjectDirectory property.
5. Click **Next**.
6. Review the export structure. You can review which packages and classes are being exported (as defined by the BridgeToOB property for specific packages and classes), and review the target project directories (as defined by the .cat file structure within Rose, and the package OBProjectDirectory properties).
7. Click **Finish**.

Your model is exported to projects in the specified directories. Your model is saved in Rose as part of the export process.

RELATED CONCEPTS

“The Rose Bridge” on page 69

“Rose Bridge import” on page 89

“Projects and models” on page 17

“Chapter 10. Team development” on page 443

RELATED TASKS

“Setting up a team environment” on page 457

“Importing Component Broker frameworks” on page 68

“Tutorial: Team development with Rose” on page 449

“Exporting a design from Rose” on page 80

“Importing XML” on page 389

“Working with an exported design” on page 88

RELATED REFERENCES

“Rose to Object Builder mapping rules” on page 97

Importing a Rose design from a team environment

You can create or update a Rose design by importing Object Builder projects into Rose. If the imported projects were originally created by a Rose export,

then the new Rose model created by the import will mirror the information in the original, exported Rose model's Logical View. If your original model has additional information in other views, you can consolidate the two models (the original exported one, and the newly imported one) using the Rose Update feature.

If the projects were created only in Object Builder, or the original design is unavailable, then the import process creates a new Rose design. When the import is completed, each project maps to a package in Rose with an associated .cat file, and each business object interface maps to a class in one of the packages. You can then work with the design in Rose, and export the changes back to Object Builder.

To import a set of interdependent Object Builder projects (that is, projects in a team environment) into Rose, follow these steps:

1. Select **File > Import from Object Builder**. The Rose Bridge wizard opens to the Source page.
2. Specify the projects you want to import:
 - a. Type the project directory path in the Project Directory field.
 - b. Click **Set** to add the path to the import list.
3. Click **Next**.
4. Enter the name of the Rose model (.mdl) file you are importing to.
5. If your projects have equivalent .cat files in your design (that is, they were created by exporting .cat files from Rose), then you need to specify any virtual path mapping information used to locate the .cat files. For each virtual path map, enter the symbol and its mapping to an actual directory path.
6. Click **Finish**.

The project in the directories you selected are imported into the Rose model file you specified, and overwrite any previously existing .cat files.

If the imported project was created by export from Rose, and the original Rose model contains information in other views besides the Logical view, then you should consolidate the new model with the original model before doing any more design work.

To merge the new model with the original model, follow these steps:

1. Click **File > Open** to open the original Rose model.
2. Specify the original .mdl file and click **Open**.
3. Click **File > Update** to apply updates from the changed model.
4. Specify the updated .mdl file (created by the import in the previous task) and click **Open**.

Your model now contains the entire updated design, and you can continue your design work. Because the Rose Bridge preserves diagram information in the Logical View, you may have multiple diagrams with the same information after the update. This will cause no problems, but you can delete the duplicate diagrams if desired. No structural information will be lost by the deletion.

Note: If you have Rose 98i or Rose 2000, you can use the Model Integrator provided with Rose to merge the models. Please read the documentation provided with Rose for an explanation of this tool.

When you are ready to switch back to Object Builder, you can export the design back to Object Builder by selecting **File > Export to Object Builder**.

RELATED CONCEPTS

- “Object Builder” on page 1
- “Projects and models” on page 17
- “Rose” on page 64
- “The Rose Bridge” on page 69
- “Chapter 10. Team development” on page 443

RELATED TASKS

- “Exporting a design from Rose” on page 80
- “Exporting a Rose design to a team environment” on page 445
- “Tutorial: Team development with Rose”

RELATED REFERENCES

- “Object Builder to Rose mapping rules” on page 123

Tutorial: Team development with Rose

Objectives

- To create an object relationship in Rose.
- To export from Rose to multiple projects.
- To import from multiple projects to Rose.

Before You Begin



This tutorial is a continuation of the tutorial sequence:

1. “Tutorial: Exporting from Rose” on page 81
2. “Tutorial: Importing into Rose” on page 94

You must complete the previous scenarios before attempting this one.

You need the following installed on your system:

- A CB Server

- A CB Client
- CB tools, including Samples
- DB2 Universal Database
- DB2 SDK
-  VisualAge for C++ and (for Java applications) VisualAge for Java
-  IBM C and C++ Compilers for AIX V3.6.4 and (for Java applications) VisualAge for Java
- Rational Rose 98, Rose 98i, or Rose 2000.

You need Rational Rose installed and set up to work with Object Builder, as described in the task “Setting up Rose 98” on page 65 or “Setting up Rose 98i and Rose 2000” on page 66.

For a complete list of prerequisites that you must install on your system, see the *Getting Started with Component Broker* documentation for the appropriate platform.

You should be familiar with the Component Broker programming model, as described in the *IBM Component Broker Programming Guide*. At minimum, you should have read chapter 1, the introduction to the programming model.

You should be familiar with Rational Rose. If you are not familiar with the tool, take the Rose tutorial included with the software.

You should be familiar with Object Builder. If you are not familiar with the tool, run through one of the introductory tutorials, for example, “Tutorial: Creating a component with transient data” on page 39.

Sample files

There are no equivalent samples for this exercise. The exercise assumes that you have created a design by importing a project from Object Builder, as described in the previous tutorial.

Description

In this exercise, you will extend the Rose model for Agent to include a second component, Customer, which Agent has a one-to-many relationship with (each Agent can have multiple Customers). You will then export the modified model to a new set of interdependent project directories, modify the exported components in Object Builder, and import the changes.

Note that when you are creating more complicated team environments (with multiple projects, each containing multiple components), you will want to minimize the number of cross-project dependencies. In Rose terms, you would

group your classes into base-level packages with associated .cat files, and minimize the cross-package relationships and references.

For this exercise, you will complete the following tasks:

1. Create a package in Rose, and assign it to a .cat file.
2. Add the Customer class to the package.
3. Edit the Agent class.
4. Export the model to Object Builder projects.
5. Edit the Agent component.
6. Edit the Customer component.
7. Import the projects and apply the changes to your Rose model and .cat file.

Creating the RelComp package

Create a new package to represent a project in Object Builder:

1. Open the model you created in the previous tutorial (for example, e:\tutorials\rosemodels\importagent.mdl).
The existing Agent class appears in the Logical view's Main class diagram.
2. Click **Tools > Create > Package**.
3. Click on the background of the class diagram to create a new package.
4. Type over the default name (NewPackage) to name it RelComp (as in "Related Components").
5. Click **File > Units > Control RelComp** to display the Save dialog box for the package.

You can now specify a location for the .cat file in which the package's information will be stored. By assigning the package a separate .cat file, you are specifying that the package represents a separate project in your target development environment.

6. Provide a name and path for the .cat file (for example, e:\tutorials\rosemodels\relcomp.cat).
7. Click **Save**. The package information will be stored in the specified .cat file, and exported as a separate project to Object Builder.

Packages that represent projects must have an assigned project directory, to which their contents can be exported. The project directory can be specified as a virtual path mapping, a relative path (a subdirectory of the Object Builder base project directory), or an absolute path. For this exercise, you will assign an absolute path for the project directory.

Assign a project directory to the new package:

1. From the pop-up menu of the RelComp package, click **Open Specification** to open its Specification notebook.

2. Turn to the IDL page.
3. Set the `OBProjectDirectory` property to a project directory path (for example, `e:\tutorials\roserel`).
4. Click **OK** to apply the changes and close the notebook.

The contents of the package are now associated with the Object Builder project directory you specified (although the project directory does not yet exist).

Adding the Customer class

Start Rose, and add the Customer class, with three attributes.

Add the Customer class:

1. Locate the RelComp package in the tree view (left-hand pane in Rose).
2. From the pop-up menu of the RelComp package, click **New > Class**, and name the new class Customer.
3. In the tree view, locate the Logical View - Main class diagram.
4. Double-click on the Logical View - Main class diagram. It opens as a Logical View window.
5. Click **Query > Add Classes** to open the Add Classes window.
6. Add Customer to the diagram, and click **OK**

Add the attributes `custName`, `sales`, and `customerNo`:

1. From the pop-up menu of Customer in the diagram, click **New Attribute**. A placeholder attribute is added (named `name`, type of `type`, initial value of `initval`).
2. Type over each of the values for the new attribute, naming it `custName`, with type `string`.
3. Click elsewhere in the diagram to apply the changes.
4. Add the following attributes in the same way:
 - `float sales`
 - `long customerNo`

Customize the attributes. For each attribute of Customer:

1. Double-click on the attribute (in the tree view) to open its Specification notebook.
2. On the General page, set export control to **Public**.
3. Click **OK**.

For the `custName` attribute of Customer:

1. Double-click on the attribute to open its Specification notebook.
2. On the DDL page, set Length to 100.
3. Click **OK**.

The class and its attributes are defined. You could add Component Broker properties to the class and its attributes in order to customize the mappings to component elements, but for this exercise simply accept the default.

Editing the Agent class

Add a one-to-many relationship from Agent to Customer:

1. Click **Tools > Create > Aggregate Association**. Your mouse pointer changes to an arrow.
2. Click and hold on Customer, and then drag to Agent, to draw the aggregation. When you release the mouse button, an arrow is drawn from Agent to Customer.
3. Double-click on the arrow to open Aggregation Specification notebook.
4. Turn to the Role B Detail page, and set the cardinality to n (one Agent can have many Customers).
5. Turn to the IDL A or B page. Review the values for the aggregation properties:
 - **MapAsObjectRelationship=True**
The aggregation will map to an object relationship in Object Builder. If this value were false, the aggregation would map to an attribute of type sequence in Object Builder.
 - **RelationshipImplementation=Local Persistent Reference**
The relationship will be implemented in the business object implementation as a local persistent reference, rather than mapping to a database query.
6. Click **OK** to close the diagram and apply your changes.

Agent now has a one-to-many object relationship to Customer.

Exporting to Object Builder

You are ready to export the classes to Object Builder. A separate project will be created for the RelComp package, and the project dependencies will be set accordingly.

To export to separate Object Builder projects, follow these steps:

1. Click **Save As** to save and rename your model (for example, as e:\tutorials\rosemodels\AgentTeam.mdl). The model file contains the information that defines Agent, and has a reference to the .cat file that defines Customer.
2. Click **File > Export to Object Builder**. The Rose Bridge Export wizard appears.
3. In **Target Project**, specify a directory to export to (for example, e:\tutorials\roseteam\). The Rose Bridge will create the directory if necessary, and turn it into an Object Builder project directory.

4. Accept the default for the other options.
5. Click **Finish**. Your existing Rose model is automatically saved as part of the export process.

Project directories are created for the two components. Your project directory structure should look something like this:

- e:\tutorials\roserel
Defines the Customer component, and has a dependency on e:\tutorials\roseteam to support Customer's relationship to Agent. The project directory was defined in the RelComp package's Specification notebook, on the IDL page, with the OBProjectDirectory property.
- e:\tutorials\roseteam
Defines the Agent component, and has a dependency on e:\tutorials\roserel to support Agent's relationship to Customer. The project directory was defined during the export process, in the Rose Bridge wizard.

You are ready to open the projects in Object Builder, review the results of the export, and edit the components in Object Builder.

Working with Customer

Open Customer's project:

1. Start Object Builder.
2. In the Open Project wizard, specify the project directory created by the export for the RelComp package (for example, e:\tutorials\roserel).
3. Click **Next** to turn to the Project Dependencies page. Note the dependency on Agent's project (for example e:\tutorials\roseteam). This has been added to support Customer's relationship with Agent.
4. Click **Finish**. The project for Customer opens.

Edit Customer's attributes:

1. Expand the User-Defined Business Objects, and locate the Customer interface (under the Customer file).
2. From the pop-up menu of the Customer interface, click **Properties** to open the Business Object Interface wizard.
3. Click the page title and turn to the Attributes page.
4. Set customerNo to be read-only.
5. Click **Finish**.

Add a key and copy helper:

1. From the pop-up menu of the Customer interface, click **Add Key** to open the Key wizard.
2. Add the customerNo attribute to the key attributes list.
3. Click **Finish**.

4. From the pop-up menu of the Customer interface, click **Add Copy Helper** to open the Copy Helper wizard.
5. Add all attributes to the copy helper attributes list.
6. Click **Finish**.

Save your changes and close Object Builder:

1. Click **File > Save**.
2. Click **File > Exit**.

You have made your changes to the project, saved them, and closed Object Builder. You are ready to work with Agent's project.

Editing Agent

Review the exported Agent component.

Open Agent's project:

1. Start Object Builder.
2. In the Open Project wizard, specify the project directory created by the export for the Rose model (for example, e:\tutorials\roseteam).
3. Click **Next** to turn to the Project Dependencies page. Note the dependency on Customer's project (for example, e:\tutorials\rosereel). This has been added to support Agent's relationship with Customer.
4. Click **Finish**. The project for Customer opens.

Edit the interface:

1. Expand the User-Defined Business Objects, and locate the Agent interface.
2. Click on the interface. Note the relationship to Customer that appears in the Methods pane, in the User-Defined Relationships folder.
3. From the pop-up menu of the Customer interface, click **Properties** to open the Business Object Interface wizard.
4. Click the title and turn to the Attributes page.
5. Change the name of the agentID attribute to id .
6. Delete the contactInfo attribute.
7. Click **Finish**.

Review the implementation:

1. Locate the AgentBO implementation (under the Agent interface in the Tasks and Objects pane).
2. From the pop-up menu of AgentBO, click **Properties** to open the Business Object Implementation wizard.
3. Click the page title and turn to the Object Relationships page.

4. Note that the relationship is implemented as a local persistent reference. This matches the RelationshipImplementation property set for the relationship in Rose.
5. Click **Finish**.

Save your changes and close Object Builder:

1. Click **File > Save**.
2. Click **File > Exit**.

You have made your changes to the project, saved them, and closed Object Builder. You are ready to import your changes from both projects into Rose, and review their effect on your Rose model.

Importing into Rose

To import your changes into Rose, follow these steps:

1. Start Rose.
2. Click **File > Import from Object Builder**. The Rose Bridge Import wizard opens.
3. In the Object Builder Project Directories field, type the path for the Agent component's project (for example e:\tutorials\roseteam\) and click **Set**. The project is added to the import list.
4. Add the Customer project directory (for example e:\tutorials\rosereel\) in the same manner.
5. Click **Next**.
6. In the Output File field, type the name of the Rose model you want to import to (for example e:\tutorials\rosemodels\ImportAgentTeam.mdl). You do not need to provide any virtual path mapping information, because you did not use virtual path maps in your original Rose design (as created in the previous tutorials).
7. Click **Finish**.

The Rose Bridge imports Agent's project into the specified model (creating a new model), and imports Customer's project into the existing .cat file (overwriting the existing .cat file).

Because the import process only maps information to the Logical View, the import process assumes that you do *not* want to overwrite the existing model (which could potentially contain unmapped information in other views). The import process does overwrite the .cat file, however, since .cat files only contain Logical View information.

In this case, all your information for both components was in the Logical View: there is no need to merge with the original model, and you could discard it if you wanted.

Even within the Logical View, there are some elements that do not have Object Builder equivalents: every time you import from or export to Object Builder, any unmapped information is preserved in a file called *modelname.xml*, which is stored in the Object Builder project's \XMI directory. When you store your project in a change control system, be sure to preserve the contents of the \XMI directory as well as the Object Builder model files, or Rose design information will be lost.

Reviewing the changes

Your changes in the Object Builder projects are applied, and have had the following effect:

- Customer now has the IDL properties `CreateKey` and `CreateCopyHelper` properties set to `True` (reflecting your creation of `CustomerKey` and `CustomerCopy`, in Object Builder)
- Customer's attribute `customerNo` now has the DDL property `isPrimaryKey` set to `True` (reflecting your inclusion of the attribute in `CustomerKey`, in Object Builder).
- Agent's attribute `agentID` has been renamed to `id`.
- Agent's attribute `contactInfo` has been deleted.

Summary

You have created a team environment with two interdependent project directories, created a cross-project object relationship, and applied changes in the team environment to your Rose model.

Setting up a team environment

To set up a team environment, you typically start by defining the structure of your application (either in a Rose design, or in a single Object Builder project), then divide the structure into working units (either by exporting from Rose, or by splitting up the single Object Builder project). Once you have the directory structure that holds your design defined, you can add an integration or build project. You can then store the structure in a change control system, set up an automated build process, and finally set up the individual development machines.

The tasks involved in setting up a team environment are as follows:

1. "Splitting up a project for team development" on page 459
2. "Adding an integration project to a team environment" on page 460
3. "Setting up a change control process" on page 464
4. "Setting up XML-based change control for CMVC" on page 471
5. "Setting up an automated build process" on page 465
6. "Setting up a team development environment" on page 467

You can also set up a team environment starting from Rose, as discussed in “Working with Rose in a team environment” on page 444.

RELATED CONCEPTS

- “Chapter 10. Team development” on page 443
- “Project divisions in a team environment”
- “Change control” on page 463
- “XML-based change control” on page 469

RELATED TASKS

- “Developing as part of a team” on page 444
- “Working in a team environment” on page 480

Project divisions in a team environment

When you create an application in a team environment, the contents of the application needs to be split up into multiple projects. You can choose to split along package lines or component layers, or combine the approaches:

- Packages
Your application design can be viewed in terms of categories or groupings of related components, which serve to partition the logical model of your application. In UML terms, these categories are packages. If you created your application design in Rational Rose, you can export the design directly to a team environment. Any packages that are stored in separate .cat files in Rose will automatically be exported to separate projects. You can set the directory for the exported project in the package’s specification notebook (IDL page, OBProjectDirectory property).

If your team development roles align with divisions in the design, this is the main strategy you will use.

- Component layers
Each component consists of a behavior layer (the business object), a data layer (the data object), and a persistence layer (the persistent object). You can split out a component into its separate component objects, and maintain them in separate, interdependent projects.

If your team development roles align with component layers (for example, an object-oriented designer at one end versus a database administrator at the other), this is the main strategy you will use.

You can mix strategies within a team environment. You will also want an additional, separate project from which you can coordinate application-wide builds and application packaging.

There are two general rules to use when deciding on project divisions:

- Avoid overlapping ownership
Each project should have a single clear owner within your team, whenever

possible. Only one person should work on a project at a time, so overlapping ownership creates access conflicts.

- **Minimize cross-project dependencies**
Components that are closely related should stay in the same project whenever possible. Some component dependencies, such as foreign key pattern relationships, nearly always belong in the same project.

RELATED CONCEPTS

“Chapter 10. Team development” on page 443

RELATED TASKS

“Setting up a team environment” on page 457

“Splitting up a project for team development”

“Changing project divisions” on page 496

Splitting up a project for team development

To divide an existing project into separate projects, follow these steps:

1. Open the existing project.
2. Click **File > Export Model**. XML files are exported for the project, as described in “XML interchange files” on page 493.

The exported files are placed in the \Export subdirectory of the project directory.

3. Create a set of project directories that represent the groupings of component objects you want, under a single parent directory.

For example, you could create the project directories

e:\allprojects\policyBO, e:\allprojects\carpolicyBO, e:\allprojects\allDOs, e:\allprojects\integration.

4. In each project directory, create a subdirectory (for example, \Import).
5. Place the component XML files for each grouping in the subdirectory of its equivalent project.

You can place as many or as few XML files in each subdirectory as you want, depending on the way you want to organize the project contents, and following the project division guidelines (avoid overlapping ownership, minimize cross-project dependencies).

In the Policy example, you could divide the files as follows:

- e:\allprojects\policyBO\Import
udbo.Policy.xml
- e:\allprojects\carpolicyBO\Import
udbo.CarPolicy.xml
- e:\allprojects\allDOs\Import
uddo.PolicyDO.xml, uddo.CarPolicyDO.xml,
uddbschema.PolicyGroup.xml, uddbschema.CarPolicyGroup.xml

- e:\allprojects\integration
uddll.PolicyS.xml uddll.PolicyC.xml, udaf.MyAppFamily.xml
6. From the command line, run the obimport command to create a set of interdependent project models based on the XML files in each project's subdirectory (for example \Import). For example:
- ```
obimport -X -d Import -newuuid e:\allprojects\policyBO
e:\allprojects\carpolicyBO e:\allprojects\allDOs e:\allprojects\integration
```
- This example would create four projects, by importing the XML files found in the \Import subdirectories of the listed directories. The -newuuid option guarantees that the imported files will have unique identifiers across the set of projects.

Once you have split the project into a set of projects in a team environment, you can continue with your team environment set up. You need to choose a change control unit, and a system to store the units in. You need to set up the team environment, and the development machines, to support either local dependency resolution (with project dependencies extracted from the change control system) or remote dependency resolution (with project dependencies resolved by a project repository on a shared network drive). You need to set the makefile generation preferences for each Object Builder installation to reflect a team environment.

#### **RELATED CONCEPTS**

- “Chapter 10. Team development” on page 443
- “Project divisions in a team environment” on page 458
- “Projects and models” on page 17
- “Model interchange with XML” on page 492

#### **RELATED TASKS**

- “Exporting XML” on page 387
- “Importing XML” on page 389
- “Changing project divisions” on page 496
- “Adding an integration project to a team environment”

#### **RELATED REFERENCES**

- “XML interchange files” on page 493

## **Adding an integration project to a team environment**

### **▶ CHANGED**

If you created your team environment by splitting up an existing project, then your integration project already exists, and simply contains the build and application configuration information from the original project. However, if you created your team environment by exporting a model from Rose, you will need to add a new integration project, as described here.

The integration project will define the build configuration options and makefiles for all the components defined in your projects. Typically, it will also contain the application configuration information, when you reach the stage of packaging the application. To add an integration project, follow these steps:

1. Create a new project. It should be part of the same directory structure as your other projects (that is, they should all be under the same parent directory). In the Open Projects wizard, Project Dependencies page, list all the projects in your team environment as dependencies.
2. Once the new project is open, click **File > Preferences** to open the Preferences notebook.
3. Click on the Tasks and Objects node in the Preferences tree view.
4. Under **Environment**, set the **Team Environment** option. This allows the project's build process to locate the generated code in the dependency projects' \Working directories.

In order to locate the other projects' generated code, the integration project's makefiles will use absolute paths instead of relative ones. These absolute paths are, by default, the paths where the dependent models are found on the Object Builder Model Path. You can change these paths for each platform for the current project on the Build Location page of the Build Configuration wizard. If your directory structure changes, you will need to regenerate the makefiles.

5. Add client and server DLLs for all the components in your application. If a DLL is already defined in one of the dependency projects, then you must use a different name for the DLL configuration node in the integration project, but you should use the same name for the built DLL file.
6. Add application families and applications, and configure the managed objects of the various components in the other projects.
7. Save and close the project.

The project will be used as the starting point for any regular automated builds, and for packaging the application.

#### **RELATED CONCEPTS**

"Chapter 10. Team development" on page 443

"Projects and models" on page 17

#### **RELATED TASKS**

"Setting up a team environment" on page 457

"Splitting up a project for team development" on page 459

"Setting up an automated build process" on page 465

## Cross-project dependencies

► NEW

### Building multiple projects

When you work with multiple projects within Object Builder, some lower-level projects that provide services for higher-level projects must be built first (before the higher-level ones). This is especially important if you have modules within your projects. Object Builder provides a Contents Ordering page for you to specify the order in which the DLLs have to be built. You can design your build system in such a way that the projects get built in an order, which is compatible with the logical layering of the system.

► JAVA

For Java compile cycles, the normal way of incrementally building Java projects is in this order:

1. Build JAR files for the lower layers
2. Then, include these JAR files on the `-classpath` command line argument when compiling higher level classes

This is automatically done for you by Object Builder when you specify the order that has to be used during the build. Object Builder resolves these dependencies between different levels of projects.

► C++

For the C++ build phases, the order of the DLLs is not very critical for the following reasons:

- compiling C++ sources (pass4) only requires access to header files of other imported files
- the creation of DLLs (pass6) does not require that other DLLs on which one depends are built yet. Only the `.lib` files have to be available, and these also get built in isolation during the previous phase (pass5).

#### RELATED TASKS

“Specifying the order of a build” on page 558

“Building the DLLs” on page 558

“Configuring builds” on page 549

“Launching a remote OS/390 build” on page 572

“Generating a makefile” on page 556

“Packaging applications” on page 574

## Change control

When you set up a team environment, you will need to provide a change control mechanism to ensure that your team members are always working with the latest versions of their projects. Whatever change control mechanism you use, you need to set a consistent unit of change control that can be used throughout your change control system. The unit you choose will be checked out and locked while a team member is working with it, and then checked back in and released when the team member has finished working with it.

You can set up your change control system to manage information in any of the following units. Whichever unit you choose should be used consistently throughout your system.

- The model files for a project (the contents of a project's `\Model` directory). This is the recommended unit of change control for most team development environments, and most of the documentation assumes you are doing so. The model files can simply be opened and worked with. If you are using external files to provide method implementations, it may be easier to package the entire `\Model` directory (for example, in a `.zip` file) and use that as the unit of change control.
- The model files, plus the `.xmi` files for a project (the contents of a project's `\Model` directory, as well as the contents of a project's `\XMI` directory). If you are working extensively with Rational Rose, this is the recommended unit of change control.
- The XML files for a project's contents  
You can check in or check out files for objects in the Tasks and Objects pane as described in "XML interchange files" on page 493. This allows you to store and exchange information at a granular level, and allows you to integrate Object Builder with your change control system using customizable check-in and check-out wizards. If you have a simple team development environment with static project divisions, and overlapping team roles that require this level of granularity, this is the recommended unit of change control.
- The generated source files for a project (the contents of a project's `\Working` directory).  
This assumes that each team member is maintaining a single copy of their project model. The generated code is stored for backup purposes only, and for regular automated builds.

Once your change control system is in place, and your project information is stored in it, you can use it to manage your team environment.

Typically, you should use a daily build process to extract all projects, build them, and make the result available to team members. A team member who

wants to make a change to a project can then check out the project from the change control system, and use the daily build structure to resolve the project's dependencies.

#### **RELATED CONCEPTS**

- "Chapter 10. Team development" on page 443
- "XML-based change control" on page 469
- "Projects and models" on page 17
- "Model interchange with XML" on page 492
- "Project divisions in a team environment" on page 458

#### **RELATED TASKS**

- "Setting up a change control process"
- "Setting up XML-based change control for CMVC" on page 471
- "Importing edited source files" on page 385

#### **RELATED REFERENCES**

- "XML interchange files" on page 493

## **Setting up a change control process**

Once you have created your team directory structure and skeleton projects (either through export from Rose, or by splitting up an existing project), you can set up a change control process, to ensure that only one person makes changes to a project at a time.

Typically, you should use a daily build process to extract all projects, build them, and make the result available to team members. A team member who wants to make a change to a project can then check out the project from the change control system, and use the daily build structure to resolve the project's dependencies.

Setting up a change control process requires the following general steps. Most of these steps are discussed in detail in their own tasks. This list gives you an overview of how the steps fit together, from a change control perspective.

1. Create the team directory structure (that is, a set of project directories that hold the elements of your application). You can create the structure either by exporting a design from Rose, or by taking a design in an existing Object Builder project, and splitting it up into multiple projects. This is discussed in other tasks.
2. Add an integration project to the team directory structure. This is discussed in another task.
3. Select a unit to use for change control (for example, the contents of each project's \Model directory).

4. If you selected XML files as your unit of change control, customize the check-in, check-out, and extract wizards in Object Builder. This is discussed in another task.
5. Check all projects, or their individual elements, into the change control system.
6. Set up a daily or nightly build process, that will extract all projects or project elements, generate code for the entire application using the integration project, and build the application DLLs. The resulting project repository (all projects, with their generated and built code) needs to be available for team members to access (for example, as a zip file in the change control system, or as a directory structure on the LAN).

#### **RELATED CONCEPTS**

“Change control” on page 463

“Chapter 10. Team development” on page 443

“Projects and models” on page 17

#### **RELATED TASKS**

“Setting up a team environment” on page 457

“Exporting a Rose design to a team environment” on page 445

“Splitting up a project for team development” on page 459

“Adding an integration project to a team environment” on page 460

“Setting up XML-based change control for CMVC” on page 471

“Setting up an automated build process”

## **Setting up an automated build process**

You can use an automated build process to create and update a project repository, which can be used to resolve the dependencies of a project being edited.

**Note:** It is advisable to run the Consistency Checker before you generate makefiles, or start building, to ensure that the configuration for your client DLL is correct. This is crucial particularly if your client DLL does not have at least one IDL file selected.

The automated build process needs to do the following:

1. Extract all projects from the change control system.  
If you are using XML-based change control, then you need to extract all XML files, and import them into their respective projects:
  - a. In each project directory, create an Import subdirectory.
  - b. Place the extracted XML files for a particular project in that project’s \Import subdirectory.
  - c. Run “obimport” on page 664 with the -x option to import the files into the projects, with resolution of cross-project references. For example:

```
obimport -x e:\projects\projectA e:\projects\projectB
e:\projects\projectC
```

2. Generate code for all projects, using “obgen” on page 685 with the -linked option. For example:

```
obgen -pF:\projects\projectA -aAll -tNT -linked
obgen -pF:\projects\projectB -aAll -tNT -linked
obgen -pF:\projects\projectC -aAll -tNT -linked
```

The above commands generate the code for all objects (-aAll) in projectA, projectB, and projectC (the equivalent of selecting **Generate > All** from the pop-up menu of the User-Defined Business Objects folder in each project), and also generates makefiles and SM DDL files for the DLLs and applications defined in the projects. The code is generated for the platform Windows NT (-tNT), and placed in each project’s \Working\NT\ directory (-linked).

3. Generate the makefiles for the application integration project. For example:

```
obgen -pF:\allprojects\Integration -aMake -linked -tNT
```

The above command generates the makefiles (-aMake) defined in the Integration project. The -linked option generates the makefiles with absolute paths to the code found in the other projects’ \Working\NT directories (if you have not specified your own paths in the Project Build Location page of the Build Configuration Properties wizard), rather than generating makefiles that assume the generated code is in the current project’s \Working\NT directory.

4. Build all the DLLs defined in the application integration project. For example, for a Windows NT machine:

```
nmake -f F:\allprojects\Integration\Working\NT\all.mak
```

The DLLs, jar files, and any other targets defined in the makefiles are built in the integration project’s \Working\NT\PRODUCTION directory.

**Note the following points:**

- If your compile command fails due to an incorrect DB2 user ID and password error, run the following command before you run the make (AIX) or nmake (NT) command:

```
▶ WIN set IVB_DB2AUTH=USER test USING password
```

```
▶ AIX ▶ SOLARIS ▶ HP-UX export IVB_DB2AUTH="USER test USING
password"
```

test is the user ID that you log on to DB2 with, and password is the password for your user ID.

- ▶ SOLARIS ▶ HP-UX On Solaris and HP-UX, you have to manually start the ‘make’ process that is used to build an Object Builder model. Where



authentication is required to complete the bind, you must run make from within an authenticated DB2 command shell, instead of from within a vanilla shell.

5. Make the full extracted directory structure available, including the \Model directories, and the \Working directories with the generated code and built DLLs.

For example, you could export F:\allprojects on the network, or zip the contents of the directory and place the zip file in the change control system.

The following sample build script extracts the project model directories for four projects, including an integration project, generates their code, generates the makefiles, builds the code, and creates a zip file:

**allprojectsbuild.bat**

```
<Extract all projects from your change control system>
obgen -pF:\allprojects\projectA -aAll -tNT -linked
obgen -pF:\allprojects\projectB -aAll -tNT -linked
obgen -pF:\allprojects\projectC -aAll -tNT -linked
obgen -pF:\allprojects\Integration -aMake -linked -tNT
nmake -f F:\allprojects\Integration\Working\NT\all.mak
zip latest.zip F:\allprojects* -r
<publish latest.zip>
```

**RELATED CONCEPTS**

“Chapter 10. Team development” on page 443

**RELATED TASKS**

“Generating code from the command line” on page 684

“Adding an integration project to a team environment” on page 460

“Setting up a team environment” on page 457

## Setting up a team development environment



Once the projects in the team environment have been stored in a change control system, and an automated build process has begun producing regularly updated project repositories, you can set up the development machines to be part of the team environment.

On each development machine, set up the team environment as follows:

1. Create a local directory that will hold the project repository created by the nightly build (for example, f:\allprojects\). Each team member will be responsible for updating their copy of the repository when required.
2. Create another local directory to hold any projects you check out from the change control system for editing (for example, f:\currentprojects\).

On each development machine, set up Object Builder’s search path:

1. Start Object Builder. The Open Project wizard opens.

2. In the Model Search Path field, include first the current projects directory, and then the project repository directory. For example:  
f:\currentprojects;f:\allprojects;

The directories, and their subdirectories, will be searched for project dependencies in the order they are listed. For example, if you check out three interdependent projects into f:\currentprojects\, then the duplicates of those projects in the f:\allprojects\ directory are ignored, because their dependencies on each other are resolved before the f:\allprojects\ directory is searched. Any additional dependencies, beyond just the three checked out projects, will be resolved in the f:\allprojects\ directory.

On each development machine, set up Object Builder for a **Team Environment**:

1. Open Object Builder, with any project.
2. Click **File > Preferences** to open the Preferences notebook.
3. Click on the Tasks and Objects node in the tree view.
4. Under **Environment**, set the **Team Environment** option. This allows the project's build process to locate code and makefiles in other project \Working directories, to resolve makefile dependencies correctly.

In order to locate code in other projects' Working directories, the generated makefiles will use absolute paths instead of relative ones. If your directory structure changes, you can edit the file prjdefs.mk to reflect the new paths, or simply regenerate the makefiles.

For each target platform, you can override the default working directory paths emitted into prjdefs.mk on the Build Location page of the Build Configuration wizard.

If a business object depends on other business objects, the paths to the directories that contain those models is written by Object Builder into the file prjdefs.mk. If you then copy the generated source another location (for example, to another platform), you must edit prjdefs.mk to reflect the paths on the new location. Alternatively, you can customize the path that is emitted for each platform on the Build Location page in the Build Configuration wizard.

5. Save and close the project.

#### **RELATED CONCEPTS**

- "Chapter 10. Team development" on page 443
- "Projects and models" on page 17

#### **RELATED TASKS**

- "Setting up a team environment" on page 457
- "Working in a team environment" on page 480
- "Maintaining a team environment " on page 490

---

## XML-based change control

You can integrate Object Builder with the change control system of your choice, using XML wizards. The wizards accept input from the user, and use the input to construct a command line, that checks in or checks out the XML file for the current Object Builder element.

This is *not* the recommended unit of change control for this version of Object Builder, especially if you are using Rational Rose: the XML does not contain information on Rational Rose mappings. If you have a team environment with many projects, or use additional tools with Object Builder, you should use the project directory, specifically its `\Model` and `\XMI` subdirectories, as the unit of change control.

If you decide to use XML-based change control, you will still need to break your application up into multiple projects, for the sake of efficient access. The projects do not need to be under change control themselves, but they do need to be regularly refreshed to reflect any changes made in the change control system to the XML files. You can create a batch process or shell script to accomplish this. The process needs to extract all XML files from the change control system, import them into their respective projects, and then publish the projects (make them available for use by the team).

Once you have your projects updated and their contents stored as XML files in your change control system, you can access the system through Object Builder using change control wizards. Open a project in Object Builder, check out a file or series of files (that are already defined in the project, but become locked in the change control system once they are checked out), and work normally within the project and its team environment until you are ready to check in your changes.

There are three different XML formats involved in the XML-based change control process:

- **Object Builder's XML format**

Defines the elements being checked in and checked out. During checkout, an XML file is taken from the change control system and imported into the current project. During checkin, an XML file is exported from the current project and put into the change control system. The XML files use a specialized tag set that can express the range of elements and relationships that exist in an Object Builder model. The tag set is defined in the DTD (document type definition) `eom.dtd`.

This is the format used in the unit of change control. Files in this format can be imported into and exported from Object Builder. Examples:  
`udbo.Claim.xml`, `uddo.ClaimDO.xml`

The files you can export in this format are described in “XML interchange files” on page 493.

- **The checkin, checkout, and extract template format**

Defines the fields required to assemble the appropriate checkin or checkout command. The default checkin template, for example, defines fields for filename (set by Object Builder’s XML export process), and for defect, release, component, family, and relative path (all set by the user).

You can define your own template format or base one on the provided samples. The samples are described in more detail in “Change control sample” on page 473

- **The checkin and checkout script or macro format**

Defines defaults for the fields, and organizes the fields into a wizard for the user

This is the format created by the SmartGuide Customizer for XML. You can use files in this format to run XML wizards, using the `xml launch` command. Typically these files have the same name as the input file, with “-macro” added to the filename. The XML wizards produce a filled-in template and feeds it back to Object Builder. Object Builder uses the filled-in template to assemble a command-line command that checks in or checks out the appropriate file. Examples: `checkin-macro.xml`, `checkout-macro.xml`

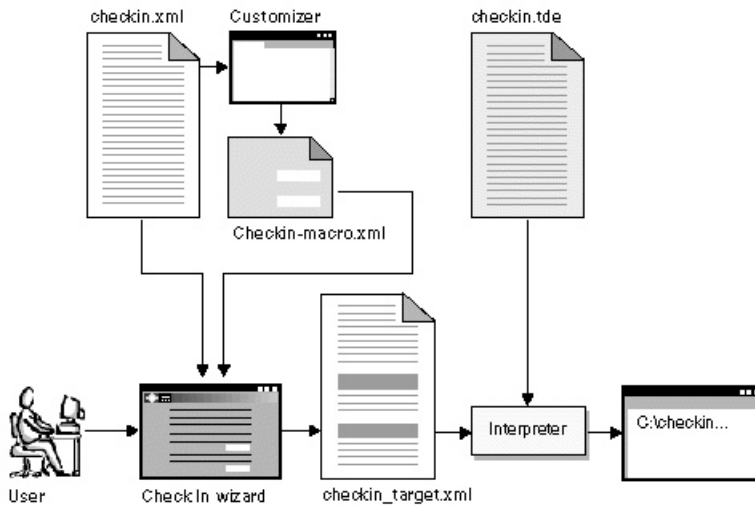
- **The template interpreter format**

Defines the way in which the filled-in template (generated by the XML wizard) will be used to construct a command line. This format is described in “Template interpreter format” on page 478.

### **How the command line for checking in a file is assembled**

In Object Builder, you select an element, display its pop-up menu, and click **Check In**. A wizard opens, whose format and content are determined by the files `checkin.xml` plus `checkin-macro.xml`. You provide information in the wizard, which it uses to create a filled-in template (based on `checkin.xml`) that is fed to a template interpreter. The filled-in template is interpreted based on rules specified in `checkin.tde`, and the assembled instruction runs on the command line.

## Check-in Command



### RELATED CONCEPTS

“Change control” on page 463

“Chapter 10. Team development” on page 443

### RELATED TASKS

“Setting up a change control process” on page 464

“Setting up XML-based change control for CMVC”

“Customizing XML-based change control” on page 475

### RELATED REFERENCES

“XML interchange files” on page 493

“Change control sample” on page 473

“Template interpreter format” on page 478

## Setting up XML-based change control for CMVC

You can check work in and out of your change control system using XML files as your unit of change control. This capability is built into Object Builder: you can check in or check out files from the Tasks and Objects pane, on a per-folder basis or on a per-file basis.

This is *not* the recommended unit of change control for this version of Object Builder. The XML does not contain information such as project dependencies, or Rational Rose mappings. If you have a team environment with many projects, or use additional tools with Object Builder, you should use the project directory, with its subdirectories, as the unit of change control.

If you have an uncomplicated team environment with relatively static project divisions and no dependencies on other tools, you can use the XML check-in and check-out feature provided with Object Builder.

Before you can begin using the feature, you must populate CMVC with your files, and customize the wizards in Object Builder.

To populate CMVC with your XML files, export all files for all projects. Store them in CMVC. Generally, you will need to include the project directory as part of each file's path (for example, MyProject\Export\udbo.AgentFile.xml). This ensures that the record of the file will be unique within the system, even if the file name is used by other projects or applications stored in the system.

The check-in, check-out, and extract wizards are actually XML wizards, and you can edit them using the SmartGuide Customizer for XML. By default, the wizards work with CMVC, and provide input fields for defect number, release, component, family, and path to the file (generally the project's \Export directory, for example e:\myproject\Export). You can also customize the wizards to work with any system that has a command-line interface, or to extend the range of information the user provides (for example, to include feature numbers).

To set up the existing check-in, check-out, and extract wizards for use with your CMVC system, follow these steps:

1. Start the SmartGuide Customizer for XML, by typing the command `xmlcustm` on the command line.
2. Click **File > Open**.
3. Locate and select the file `checkin-macro.xml`, located in your `<CBroker>\bin` directory.
4. Click **OK**. The file is opened in the Customizer.
5. In the Contents pane, expand each element in turn, and review its properties and the properties of its attributes. There are separate elements that define the defect number, release, family, and relative path for the check-in command.

Note that the DEFECT element has been specified as repeatable, and instructions for specifying multiple defects are provided as introduction text for the wizard page. A repeatable element gets its own page in the XML wizard, where a tree view allows the user to specify and organize multiple instances of the element (in this case, multiple defect numbers to be used when checking in a file).

6. Expand the FAMILY element, and click on its name node (the value of the FAMILY element).

7. Change the default value for name from “family” to the name of your CMVC family. This saves you the trouble of providing the family name every time you check in a file.
8. Click **File > Save**.
9. Customize the checkout-macro.xml file and extract-macro.xml file in the same manner.

You can go beyond this simple customization, to include constraints on valid values, organize the way the information is laid out in the wizard, and provide flyover help or HTML help for the wizard fields and pages.

Once XML-based change control is set up, you can open a project in Object Builder, check out a file or series of files (that are already defined in the project, but become locked in the change control system once they are checked out), and work normally within the project and its team environment until you are ready to check in your changes.

When you access the change control wizard from within Object Builder (by checking in or checking out a file), the wizard takes your input, creates a customized XML document, and feeds the document to the template interpreter, which resolves the macros and assembles a command line instruction that completes the check-in, check-out, or extract action.

With the contents of each project under change control in the form of XML files, the project models do not need to be stored as well. However, they do need to be regularly refreshed to reflect any changes made in the change control system. This can be accomplished using a batch process, which extracts all XML files from the change control system, imports them into their respective projects, and then publishes the projects (makes them available for use by the team).

#### **RELATED CONCEPTS**

“XML-based change control” on page 469

“Change control” on page 463

#### **RELATED TASKS**

“Customizing XML-based change control” on page 475

“Exporting XML” on page 387

“Exporting XML from the command line” on page 660

#### **RELATED REFERENCES**

“XML interchange files” on page 493

“Change control sample”

## **Change control sample**

The change control sample consists of the following files:

- checkout.xml, checkout.dtd, checkout-macro.xml, checkout.tde  
Define a process for checking out a file from CMVC.
- checkin.xml, checkin.dtd, checkin-macro.xml, checkin.tde  
Define a process for checking in a file to CMVC
- extract.xml, extract.dtd, extract-macro.xml, extract.tde  
Define a process for extracting a file from CMVC

Each set of files supports an equivalent process in Object Builder. For example, for the checkout process, the user selects an element within a project (such as a business object file) and selects **Checkout** from its pop-up menu. A wizard appears, prompts the user for information, and uses the information to construct and run a command-line instruction that performs the actual checkout process (by locating the equivalent XML file in the change control system, checking it out, and importing it into Object Builder).

The files have the following purpose:

- checkout.xml, checkin.xml, extract.xml  
Define the elements that are required to construct the command line (for example, defect number, release, family). These act as templates, that define the format by example.
- checkout.dtd, checkin.dtd, extract.dtd  
Formally define the formats using the document type definition language specified in the XML standard.
- checkout-macro.xml, checkin-macro.xml, extract-macro.xml  
Define the way in which the user fills in the templates. These files are in the format generated by the SmartGuide Customizer for XML, and identify which information in the template needs to be provided by the user. They act as scripts for XML wizards, which run when the user clicks **Checkout**, **Checkin**, or **Extract**. The XML wizards produce filled-in templates, which are passed to an interpreter to be used in assembling the equivalent command line instructions.
- checkout.tde, checkin.tde, extract.tde  
Define the way in which the filled-in templates are interpreted, and the user-provided information is assembled into command-line instructions.

There is also a simpler version of the sample that does not use a DTD. It consists of the following simplified files:

- checkout2.xml, checkin2.xml, extract2.xml  
Define the elements that are required to construct the command line. Rename the files to checkout.xml, checkin.xml, and extract.xml in order to use them with Object Builder.
- checkout2-macro.xml, checkin2-macro.xml, extract2-macro.xml  
Define the way in which the user fills in the templates. Rename the files to checkout-macro.xml, checkin-macro.xml, and extract-macro.xml in order to use them with Object Builder.



The simplified sample uses the same .tde files as the original sample, and does not use DTDs.

#### **RELATED CONCEPTS**

“XML-based change control” on page 469

“Change control” on page 463

#### **RELATED TASKS**

“Setting up XML-based change control for CMVC” on page 471

“Customizing XML-based change control”

“Template interpreter format” on page 478

## **Customizing XML-based change control**

You can check work in and out of your change control system using XML files as your unit of change control. This capability is built into Object Builder: you can check in or check out files from the Tasks and Objects pane, on a per-folder basis or on a per-file basis.

This is *not* the recommended unit of change control for this version of Object Builder. The XML does not contain information such as project dependencies, or Rational Rose mappings. If you have a team environment with many projects, or use additional tools with Object Builder, you should use the project directory, with its subdirectories, as the unit of change control.

If you have an uncomplicated team environment with relatively static project divisions and no dependencies on other tools, you can use the XML check-in and check-out feature provided with Object Builder. Before you can begin using the feature, however, you must customize it to work with your change control system.

If your change control system is CMVC, you can use the samples provided with Object Builder as a starting point, as described in “Setting up XML-based change control for CMVC” on page 471.

If you are using a change control system other than CMVC, or want to extend the sample CMVC support, follow these steps:

1. Create backup copies of the main change control XML sample: checkin.xml, checkin-macro.xml, checkout.xml, checkout-macro.xml, extract.xml, extract-macro.xml
2. Locate the sample files in the <CBroker>\samples\teamlib directory (checkin2.xml, checkout2.xml, and so on). These form a simpler version of the change control sample, and will be easier to start with if you are unfamiliar with XML.
3. Create copies of them, renamed to checkin.xml, checkout.xml, and so on.

4. Move the copies over to the original directory (<CBroker\bin).
5. Modify the files checkin.xml, checkout.xml, and extract.xml. They need to define the fields for which you will need to gather data during the checkin and checkout processes.

When you modify the files, create a separate element tag for each type of information you need to gather (for example, <DEFECT> for defect numbers, <FEATURE> for feature numbers). This is necessary so that you can identify an element's content when you construct and execute the command-line command.

A simple prompting structure might look something like this:

```
<?xml version="1.0" standalone="yes"?>
<CHECKOUT>
 <USER>username</USER>
 <PASSWORD>password_here</PASSWORD>
 <COMMENTS>why you are checking it out</COMMENTS>
</CHECKOUT>
```

A more sophisticated XML document could include a DTD, and IDs and attributes, for a more robust macro implementation and DTD-enforced value constraints. The main sample (the original checkin.xml, checkout.xml, and so on) illustrates such an implementation. It includes a DTD and IDs. For the purpose of constructing a command-line instruction, however, a simple document, as shown in the secondary sample, should be sufficient.

6. Start the SmartGuide Customizer for XML, by entering the command `xmlcustm` on the command line.
7. Click **File > Open**, and open checkin.xml.
8. Locate the Text node for each element (the content you are going to be prompting the user for).
9. For each Text node, set its **Macro** value in the Contents pane to Editable.
10. For each Text node, apply any constraints you want enforced by the wizard interface. By default, no spaces are allowed in the value. In the example above, you would need to change the constraint for <COMMENTS> to **Any**, to allow the user to enter text that includes spaces.
11. If any of the elements are repeatable (such as <DEFECT>), click on the element and check the **Repeatable** option in the Contents pane. Also set the **Macro** value to Hidden.
12. Click **File > Save**. The file checkin-macro.xml is created. This is the file that runs (using `xmllaunch`) when you click **Checkin** in Object Builder (from a folder or file pop-up menu).
13. Do the same for checkout.xml and extract.xml.
14. Open the file checkout.tde in a plain text editor (for example, notepad). This file takes the content supplied by the user (a filled-in XML template

generated by the checkout XML wizard) and uses it to construct a checkout command line. Edit the file to construct the command line you require. For example, a template interpreter file for the sample structure above could look like this:

```
<?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "../dtd/tde.dtd">
<_TDEBlock_>
<sameline>
 File -checkout %1
 <hasscope NAME="USER">
 -user $USER//TEXT$
 </hasscope>
 <hasscope NAME="PASSWORD">
 -passwd $PASSWORD//TEXT$
 </hasscope>
 <hasscope NAME="COMMENTS">
 -comments "$COMMENTS//TEXT$"
 </hasscope>
</sameline>
</_TDEBlock_>
```

You can create multiple-line commands using the <sameline> grouping. Each set of instructions grouped within a <sameline> element is assembled into a single line. If you include multiple <sameline> elements, the assembled commands will be run in sequence.

Use %1 or \$1 to identify where in the command line the name of the file should go. The file name is defined by Object Builder, based on the element being checked out.

Within the <sameline> element, you can have multiple elements that pull in variable information from the XML file the XML wizard produces. For example, the user name, password, and any comments provided by the user can now be pulled in to the assembled command line. The information to be used is identified by source element name.

For example, if the user specifies "abc123" as a password in the checkout wizard, then the element:

```
<hasscope NAME="PASSWORD">
 -passwd $PASSWORD//TEXT$
</hasscope>
```

queries the XML file created by the checkout wizard, gets the content of the file's <PASSWORD> element, and assembles it into the command-line fragment "-passwd abc123".

You can use <repeatscope> elements for repeatable sections of the command line (as identified when you defined the XML wizard). Use

<hasscope> elements for optional non-repeatable elements (such as comments). You can also use <Scope> elements if a non-repeatable element is required.

15. Save and exit the file.
16. Customize checkin.tde and extract.tde in a similar manner, to query information from checkin.xml and extract.xml and assemble the information into the appropriate commands for your change control system.

#### **RELATED CONCEPTS**

"XML-based change control" on page 469

"Change control" on page 463

#### **RELATED TASKS**

"Setting up XML-based change control for CMVC" on page 471

"Exporting XML" on page 387

"Exporting XML from the command line" on page 660

#### **RELATED REFERENCES**

"XML interchange files" on page 493

"Change control sample" on page 473

## **Template interpreter format**

The template interpreter format takes information provided in a checkout, checkin, or extract wizard and turns it into a command line instruction or instructions. The format is organized as follows:

- Header
- Body
- Lines
- %1 or \$1
- Scopes (repeatscope, hasscope, Scope)

### **Header**

```
<?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "../dtd/tde.dtd">
```

Identifies the format as XML, names the document type as \_TDEBlock\_, and provides the location of the document type definition (tde.dtd).

### **Body**

```
<_TDEBlock_>
...
</_TDEBlock_>
```

Identifies the start and end of the XML information.

## Lines

```
<sameline>
...
</sameline>
```

Identifies information that should be assembled into a command line instruction. The body can contain multiple sameline elements, which will construct multiple command lines that will be run in sequence.

The <sameline> element contains a mix of plain text information (command-line fragments that will always be executed) and scoped information (command-line fragments that will vary based on user-provided information).

## %1

To pull in the name of the file being checked in, checked out, or extracted, use the special flag %1. Wherever %1 appears within a <sameline> element, the name of the file will be substituted. The file name is defined by Object Builder, based on the element selected within the Tasks and Objects pane.

## Scopes

```
<repeatscope>...</repeatscope>
<hasscope>...</hasscope>
<Scope>...</Scope>
```

Identifies information that is pulled from the wizard-produced XML file (that is, information provided by the user).

## <repeatscope>

```
<repeatscope NAME="SOURCE_ELEMENT">some text $SOURCE_ELEMENT$
some text</repeatscope>
```

Use for elements that have been specified as repeatable in the XML wizard (using the SmartGuide Customizer for XML). Allows for 1-n elements with this name, turned into a corresponding number of command-line fragments.

Example:

```
<repeatscope NAME="DEFECT">-defect=$DEFECT$</repeatscope>
```

## <hasscope>

```
<hasscope NAME="SOURCE_ELEMENT">some text $SOURCE_ELEMENT$
some text</hasscope>
```

Use for elements that are required. Allows for one element with this name, turned into a single command-line fragment.

Example:

```
<hasscope NAME="FAMILY">-family=$FAMILY$</hasscope>
```

**<Scope>**

```
<Scope NAME="SOURCE_ELEMENT">some text $SOURCE_ELEMENT$ some text</Scope>
```

Use for optional elements. Allows for 0 or 1 element with this name, turned into a single command-line fragment if present.

Example:

```
<Scope NAME="COMMENTS">-comments "$COMMENTS$"<Scope>
```

#### **RELATED CONCEPTS**

"XML-based change control" on page 469

"Chapter 10. Team development" on page 443

#### **RELATED TASKS**

"Setting up a change control process" on page 464

"Setting up XML-based change control for CMVC" on page 471

"Customizing XML-based change control" on page 475

#### **RELATED REFERENCES**

"XML interchange files" on page 493

"Change control sample" on page 473

---

## **Working in a team environment**

The same rules that apply to developing an application within a single project apply to developing an application across multiple projects. You must define parent components before child components, and referenced interfaces before referencing interfaces.

There are some considerations that are specific to development in a team environment, including how you check information in and out of a change control system, and how you work with references across projects. They are described in the following tasks:

1. "Checking out files" on page 482
2. "Checking in files" on page 483
3. "Extracting files" on page 484
4. "Creating a project in a team environment" on page 481
5. "Editing a project in a team environment" on page 485
6. "Deleting a project in a team environment" on page 486

7. "Building DLLs in a team environment" on page 487
8. "Packaging an application in a team environment" on page 489
9. "Testing cross-project applications with QuickTest" on page 489

#### **RELATED CONCEPTS**

"Chapter 10. Team development" on page 443

#### **RELATED TASKS**

"Developing as part of a team" on page 444

"Working with Rose in a team environment" on page 444

"Maintaining a team environment " on page 490

## **Creating a project in a team environment**

When you create a project in a team environment, make sure it contains an appropriate amount of the overall application. Avoid overlapping ownership of projects: try to have only one owner or developer per project. Minimize cross-project dependencies: try to keep related components in the same project.

To create a project in a team environment, follow these steps:

1. Create a project directory for the project, in your local project directory structure (for example, f:\currentprojects\mynewproject).
2. Identify any existing interfaces that will be required by the new project's component (for example, parent interfaces, or interfaces that will be used as attribute types or in method signatures).
3. List the projects that contain those interfaces as dependencies in the Open Project wizard, Project Dependencies page (for example, f:\allprojects\projectA, f:\allprojects\projectB).
4. Open the project, and begin your work on the new application elements it will contain. Add components in the same way you would in a stand-alone environment (starting with the business object interface file, an imported DB or PA schema, or data object interface file).
5. Save the project, and check your work (either the project model, or its associated XML files) into your change control system.
6. Check out the integration project (either the project model, or its associated dll.xml file) and add any build configuration nodes that apply (for example, client and server DLL configurations for any components you added to the new project).
7. Save your changes to the integration project, and check your work back into the change control system.
8. Update the automated build process to include the new project.

#### RELATED CONCEPTS

- “Chapter 10. Team development” on page 443
- “Project divisions in a team environment” on page 458

#### RELATED TASKS

- “Working in a team environment” on page 480
- “Editing a project in a team environment” on page 485
- “Setting up a change control process” on page 464
- “Setting up XML-based change control for CMVC” on page 471
- “Setting up an automated build process” on page 465

#### RELATED REFERENCES

- “Naming objects” on page 128
- “Internationalization of data” on page 132

## Checking out files



If you are using XML-based change control, you can check XML files in and out of your change control system directly from Object Builder. You must first have the contents of the project stored as individual XML files in your change control system, and have Object Builder’s check-in and check-out wizards set up to work with your change control system’s command-line interface.

To check out a file from your change control system into Object Builder:

1. Start Object Builder.
2. Make sure that the check box **Enable team library check-in and check-out** is selected in the Environment page for the Tasks and Objects folder. (Select **File > Preferences > Tasks and Objects**.) If the check box is not selected, you will need to reload the project or restart Object Builder to make the option effective.
3. Open the project that contains your work.
4. Locate the element that you want to check out in the Tasks and Objects pane.

#### Note the following points:

- The element that you want to check out must already exist as an XML file in your change control system.
- It should be read-only in the model. This state is indicated by a different icon from the one it is usually represented by in the model. The read-only icon looks like the read-write icon enclosed in a box.
- Even if the object is read-write, you can manually change its state from the command-line:

```
attrib +R filename
```

Once you have done this, the object’s icon changes, and the **Check Out** option appears on its pop-up menu.



5. From the pop-up menu of the element (for example, a business object file), click **Check Out** to open the XML Check Out wizard.
6. Fill in the information requested by the wizard (such as user name, defect number, and so on). The exact information required will depend on your change control system, and the way you set up the check-out wizard.
7. Click **Finish**.

The XML for the selected element is taken from your change control system and imported into the current Object Builder project. The definition of the element in Object Builder is updated, its icon and the icons of all its children change (indicating the state is read-write), and its definition in your change control system is locked, to prevent others from accessing it while you are working with it.

You can now make your changes to the element (using its Properties wizard), and check the element back into the change control system when you are done (the **Check In** pop-up menu option is enabled, and it does not have the **Check Out** option at this point).

#### **RELATED CONCEPTS**

“Chapter 10. Team development” on page 443

“XML-based change control” on page 469

#### **RELATED TASKS**

“Working in a team environment” on page 480

“Setting up a team environment” on page 457

“Checking in files”

## **Checking in files**



If you are using XML-based change control, you can check XML files in and out of your change control system directly from Object Builder. You must first have the contents of the project stored as individual XML files in your change control system, and have Object Builder’s check-in and check-out wizards set up to work with your change control system’s command-line interface.

To check in a file from Object Builder into your change control system:

1. Start Object Builder.
2. Ensure the **Enable team library check-in and check-out** option is enabled in the Preferences dialog. If the check box is not selected, you will need to reload the project or restart Object Builder to make the option effective.
3. Open the project that contains your work.
4. Locate the element you want to check in, in the Tasks and Objects pane.

**Note the following points:**

- The element that you want to check in must already exist as an XML file in your change control system, and you must have it currently checked out.
  - It should be read-write in the model. This state is indicated by the same icon the object is usually represented by in the model.
  - Even if the object is read-only, you can manually change its state from the command-line:  
`attrib -R filename`  
 Once you have done this, the object's icon changes, and the **Check In** option appears on its pop-up menu.
5. From the pop-up menu of the element (for example, a business object file), click **Check In** to open the XML Check In wizard.
  6. Fill in the information requested by the wizard (such as user name, defect number, and so on). The exact information required will depend on your change control system, and the way you set up the check-in wizard.
  7. Click **Finish**.

The XML for the selected element is exported from Object Builder (into the Export directory), and then put into your change control system. The XML file in the change control system is now updated with your changes, and your lock on the file is released, to allow others to check out the file and make changes to it.

The object and its children are now rendered read-only in the model. (They are represented by the different icons in the Tasks and Objects pane.)

#### **RELATED CONCEPTS**

"Chapter 10. Team development" on page 443

"XML-based change control" on page 469

#### **RELATED TASKS**

"Working in a team environment" on page 480

"Setting up a team environment" on page 457

"Checking out files" on page 482

## **Extracting files**

If you are using XML-based change control, you can extract XML files from your change control system directly into Object Builder. You must first have the contents of the project stored as individual XML files in your change control system, and have Object Builder's extract wizard set up to work with your change control system's command-line interface. The extraction is meant to refresh the version of the element in your project, without actually checking it out, which would prevent other users from making any changes to it.

To extract a file from your change control system into Object Builder:

1. Start Object Builder.
2. Open the project that contains your work.
3. Locate the element you want to extract (refresh), in the Tasks and Objects pane.  
The element you want to extract must already exist as an XML file in your change control system.
4. From the pop-up menu of the element (for example, a business object file), click **Extract** to open the XML Extract wizard.
5. Fill in the information requested by the wizard (such as family, release, and so on). The exact information required will depend on your change control system, and the way you set up the extract wizard.
6. Click **Finish**.

The XML for the selected element is extracted from the change control system and imported into Object Builder. Any changes that had been made to the file since you last extracted or checked it in are now reflected in your project.

#### **RELATED CONCEPTS**

#### **RELATED TASKS**

### **Editing a project in a team environment**

To edit a project in a team environment, follow these steps:

1. Ensure your local copy of the project repository (created by the automated build process) is up-to-date.
2. If you are using project or model-based change control, check out the project you want to edit from the change control system.
3. Open the project. Its dependencies on other projects should resolve using the project repository (based on the paths specified in the Open Project wizard's Project Search Path, or the paths specified in the OBModelPath environment variable).
4. If you are using XML-based change control, check out the elements you want to edit from your change control system.
5. Make your changes to the project. If your changes affect the way the project relates to other projects (for example, you want to add a reference that requires another project as a dependency, or you delete a reference that justifies an existing dependency), you will need to close and open the project again, to update the listed project dependencies (for example, add the dependency first, then add the new reference; or delete the reference, then remove the dependency).
6. Generate updated code for the project.

7. Build your code, using either a local definition of the DLLs you want to build, or using the integration project's DLL configurations.
8. If you are using XML-based change control, check in the elements you edited.
9. If you are using project or model-based change control, close Object Builder and check in the project or model.

Some of your changes may affect other projects, and generated code. For instance, if you change the type of an attribute, or the signature of a method in a business object interface in model A, and you override these artifacts in a child business object implementation (for example) in model B, Object Builder warns you that the child model is not writable. Follow these steps:

1. Make model B available for write mode.
2. Click **Continue** to have Object Builder open the model and propagate the change correctly, or click **Cancel** to return to the wizard, where you can cancel the change.

If you rename an interface, any methods, attributes, constructs, or relationships that reference the interface have their type renamed automatically. If a referenced interface is deleted, the reference type becomes `invalidType`. For example, if Customer has an attribute `custAgent` of type `Agent`, and the `Agent` interface is deleted, Customer now has an attribute `custAgent` of type `invalidType`. You can locate all occurrences of `invalidType` within a project by running the Model Consistency Checker.

#### **RELATED CONCEPTS**

"Chapter 10. Team development" on page 443

#### **RELATED TASKS**

"Working in a team environment" on page 480

"Checking out files" on page 482

"Checking in files" on page 483

"Building DLLs in a team environment" on page 487

"Checking a model for consistency" on page 31

## **Deleting a project in a team environment**

To delete a project in a team environment, follow these steps:

1. If you are using project or model-based change control, delete the project from your change control system.
2. If you are using XML-based change control, delete the project's XML files from your change control system.
3. Update the automated build process, to remove any reference to the project.

4. Check out the integration project (either the project model, or its associated `dll.xml` file), and remove any build configuration nodes for code in the project.
5. If you are using project- or model-based change control, check out any projects that have dependencies on the deleted project, and remove their dependencies on the Open Project wizard's Project Dependencies.
6. Check in the checked-out projects or XML files.

Any references within a project that depend on the deleted interface will be automatically modified to point to type `invalidType` the next time the project is opened. The following properties will be automatically modified:

- Attributes are modified when the interface whose type they are is deleted.
- Methods are modified when an interface used as their return type, or as a parameter, is deleted.
- Object relationships are modified when the interface they refer to is deleted.

You can find `invalidType` references within a project by checking the project model's consistency.

#### **RELATED CONCEPTS**

"Chapter 10. Team development" on page 443

#### **RELATED TASKS**

"Working in a team environment" on page 480

"Checking out files" on page 482

"Checking in files" on page 483

"Checking a model for consistency" on page 31

## **Building DLLs in a team environment**

When your project is part of a team environment, typically the entire application will share a single integration project, that defines the build configuration options for all the components in the team environment, regardless of the project they are defined in.

When you have edited a project and want to rebuild the code that was affected by your changes, you can use the integration project in the project repository to rebuild the affected DLLs. This will only affect your local copies of the DLLs; once you check an edited project back into your change control system, the DLLs will be rebuilt by your automated build process, and the updates will be made available in the next version of the project repository.

**Note:** It is advisable to run the Consistency Checker before you generate makefiles, or start building, to ensure that the configuration for your client DLL is correct. This is crucial particularly if your client DLL does not have at least one IDL file selected.

To build a DLL locally in a team environment, after you have made changes to a checked-out project, follow these steps:

1. Regenerate the makefiles in your integration project (in your local copy of the project repository). This creates a version of the makefiles that correctly points to the updated code in your local check-out directory.
2. Build the updated makefiles, using the integration project's all.mak file.

If your project repository is editable, then you can regenerate the makefiles and build the code from within Object Builder:

1. Open the integration project.
2. From the Build Configuration folder's pop-up menu click **Generate > All Targets**
3. From the same pop-up menu click **Build > Out-of-Date Targets > Default**.

If your project repository is read-only, then you can regenerate the makefiles and build the code from the command line. For example:

1. `obgen -pF:\allprojects\Integration -aMake -linked -tNT`
2. `nmake -f F:\allprojects\Integration\Working\NT\all.mak`

**Note:** If your compile command fails due to an incorrect DB2 user ID and password error, run the following command before you run the make (AIX) or nmake (NT) command:

```
export IVB_DB2AUTH="USER test USING password"
set IVB_DB2AUTH=USER test USING password
```

You can also create a build configuration definition within the checked-out project, without using the integration project. To do so, simply add client and server DLL definitions in the usual manner. You can use the same DLL file names as those in the integration project, but should define different DLL configuration node names.

#### **RELATED CONCEPTS**

"Chapter 10. Team development" on page 443

#### **RELATED TASKS**

"Working in a team environment" on page 480

"Setting up an automated build process" on page 465

"Adding an integration project to a team environment" on page 460

"Generating code from the command line" on page 684

"Configuring builds" on page 549

"Packaging an application in a team environment" on page 489

## Packaging an application in a team environment

You can use the integration project of your team environment to do your application packaging. Add application families, applications, and managed object configurations in the usual manner. Before you create the install image, you will need to build the DLLs for the entire application, so that the built files in the project's \Working directories are up-to-date.

If you are packaging your application in a different project than your integration project, you will still need to build from the integration project first, and then copy the contents of the integration project's \Working directory (including subdirectories) to the application packaging project's \Working directory.

You can then configure the application, and create the install image, in the same way you would in a standalone project environment.

### RELATED CONCEPTS

"Chapter 10. Team development" on page 443

### RELATED TASKS


"Working in a team environment" on page 480

"Packaging applications" on page 574

## Testing cross-project applications with QuickTest

The source of the QuickTest-generated Java code is in the <Working directory>\QT directory. When these files are compiled, the resultant JAR file is located under the target environment directory. For example, if the production directory is the target, then these files are located in <Working directory>\PRODUCTION\QT. After the QuickTest Java files are compiled, the tests may be executed by selecting **Build Configuration > Build > Run QuickTest**.

To run a QuickTest client for a server application in a team environment (with components in multiple projects working together, for example, various JCB files residing on different machines), follow these steps:




1.  Run the qtgen.bat file

 Run the qtgen.ksh file

**Note:** Both these files are located in the <CBroker>\bin directory. This will start QuickTest with a menu option **File > Generate**. The QuickScript Generator window opens.

2. Use this window to select all of the JCB and QT JAR files for every project that is part of the team development environment.

3. When you have selected the appropriate JAR files, select **File > Save**. All the selected JAR files are copied to the selected directory location, and a qt.bat file (on Windows NT), or qt.ksh file (on AIX) is created that will invoke QuickTest to support the test cases.

   **Note:** If you have debug and trace enabled, you must make sure that the JAVA\_HOME environment variable is set to the directory in which your JDK is installed. You must do this before you run Object Builder (if you will launch QuickTest from Object Builder), or before you start QuickTest (that is, before you can invoke the qt.bat, or the qt.ksh file from the command line).

#### RELATED CONCEPTS

“Chapter 10. Team development” on page 443  
“QuickTest” on page 611

#### RELATED TASKS

“Working in a team environment” on page 480  
“Building DLLs in a team environment” on page 487  
“Chapter 13. Testing applications with QuickTest” on page 611

#### RELATED REFERENCES

“Platform-specific information” on page 20

---

## Maintaining a team environment

After your team environment is defined (either through migration or evolution), maintenance of the environment must provide for the relocation of projects, the relocation of objects between projects, and the management of cross-project dependencies.

The main strategies for managing multiple projects are:

- Make changes in logical units (move parent and child components together when possible).
- Make changes in logical order (change parent components before child components, change referenced components before referencing components).
- Edit project divisions by selectively exporting the project’s contents as XML files, deleting the content from the project, then importing the XML files into another project or projects.
- When you have two versions of a project, resolve the differences by exporting and merging their XML with the Compare and Merge Tool for XML.

The specific tasks for maintaining a team environment are as follows:



1. “Exporting XML” on page 387
2. “Importing XML” on page 389
3. “Moving a project”
4. “Changing project divisions” on page 496
5. “The Compare and Merge Tool for XML” on page 497
6. “Comparing files with the Compare and Merge Tool for XML” on page 497
7. “Merging files with the Compare and Merge Tool for XML” on page 498
8. “Managing cross-project dependencies” on page 499
9. “Documenting projects” on page 501

#### **RELATED CONCEPTS**

“Chapter 10. Team development” on page 443

#### **RELATED TASKS**

“Developing as part of a team” on page 444

## **Moving a project**

To change the location of a project, follow these steps:

1. Move the project directory, and its subdirectories, to its new location.
2. Update your project search path to point to the new directory, so that any other projects that depend on the moved one will still be able to find it.

To set the project search path:

1. Start Object Builder. The Open Project wizard opens.
2. In the Project Search Path field, type the directories Object Builder should search for related projects. You can create a new project search path, or edit a previously defined one.
3. Select any project to open.
4. Click **Finish**.
5. Close Object Builder. The new project search path is saved, and will be available for selection whenever you use Object Builder on the current machine.

If you prefer, you can also use the `OBMODELPATH` environment variable, instead of defining search paths through Object Builder. To set the `OBMODELPATH` environment variable, use the following command:

```
set OBMODELPATH=[directory1;directory2;...directoryn]
```

For example:

```
set OBMODELPATH=f:\project1;g:\project2
```

#### RELATED CONCEPTS

“Chapter 10. Team development” on page 443

#### RELATED TASKS

“Maintaining a team environment ” on page 490

## Model interchange with XML

You can exchange model information between projects using an exported XML format. This format should **not** be edited directly.

You cannot exchange XML files between a 2.0 version project and a 3.0 version project. The 2.0 project must be migrated to 3.0, and the XML files re-exported.

Version 3.5 supports XML files of version 3.0 format, and such files can be imported directly into Object Builder version 3.5. No migration is needed.

When you export XML, it is placed in the exporting project’s \Export directory.

You can export information at the project, folder, or object level. The exported XML conforms to a DTD (document type definition) for Component Broker models: eom.dtd. Only XML that conforms to the DTD can be imported.

The files you can export are documented in “XML interchange files” on page 493. You can export XML for each level as follows:

### Project

You can export the entire project model by selecting **File > Export Model** in Object Builder. The XML files that define the project’s contents are generated to the \Export directory. This is equivalent to selecting **Export** from the pop-up menu of each folder in the Tasks and Objects pane, with the addition of the file uddep.xml, which defines project dependencies.

The file uddep.xml is *only* created when you export at the project level.

### Folder

You can export the contents of a particular folder by selecting **Export** from the folder’s pop-up menu. The XML files representing the folder’s contents are generated to the \Export directory. For folders that have files as their top-level element, this is equivalent to selecting **Export** from the pop-up menu of each file in the folder.

You can generate XML files for all the folders in the Tasks and Objects pane except for Framework Interfaces and Default Homes.

### Component or object

You can export XML for a single file or top-level object within a folder by selecting **Export** from the file or object's pop-up menu. The XML file representing the file's associated objects is generated to the \Export directory. You can generate files for individual objects in all folders in the Tasks and Objects pane except for Framework Interfaces and Default Homes.

#### RELATED CONCEPTS

"Chapter 10. Team development" on page 443

#### RELATED TASKS

"Exporting XML" on page 387

"Exporting XML from the command line" on page 660

"Importing XML" on page 389

"Importing XML from the command line" on page 663

#### RELATED REFERENCES

"XML interchange files"

"obexport" on page 661

"obimport" on page 664

## XML interchange files

You can export XML files from Object Builder at the project level, at the folder level within a project, or at the file or top-level object level within a folder.

Folder	Exported files	Contents
Whole project	uddep.xml plus all others  <b>Example:</b> uddep.xml depends-on ProjectA depended-on-by ProjectB	Project dependencies

Folder	Exported files	Contents
<b>Local-Only Objects</b>	udlocal.lofilename.xml  <b>Example:</b> <b>udlocal.LFile.xml</b> LOModule::SetRWAgentLO LOModule::GetAllAgentLO  <b>udlocal.LFile.xml</b> LOModule::SetRWAgentLO LOModule::GetAllAgentLO	A local-only object
<b>User-Defined Business Objects</b>	udbo.bofilename.xml  <b>Example:</b> <b>udbo.AFile.xml</b> AModule::Agent AModuleBO::AgentBO AModuleKey::AgentKey AModuleCopy::AgentCopy AModuleMO::AgentMO	<ul style="list-style-type: none"> <li>• Business object interfaces (one file, plus the modules and interfaces it contains)</li> <li>• Business object implementations</li> <li>• Keys and copy helpers for the business object interfaces</li> <li>• Managed objects for the business object implementations</li> </ul>
<b>User-Defined Compositions</b>	udcb.compfile.xml  <b>Example:</b> <b>udcb.ACFile.xml</b> ACModule::AccountComposition	Compositions (one composition file, plus the modules and compositions it defines)
<b>User-Defined Data Objects</b>	uddo.dofile.xml  <b>Example:</b> <b>uddo.AFileDO.xml</b> AModuleDO::AgentDO AModuleDOImpl::AgentDOImpl	Data objects (one data object file, plus the modules and interfaces it contains, along with their associated implementations)
<b>DBA-Defined Schemas</b>	uddbschema.schemagroup.xml  <b>Example:</b> <b>uddbschema.CBSampDBGGroup.xml</b> CBSampDBGGroup CBSampDB.Agent AgentPO	<ul style="list-style-type: none"> <li>• A schema group</li> <li>• Schemas in the schema group</li> <li>• Persistent objects for the schemas</li> </ul>

Folder	Exported files	Contents
<b>User-Defined PA Schemas</b>	udpaschema. <i>paschema.xml</i> <b>Example:</b> udpaschema.SampPA.xml SampPA AgentPO	A PA schema and its persistent objects
<b>Non-IDL Type Objects</b>	udnidl. <i>nidltype.xml</i> <b>Example:</b> udnidl.IDate.xml IDate	A non-IDL type
<b>Build Configuration</b>	uddll. <i>dllconfig.xml</i> <b>Example:</b> uddll.AgentS.xml AgentS.dll AgentS.jar	A build configuration node (DLL configuration)
<b>Application Configuration</b>	udaf. <i>appfamily.xml</i> <b>Example:</b> udaf.SampleAppFam.xml SampleAppFam SampleApp AgentMO CustomerMO	An application family and its contents
<b>Container Definition</b>	udcontainer. <i>container.xml</i> <b>Example:</b> udcontainer.ContainerOfAgents.xml ContainerOfAgents	A user-defined container
<b>EJB</b>	udejb. <i>ejbname.xml</i> <b>Example:</b> udejb.AgentBean.xml AgentBean	A user-defined EJB bean

#### RELATED CONCEPTS

“Chapter 10. Team development” on page 443

“Model interchange with XML” on page 492

#### RELATED TASKS

- “Exporting XML” on page 387
- “Exporting XML from the command line” on page 660
- “Importing XML” on page 389
- “Importing XML from the command line” on page 663

#### RELATED REFERENCES

- “obexport” on page 661
- “obimport” on page 664

## Changing project divisions

You can move information from one project to another by exporting the information in XML format, and then importing it into the other project.

When possible, follow these guidelines for transferring information. Otherwise, relationships or references may be automatically deleted during the transfer.

- Move information in logical units (move parent and child components together when possible)
- Move information in logical order (move parent components before child components, move referenced components before referencing components)

To transfer information from one project to another, follow these steps:

1. Select the element that you want to transfer. The elements you can export are described in “XML interchange files” on page 493.
2. From the element’s pop-up menu, click **Export**.  
An XML file corresponding to the element is exported into the project’s \Export subdirectory.
3. Delete the element from the project.
4. Save and close the project.
5. Open the target project.
6. Click **File > Import Model**.
7. Click **Find** and select the XML file you had exported.
8. Click **Finish**. The information is imported, and appears in the folder.
9. Save and close the project.

If you are using XML-based change control and CMVC, you will also need to rename the file in CMVC to reflect its new path.

#### RELATED CONCEPTS

- “Chapter 10. Team development” on page 443
- “Model interchange with XML” on page 492

**RELATED TASKS**

- “Maintaining a team environment ” on page 490
- “Managing cross-project dependencies” on page 499
- “Setting up XML-based change control for CMVC” on page 471
- “Exporting XML” on page 387
- “Importing XML” on page 389

**RELATED REFERENCES**

- “XML interchange files” on page 493

## The Compare and Merge Tool for XML

You can use the Compare and Merge Tool for XML to compare the XML files generated from project models based on node identification, and then merge them. You can decide which differences to include in the resultant, merged file.

You can use the tool in two specific scenarios: you can review changes that you made to a file over a course of time, and you can merge the changes if you want to, or you can use the tool to review and consolidate changes made to a single XML file by different users, who essentially work on the project in a team development environment.

**Note:** To eliminate any inconsistencies that might exist in the resulting model, you can use another tool called the Model Consistency Checker.

**RELATED CONCEPTS**

- “Chapter 10. Team development” on page 443
- “Model interchange with XML” on page 492

**RELATED TASKS**

- “Comparing files with the Compare and Merge Tool for XML”
- “Merging files with the Compare and Merge Tool for XML” on page 498
- “Comparing files with the Compare and Merge Tool for XML”
- “Merging files with the Compare and Merge Tool for XML” on page 498
- “Maintaining a team environment ” on page 490
- “Importing XML” on page 389
- “Exporting XML” on page 387

## Comparing files with the Compare and Merge Tool for XML

You can compare XML files at the element level (a level higher than the file level), based on node identification. Follow these steps:

1. Launch the Compare and Merge Tool for XML using the command `xmldiff` from a command line.

2. Use the menu to import the files to be compared into the tool: from the File Menu, choose Open. The Select Base XML File dialog box opens. Type the name of the base (control) file against which to base your comparison. The tool parses the file.
3. The Select Modified XML File dialog box opens. Type the name of the modified file that you want to compare with the base file. The tool parses the file, and displays a preliminary, combined view of the two files in the Merged View pane. Symbols and color highlight the differences between the two files.

#### RELATED CONCEPTS

“The Compare and Merge Tool for XML” on page 497

“Model interchange with XML” on page 492

“Chapter 10. Team development” on page 443

#### RELATED TASKS

“Merging files with the Compare and Merge Tool for XML”

## Merging files with the Compare and Merge Tool for XML

Once you have a display of the combined XML files in the Merged View pane, you can walk through the changed nodes and decide whether the change should be incorporated in the merged file from either the base file, or the phase file.

Every modified node in the tree has an associated pop-up menu, with choices that enable you to implement the decision whether to incorporate properties from either the base file, or the modified file.

The new nodes have the following pop-up menu choices:

- **Do not use new:** the new node is not incorporated in the merged file.
- **Use new element:** the merged file has the new node, and its children, if any, as they are in the modified file.

The deleted nodes have the following pop-up menu choices:

- **Do not delete:** the merged file has the node as it exists in the base file.
- **Delete from base file:** the merged file does not have the node that was deleted from the base file.

The changed nodes have the following pop-up menu choices:

- **Use old, where conflict:** the merged file has the nodes as they are in the base file for the current node, and any of its unresolved children (those modified child nodes for which you have not made a decision about incorporation in the merged file yet).
- **Use new, where conflict:** the merged file has the nodes as they are in the modified file for the current node, and any of its unresolved children.



These choices are also available from the Selected menu.

To merge the two XML files, follow these steps:

1. Select one of the highlighted nodes in the merged tree view.
2. Select **Use modified file for node and unresolved children** from the pop-up menu of the node if you want to have the merged file incorporate the changes that were made in the modified XML file. Select **Use base file for node and unresolved children** from the pop-up menu of the node if you want to have the merged file have the older version of the corresponding node.
3. Use **Edit > Undo** to undo all of your actions up to the last time you saved your work.

The menu selection you make on a node is applied to the node, as well as to all its unresolved child nodes.

For example, if you select a new node to be part of the merged file (**Use modified file for node and unresolved children**), then all its children will also be in the merged file; in addition, if this was the only node with a conflict under its parent, then the parent would be marked as resolved (its red cross-bar will disappear). Similarly, if you use the phase file as the source for change propagation for a deleted node, the node and all its children will not be present in the merged file.

Whenever the changes of all the child nodes are resolved, the parent node and all its child nodes will have a check mark in front of them.

#### **RELATED CONCEPTS**

“The Compare and Merge Tool for XML” on page 497

#### **RELATED TASKS**

“Comparing files with the Compare and Merge Tool for XML” on page 497

## **Managing cross-project dependencies**

Each project maintains its own list of dependencies. The list covers both the dependencies it has on other projects (displayed in its Open Project wizard, Project Dependencies page), and the dependencies other projects have on it (displayed in the Project Dependencies page of other projects).

When you create a dependency from one project on another, the dependency is added in the dependencies files for both projects.

When you open a project that has dependencies, the models of the projects depended on are opened in read-only mode. The dependency files for the projects are opened in read-write mode.

When you delete a dependency from a project, its listing is removed from the dependencies files for both projects (the dependent project and the depending project).

To avoid managing the dependency files when you move a project or change the directory structure, use the project search path, which you can define or select whenever you open a project. The directories you specify, and their subdirectories, will be searched to locate project dependencies.

If you prefer not to use the project search path, you can use the `OBMODELPATH` environment variable instead.

To set the `OBMODELPATH` environment variable, use the following command:

```
set OBMODELPATH=[directory1;directory2;...directoryn]
```

For example:

```
set OBMODELPATH=f:\project1;g:\project2
```

The directories you list, and their subdirectories, will be searched for project dependencies whenever you open an Object Builder project.

**Note:** The more directories Object Builder searches, the longer it will take to open projects. Try to achieve a compromise between completeness (searching all appropriate project directories) and speed (avoid listing the root directory of every drive).

If you are importing XML that contains cross-project references, you can use the `obimport` command with the `-X` option to import the XML for all the affected projects at once, while preserving the cross-project references.

For example:

```
obimport -X -d Import e:\myRBprojects\projA e:\myRBprojects\projB
e:\myRBprojects\projC
```

This command looks in the `Import` subdirectory of each listed project directory, and imports the XML files it finds there.

#### **RELATED CONCEPTS**

“Chapter 10. Team development” on page 443

**RELATED TASKS**

- “Maintaining a team environment ” on page 490
- “Importing XML” on page 389
- “Moving a project” on page 491

**RELATED REFERENCES**

- “XML interchange files” on page 493
- “obimport” on page 664

## Documenting projects

You can document projects, or parts of projects, by applying XSL style sheets to exported XML files. An XSL style sheet can filter and format the XML file into a browsable HTML document.

To get you started, there is an XSL style sheet sample. It does not cover all aspects of a project, however. You can extend or replace the sample with your own style sheet.

**RELATED CONCEPTS**

- “Chapter 10. Team development” on page 443
- “XML browsing with XSL” on page 538
- “The XSL sample” on page 541

**RELATED TASKS**

- “Maintaining a team environment ” on page 490
- “Browsing XML files” on page 539



---

## Chapter 11. Customizing Object Builder

You can extend the way you work with Object Builder by creating new wizards to work with project elements, by developing for multiple platforms, and by filtering the view of your projects in the Tasks and Objects pane.

- “Creating an XML wizard” on page 504
- “Setting platform constraints” on page 421
- “Tutorial: Developing a multi-platform application” on page 429
- “Filtering the Tasks and Objects pane” on page 537
- “Creating a filter for the Tasks and Objects pane” on page 537

### RELATED CONCEPTS

“Object Builder” on page 1

### RELATED TASKS

“Developing in Object Builder” on page 19

“Setting up Object Builder” on page 24

---

## XML wizards

When you create a complex XML document, one of the standard authoring strategies is to look at an example document first, and then re-use its structure and content, customizing only the parts that you need. In this way you start with a valid structure that roughly meets your needs, and then extend or change it only as necessary. This reduces the time you need to learn a DTD before working in it, and makes it both quicker and easier to create valid, useful XML documents.

This process can be made even simpler and more repeatable by creating a wizard as an interface to editing the example document. You can create an XML wizard, or SmartGuide, using the SmartGuide Customizer for XML. The wizard or SmartGuide allows you to selectively add and edit element types in the document.

Begin the process of creating an XML wizard by identifying or creating the sample XML file. You can then open the file in the SmartGuide Customizer for XML, and explicitly mark those sections you want to change or extend. When you are done, you can generate an XML wizard script. When you run the script, the wizard exposes the elements you chose to be editable, and applies your edits to create a new document based on the original. The new document is extended only in the ways you selected; the structure and context

of the original file is preserved. All your changes are applied through the wizard, without editing the source directly.

Once you have created the wizard, anyone can use it to create new documents following the pattern you set, without having to understand the XML DTD at all, or ever work in XML directly. You can also add help text and hover text to the wizard, and default values for its fields, to make it even easier to use.

#### **RELATED CONCEPTS**

“Model interchange with XML” on page 492

#### **RELATED TASKS**

“Creating an XML wizard”

“Exporting XML” on page 387

“Importing XML” on page 389

## **Creating an XML wizard**

You can use an XML wizard, or SmartGuide, to allow selective editing and extension of an XML file through a wizard interface that provides constraints, descriptions, hover help, and HTML help. The wizard script can include customized default values, derivation relationships between values, and customized lists of selectable values.

To create an XML wizard, you first need an example XML file, that you can use as a template for the output that the wizard will generate. Once you have the example XML file, open it in the SmartGuide Customizer for XML, and begin working with its elements.

To create an XML wizard, you can follow these steps:

1. “XML template design” on page 505
2. “Starting the SmartGuide Customizer for XML” on page 506
3. “Defining XML wizard macros” on page 507
4. “Customizing value lists in an XML wizard” on page 510
5. “Deriving values in an XML wizard” on page 511
6. “Propagating values in an XML wizard” on page 513
7. “Constraining values in an XML wizard” on page 515
8. “Defining the layout of an XML wizard” on page 516
9. “Testing an XML wizard” on page 517

Once you have created the XML wizard, you can run it to produce new XML files based on your original template.

You can work with existing XML wizards in the following ways:

- “Running an XML wizard” on page 518
- “Editing an XML wizard” on page 519
- “Distributing an XML wizard” on page 519

#### **RELATED CONCEPTS**

“XML wizards” on page 503

“Model interchange with XML” on page 492

#### **RELATED TASKS**

“Chapter 11. Customizing Object Builder” on page 503

“Exporting XML” on page 387

## **XML template design**

When you create an XML wizard template:

- Use a format that already has a DTD, if one is available, or define a DTD for the format.
- Use a format that has meaningful tag names.
- If you are going to have multiple instances of a particular element, make sure the element has an ID attribute that is identified as such in the DTD.

Although you can use a template that has no DTD, a DTD can provide considerable value both as a check on the validity of generated documents and as a source of default information for the SmartGuide Customizer which creates the wizard.

One of the more important things a DTD can provide is a definition of ID attributes. While you can define IDs in the Customizer, this is not the same as defining them in a DTD. If you define ID attributes in the DTD then the XML wizard script, which will apply a series of user changes to the template, can identify the target for the changes more accurately, based on element IDs.

If you do not use IDs, the target for the changes will be determined by position information (for example, second child of third element has changed). If the original template changes, this can render the positioning information inaccurate by changing the target element’s position count. If you use IDs, the target for the changes is determined by ID, and is unaffected by changes to the original template.

### **Editing templates**

When you edit an XML wizard template, XML wizard scripts that are based on the template can be rendered not valid. It may be safest to create new XML wizard scripts after you edit a template.

There are some situations, however, in which you may want to edit the template without rebuilding the wizard. In these cases, make sure that the

template's contents use ID attributes that are defined in a DTD. This allows the XML wizard's changes to be applied to the correct element in the template, even if the content of the template (for example, the order or number of elements) changes.

You might need this functionality if the XML document you intend to produce will have multiple authors, each of which will have authoring control of a different section of the XML document. In this case, the lifecycle of the document might look something like this:

- Define the original template `Original.xml`, with IDs and a DTD. The template has three main sections: A, B, C.
- Define three wizards (`AWizard`, `BWizard`, `CWizard`): `AWizard` allows editing of section A, `BWizard` allows editing of section B, and `CWizard` allows editing of section C.
- User A runs `AWizard` against `Original.xml`, and produces an XML file: `Original.A.xml`. Section A of the document has been completed.
- User B runs `BWizard` against `Original.A.xml`, and produces an XML file: `Original.A.B.xml`. Section B of the document has been completed.
- User C runs `CWizard` against `Original.A.B.xml`, and produces the final XML file: `Original.A.B.C.xml`. All sections of the document have been completed.

#### RELATED CONCEPTS

"XML wizards" on page 503

#### RELATED TASKS

"Creating an XML wizard" on page 504

## Starting the SmartGuide Customizer for XML

You can use the SmartGuide Customizer for XML to build an XML wizard, or SmartGuide, for creating XML documents.

Before you start the Customizer, you should have a sample of the XML document type you want your XML wizard to create. You will use this sample as a template, which the XML wizard's output will be based on.

To start the Customizer, follow these steps:

1. Locate or create the sample XML file you want to start with. If the sample XML has an associated XML DTD, then the sample file must include the DTD, or point to a place where the DTD is available.
2. Run the SmartGuide Customizer. From the command line, type the command:  

```
xmlcustm
```
3. In the Customizer, click **File > Open** and select the example XML file.



The Customizer parses the XML document, and displays its content as a tree of element nodes in the left-hand pane.

You are now ready to begin identifying the elements that your XML wizard will work with.

#### **RELATED CONCEPTS**

“XML wizards” on page 503

“XML template design” on page 505

#### **RELATED TASKS**

“Creating an XML wizard” on page 504

“Defining XML wizard macros”

## **Defining XML wizard macros**

When you load an XML file in the SmartGuide Customizer, you see the structure of the document displayed in a tree view in the left-hand pane of the tool. This structure contains two types of node:

- **Container element nodes**  
These are XML elements that organize sub-elements with content, or have attributes with content, but do not have their own content aside from this.
- **Element content or attribute nodes**  
These are either the content of an XML element, or the value of an element attribute. They appear under a container element. Content nodes are labeled as **Text** in the tree view. Content nodes only appear when there is actual content in the sample XML file; if the element in the sample file has no content, then it does not have a content node in the SmartGuide Customizer.

When you click on a container element, the right-hand pane enables settings for you to define a wizard page for that element’s contents. By default, none of the container elements have wizard pages associated with them.

You can also select whether the element structure is repeatable. If you check the **Repeatable** option, then the user will be able to add multiple instances of the selected element, using a tree view control on the wizard page. Each instance the user adds will have the editable properties you set for the element’s content or attributes.

When you click on an element’s content or attribute, the right-hand pane enables settings for you to make the content or attribute a macro. When you set an element’s content or attribute to be a macro, its value becomes part of the wizard’s XML script.

Macros are also automatically defined when you create a derivation or propagation relationship between elements.

To define an element's content or attribute as a simple wizard macro (without derivation or propagation), follow these steps:

1. In the left-hand pane, click on a content or attribute node. Its settings appear in the right-hand pane.
2. From the **Macro** pulldown, select one of the following:
  - **Hidden**  
The content or attribute value will not appear in the wizard interface, but will be used internally by the wizard (for example, it might have a derived value).
  - **Editable**  
The content or attribute value appears in the wizard interface, and is editable by the wizard user.
  - **Read-only**  
The content or attribute value appears in the wizard interface, but is not editable by the wizard user.

The default value is **None**, which indicates that all settings for the content or attribute are ignored, and the original value from the source XML file is used instead.

3. Type a label for the content or attribute in the **Label** field. If you are creating an Editable or Read-only macro, this label appears in the wizard interface as the label for the associated value (for example, **Name:** ).
4. Type a default value for the content or attribute in the **Default** field's **Value** section. If you do not change the default value, the value from the original sample file is used.

If the definition for the element or attribute in the XML DTD prescribes a list of valid values, then the **Value** section becomes a drop-down list which you can select valid values from. You can also define your own list, without reference to the DTD, by clicking on **Values** and specifying the values you want in the list.

If you provide a list of values (either in the DTD or in the Customizer), and the macro type is **Editable** or **Read Only**, then the wizard user will be presented with this list as well. When set to **Editable**, the macro allows users to select from the list or define their own value. When set to **Read Only**, the macro requires the user to select one of the predefined values. You can customize the terms used in the list (but should not change the underlying values). Click **Values** to customize the terms for the wizard user, as described in the customizing value lists task.

Even if the definition for the element or attribute allows any content type, you can limit the user's choices to a set of values that make sense for the

wizard's intended use. If the macro type is **Editable**, the user will be presented with a drop-down list that contains only the values you specify. You can create or customize the value list the user sees by clicking **Values**, as described in the customizing value lists task.

You can use the prefix and suffix sections to add a prefix or suffix to the value provided by the user. The prefix and suffix sections are also important for deriving or propagating values between related elements, as described in the derivation and propagation tasks.

If the macro type is **Hidden**, then the default value is always applied in the wizard's output, except where overridden by derivation or identity rules, as described in the derivation and propagation tasks, and in the attribute identity topics.

If the macro type is **Read Only**, then the default value is displayed in the wizard, but cannot be changed by the user. If there is a list of valid values, then the user can select which one to apply, but cannot define any new values.

If the macro type is **Editable**, then the default value appears in the wizard interface as a default, and can be typed over by the wizard user.

5. If the attribute is the ID for the element, set the "Attribute identity options" on page 533 to **ID**. If the current element is repeatable, identify any referencing elements that need to be created to match new instances of the current (referenced) element.
  6. If the attribute is an ID reference to another element, set the "Attribute identity options" on page 533 to **Reference to ID**. If the current element is repeatable, specify whether a new target element should be created to match each new referencing element (**Point to existing target** or **Create new target**).
- If you select to create a new target element, specify how and when the target element should be created (**Select Owner** and **Select Target**).
7. If the macro type is **Editable**, select the constraint (if any) you want to apply to the user's input. You can select from the following values:

- NoSpace
- C++
- CORBA
- SQL
- LongFile
- File83
- File8
- Any
- Action

You can also define your own constraints, as described in the constraining values task.

8. If the macro type is **Editable**, type a brief description of the element content or attribute in the **Fly-Over Help** field. This description will appear when the wizard user moves the mouse pointer over the field.

#### **RELATED CONCEPTS**

- “XML wizards” on page 503
- “Attribute identity in XML” on page 520

#### **RELATED TASKS**

- “Creating an XML wizard” on page 504
- “Customizing value lists in an XML wizard”
- “Deriving values in an XML wizard” on page 511
- “Propagating values in an XML wizard” on page 513
- “Constraining values in an XML wizard” on page 515

#### **RELATED REFERENCES**

- “Macro setting” on page 530
- “Element properties” on page 530
- “Attribute properties” on page 531
- “Attribute identity options” on page 533
- “Constraints” on page 535

## **Customizing value lists in an XML wizard**

When you define a macro of type **Editable** for an element content or attribute, the wizard user will be able to enter a value for that content or attribute in the wizard. You can provide the user with a set of values to choose from that make sense for the wizard’s intended use. These values will be displayed in the wizard interface in one of the following forms, depending on the number of possible values:

**Two** Check box (for choices such as True or False, Yes or No, Y or N)

#### **Three to Four**

Radio buttons in a group

#### **More than Four**

Drop-down list

If you have more than four choices, then the user also has the option to type in a different value in the list field. To restrict the user to the listed choices only, change the macro type to **Read only**.

If an element content or attribute already has a set of acceptable values enumerated in the DTD, the SmartGuide Customizer for XML displays them

as a set of choices by default. You can customize the terms used in the list to make them more descriptive for your wizard users, but should not change the underlying values; otherwise the XML wizard will generate XML that is not valid.

To create or customize a value list, follow these steps:

1. In the SmartGuide Customizer tree view, click on the element's content or attribute whose value list you want to customize.

In the right-hand pane, the properties of the selected node appear. The **Default** field's **Value** section should be a drop-down list. If the section is an entry field (no drop-down arrow), then the content or attribute does not have a list of acceptable values in the XML DTD, and there is no value selection list to customize.

2. Click the **Values** button. The Customize Value List dialog opens. The existing value appears in the list as the default. Any additional valid values defined in the DTD appear as well.
3. If the DTD defines a list of valid values, you can change the terms used in the list, or remove terms from the list, but should not change the underlying values or add new values that are not compliant with the DTD definition.
4. If the DTD does not define a list of valid values, you can change or add new values, and customize the terminology, as required.
5. Select which term you want to be selected by default.
6. Click **OK**. The **Value** section of the **Default** field becomes a drop-down list, if it was not already such a list.

When you create the XML wizard, the wizard user will be able to select a value for this element content or attribute from the list you created. The user will see the terms you specified, which will map to the underlying values according to the mapping you created.

#### **RELATED CONCEPTS**

"XML wizards" on page 503

#### **RELATED TASKS**

"Creating an XML wizard" on page 504

"Defining XML wizard macros" on page 507

"Constraining values in an XML wizard" on page 515

## **Deriving values in an XML wizard**

When you create an XML wizard with the SmartGuide Customizer for XML, you can derive the value for one element's content or attribute from the value given for another element's content or attribute, rather than identifying the

values separately. For example, you could make the first value editable, and then the second, derived value be a simple reflection of what the user enters for the first value, with the addition of a prefix or suffix.

This task deals with creating a derivation relationship, starting with the derived element value and specifying what its source should be. You can also work in the other direction to create a propagation relationship, starting with an existing value and propagating it to a number of deriving elements.

There are two ways you can create a derivation relationship:

- Using the implicit derivation rules in your XML sample document, based on the comparison of content strings in the different elements.
- Making an explicit association, regardless of the content strings in the sample document.

To create a derivation relationship based on the existing content of the sample document, follow these steps:

1. Select the element content or attribute for which you want to specify a source.
2. In the **Default** field's **Value** section, type the substring of its value that was derived, in the original XML file.  
For example, if you know that the attribute's current value PersonBO is derived from another attribute's value Person, then decompose the value into **Value** of Person and **suffix** of BO. You can then search to find all other attributes or element content with the value Person, and select one of them as the source to be derived from.
3. In the toolbar, click the **Derive** button. A Derive Value dialog appears, and highlights the first node in the tree view that has the value you listed.
4. Click **Find Next** or **Find Previous** to navigate through all the nodes in the tree that have the listed value, and are valid sources for a derivation relationship.
5. When you have found the node you want to derive from, click the **Derive From** button.

The derivation relationship is created. The **Derived value** option is checked, and the selected source node is marked as a macro. The **Default** field's **Value** section is greyed out, to prevent you from editing the derived value.

6. Mark the content or attribute as a macro (**Hidden**, **Editable**, or **Read-only**). The relationship will be implemented in the XML wizard.

To create a derivation relationship regardless of existing content, follow these steps:

1. In the tree view, locate the element content or attribute that you want to specify as derived.

2. From the pop-up menu of the node, click **Derive Value From**.

A Derivation Tree appears. You can select any node in the tree as the source to derive from. You should ensure that the node you select to derive from has a value associated with it, and that the value is of an appropriate type to act as the source for the deriving value.

3. Select the node you want to derive from.
4. Click **OK**.

The derivation relationship is created. The **Derived value** option is checked, and the selected source node is marked as a macro. The **Default** field's **Value** section is greyed out, to prevent you from editing the derived value.

5. Fill in any prefix or suffix that you want to apply to the derived value.
6. Mark the content or attribute as a macro (**Hidden**, **Editable**, or **Read-only**). The relationship will be implemented in the XML wizard.

#### **RELATED CONCEPTS**

"XML wizards" on page 503

#### **RELATED TASKS**

"Creating an XML wizard" on page 504

"Defining XML wizard macros" on page 507

"Propagating values in an XML wizard"

## **Propagating values in an XML wizard**

When you create an XML wizard with the SmartGuide Customizer for XML, you can derive the value for one element from the value given for another element, rather than identifying the values separately. In other words, you can base the value for an element content or attribute on the value given for another element's content or attribute within the same XML document. For example, you could make the first value editable, and then the second, derived value be a simple reflection of what the user enters for the first value, with the addition of a prefix or suffix.

This task deals with creating a propagation relationship, starting with the source content or attribute value and specifying what other values are derived from it. You can also work in the other direction to create a derivation relationship, starting with derived values and specifying where the value should be derived from.

There are two ways you can create a propagation relationship:

- Using the implicit propagation rules in your XML sample document, based on the comparison of content strings in the different elements.

- Making an explicit association, regardless of the content strings in the sample document.

To create a propagation relationship based on the existing content of the sample document, follow these steps:

1. Select the element content or attribute whose value you want to propagate.
2. In the toolbar, click the **Propagate** button. A Propagate Value dialog appears, and highlights the first node in the tree view that has the selected value.

For example, if the current value is Person, then you can search through all other nodes whose value includes that substring (PersonBO, PersonDOImpl, iPersonPO, and so on).

3. Click **Find Next** or **Find Previous** to navigate through all the nodes in the tree that have the listed value, and are valid targets for a propagation relationship.
4. When you have found a node you want to propagate to, click the **Propagate To** button.

A propagation relationship is created. The selected target's **Derived value** option is checked, and the source node is marked as a macro. The selected target's **Default** field's **Value** section is greyed out, to prevent you from editing the derived value.

The Propagate Value dialog remains open, for you to select additional nodes to propagate to.

5. When you have finished propagating values, click **Cancel** to close the dialog.
6. Edit each propagation target:
  - a. Fill in any prefix or suffix that you want to apply to the derived value.
  - b. Mark the target node as a macro (**Hidden**, **Editable**, or **Read-only**). The relationship will be implemented in the XML wizard.

To create a propagation relationship regardless of existing content, follow these steps:

1. In the tree view, locate the element content or attribute whose value you want to propagate.
2. From the pop-up menu of the node, click **Propagate Value To**.

A Propagation Tree appears. You can select any nodes in the tree as targets to propagate values to. You should ensure that the nodes you select to propagate to have values associated with them, and that the values are of an appropriate type to act as the target for your source value.

3. Select the nodes you want to propagate to.
4. Click **OK**.



The propagation relationships are created. The selected targets' **Derived value** options are checked, and the source node is marked as a macro. The selected targets' **Default** fields' **Value** sections are greyed out, to prevent you from editing the derived value.

5. Edit each propagation target:
  - a. Fill in any prefix or suffix that you want to apply to the derived value.
  - b. Mark the target node as a macro (**Hidden**, **Editable**, or **Read-only**).  
The relationship will be implemented in the XML wizard.

#### **RELATED CONCEPTS**

"XML wizards" on page 503

#### **RELATED TASKS**

"Creating an XML wizard" on page 504

"Defining XML wizard macros" on page 507

"Deriving values in an XML wizard" on page 511

## **Constraining values in an XML wizard**

When you define an XML wizard macro as **Editable** in the SmartGuide Customizer for XML, you can also select a constraint to apply to it. This will prevent the wizard user from entering a value outside the selected constraint.

The following constraints are provided by default:

- NoSpace
- C++
- CORBA
- SQL
- LongFile
- File83
- File8
- Any
- Action

You can provide your own constraint by creating a Java class that implements the Constraint interface, provided with the SmartGuide Customizer for XML. To apply the constraint, select the **Actions** option in the **Constraints** field, and then type over the selection with the name of the class. When the XML wizard runs, it will look for a Java class with that name, and call its test() function with the value the user entered as a parameter.

#### **RELATED CONCEPTS**

"XML wizards" on page 503

#### RELATED TASKS

“Creating an XML wizard” on page 504

“Defining XML wizard macros” on page 507

“Customizing value lists in an XML wizard” on page 510

## Defining the layout of an XML wizard

The XML wizard you create, using the SmartGuide Customizer for XML, will walk through all the macros you identify, displaying entry fields or drop-down lists for macros that are **Editable** and displaying read-only text for macros that are **Read-only**.

Macros are generally grouped by the element that contains them. For each element that contains macros, you can select whether to start a new page for the contained and any subsequent macros.

To define an XML wizard page, follow these steps:

1. In the SmartGuide Customizer tree view, click on the element you want to define a page for.
2. In the properties pane, select whether the element is **Repeatable**.  
If you make an element repeatable, then the wizard will display a tree view for the element, to which the user can add instances of the element. Each element instance will have its own set of values, as defined in the SmartGuide Customizer. Values marked as **Editable** are editable by the user, on a per-instance basis.
3. In the properties pane, click the **Start new page** option.  
The values of the current element will now be on a new page. Values of subsequent elements will also appear on the current page, until the next element defined as the start of a page.  
Generally, if you make an element **Repeatable**, it should have its own page. In other words, it should have the **Start new page** option checked, and the next element that contains **Editable** or **Read-only** macros should also have the **Start new page** option checked.
4. Type a title for the new page in the **Title** field.
5. Type a description for the new page in the **Description** field. The description appears directly below the title, and above any editing controls for the element contents and attributes on the page.
6. Type a URL for an HTML file that provides help for the page in the **Help URL** field. The URL can be absolute (for example, <http://mycompany.intranet/product/wizard1/NamePage.html>) or relative to the location of the wizard macro script (for example, <help/NamePage.html>). This URL will be associated with the **Help** button on the wizard page, and the HTML file will be loaded in the user’s default web browser when the user clicks **Help**.

Once you have defined the layout, you can view the results from within the SmartGuide Customizer by testing the XML wizard.

#### RELATED CONCEPTS

“XML wizards” on page 503

#### RELATED TASKS

“Creating an XML wizard” on page 504

“Defining XML wizard macros” on page 507

“Testing an XML wizard”

## Testing an XML wizard

Once you have selected the elements you want represented in the XML wizard, and customized the XML wizard’s layout, you can save the XML macro file for the wizard, and test it through the SmartGuide Customizer’s interface.

To generate the XML wizard script, follow these steps:

1. Select **File - Save**.
2. Select a location in which to save the file.
3. Type the name of the file as *name.xml*.
4. Save the file.

Once you have generated the script, and before you use it to create new files, you can test it from within the SmartGuide Customizer. To test the script, follow these steps:

1. Select **File > Test**.
2. Run through the wizard pages, and review the result of your layout selections in the SmartGuide Customizer.
3. Click **Finish**.

You are prompted for the location of the original XML file (on which the wizard is based), and a path and file name for the resulting wizard-generated XML file.

4. Provide the names and click **Finish**, or click **Cancel** to return to the SmartGuide Customizer without saving the results of your test.

You can also run the wizard script from the command line, by running the SmartGuide Launcher for XML (type `xmllaunch` on the command line).

#### RELATED CONCEPTS

“XML wizards” on page 503

#### RELATED TASKS

- “Creating an XML wizard” on page 504
- “Defining XML wizard macros” on page 507
- “Defining the layout of an XML wizard” on page 516
- “Running an XML wizard”

## Running an XML wizard

Once you have created an XML wizard script in the SmartGuide Customizer for XML, you can run the wizard using the SmartGuide Launcher for XML, and use the wizard to create new XML documents.

In order to run an XML wizard, you need the following:

- The XML wizard script, generated by the SmartGuide Customizer for XML.
- Any help files for the wizard script, if they are linked using a relative path (rather than, for example, an http address on your intranet).
- The original XML file on which the wizard script is based.
- The DTD for the original XML file, either contained in, or referenced by, the original file. If the DTD is referenced using a relative path, the path is resolved relative to the current directory (the directory from which the SmartGuide Launcher tool is run).
- Any classes that provide customized input constraints.
- The SmartGuide Launcher tool, which runs the script.

To run an XML wizard, follow these steps:

1. On the command line, type the following command:  
`xmllaunch`  
The SmartGuide Launcher wizard opens.
2. Type the name of the wizard script you want to run.
3. Click **Finish**.  
The XML wizard opens.
4. In the XML wizard, follow the prompts to add elements and edit element values. Hover help, and HTML help for each page, are available if they were defined in the SmartGuide Customizer.
5. Click **Finish**.  
You are prompted for the location of the original file on which the script is based, and a location in which to save the new document generated by the wizard.
6. Provide the location of the original source XML file.
7. Provide a name and path in which to save the new XML document, which is based on that original.
8. Click **Finish**.

The XML file is created, with the name and path you specified.

#### **RELATED CONCEPTS**

“XML wizards” on page 503

“Model interchange with XML” on page 492

#### **RELATED TASKS**

“Creating an XML wizard” on page 504

“Editing an XML wizard”

“Importing XML” on page 389

“Distributing an XML wizard”

## **Editing an XML wizard**

You can customize an XML wizard by loading its XML script file into the SmartGuide Customizer for XML, selecting elements to expose in the wizard interface, setting how they will be exposed, and then regenerating the XML wizard script.

To open an existing XML wizard script in the SmartGuide Customizer, follow these steps:

1. Run the SmartGuide Customizer. From the command line, type the command:  
`xmlcust`
2. Open the script file in the SmartGuide Customizer. Click **File > Open** and select the file.

The SmartGuide Customizer recognizes the XML file as a script or macro file, and displays it in terms of its original source document structure (rather than interpreting the contents of the file literally), with the macros applied as a set of modifications and selections.

3. Edit the macros, and create new ones, in the same you would when creating a new XML wizard.
4. Click **File > Save** to save the XML script with your changes applied.

#### **RELATED CONCEPTS**

“XML wizards” on page 503

#### **RELATED TASKS**

“Creating an XML wizard” on page 504

“Running an XML wizard” on page 518

## **Distributing an XML wizard**

Once you have created and tested the XML wizard, you can package it for use by others.

Each package should include the following:

- The XML wizard script, generated by the SmartGuide Customizer for XML.
- Any help files for the wizard script, if you linked to the files using a relative path (rather than, for example, an http address on your intranet).
- The original XML file on which the wizard script is based.
- The DTD for the original XML file, either contained in, or referenced by, the original file. If the DTD is referenced using a relative path, the path is resolved relative to the current directory (the directory from which the SmartGuide Launcher tool is run).
- Any classes you defined to provide customized input constraints.
- The SmartGuide Launcher tool, which runs the script.

The relative paths from the wizard script's location to the original XML file and its DTD should be preserved in the package.

#### **RELATED CONCEPTS**

"XML wizards" on page 503

#### **RELATED TASKS**

"Creating an XML wizard" on page 504

"Constraining values in an XML wizard" on page 515

"Testing an XML wizard" on page 517

"Running an XML wizard" on page 518

---

## **Attribute identity in XML**

You can use the SmartGuide Customizer for XML to define an XML wizard that allows users to define new XML documents based on an initial template.

When the initial template contains elements with IDs, and references between elements, then you need to decide how issues of attribute identity will be resolved.

Attribute identity primarily has implications for elements that are repeatable. When you specify in the Customizer that an element is repeatable, the user of the resulting XML wizard can add multiple instances of the element. If the element contains an ID attribute, or references other elements by ID, you can use the attribute identity options in the attribute's property page to specify how the new element instances will relate to other elements (that is, how references involving the new instance will be resolved).

There are three main cases for a repeatable element with attributes that involve identity:

- “XML ID attributes”
- “XML references” on page 522
- “XML references with customized targets” on page 525

#### RELATED CONCEPTS

“XML wizards” on page 503

#### RELATED TASKS

“Creating an XML wizard” on page 504

“Defining XML wizard macros” on page 507

#### RELATED REFERENCES

“Attribute identity options” on page 533

## XML ID attributes

When you create an XML wizard using the SmartGuide Customizer for XML, you can specify elements as repeatable. If the element has an ID attribute (that uniquely identifies the element within the document), then you need to specify how to handle the ID attribute when the wizard user creates new element instances.

The first step is identifying the attribute as an ID. In the SmartGuide Customizer, click on the attribute in the Contents pane to display its properties. Then in the Properties pane, click the **ID** option. In the XML wizard, when a user adds multiple instances of the element, the additional instances will be assigned new IDs by the XML wizard. The new IDs are generated using a DCE-compliant algorithm and are unique.

When you specify that an attribute is an ID, the **Referencing Elements** choice is enabled. This allows you to relate the creation of specific referencing elements to the creation of the current element. When a new instance of the current element is created, new instances of the selected referencing elements will be created to match.

### Example: selecting referencing elements

You create an XML template file that contains the following:

```
<Student xmi.id="key1" name="Richard"/>
<University.allStudents>
 <UStudent xmi.idref="key1"/>
</University.allStudents>
```

In the SmartGuide Customizer, you specify:

- Student is a repeatable element.
- Student//name is an editable attribute (in fact, the only editable attribute).

- Student//xmi.id is the ID for Student, with UStudent as a referencing element.
- UStudent//xmi.idref is a reference to Student//xmi.id

You run the resulting XML wizard. You are prompted to add students. The default entry is “Richard” (the ID is not shown). You add a second student named “Mike”, and click **Finish**. The final result is an XML file that contains the following:

```
<Student xmi.id="key1" name="Richard"/>
<Student xmi.id="key2" name="Mike"/>
<University.allStudents>
 <UStudent xmi.idref="key1"/>
 <UStudent xmi.idref="key2"/>
</University.allStudents>
```

The new Student element `<Student xmi.id="key2" name="Mike"/>` has a name specified by the user, and an ID generated by the wizard (in actual fact the ID would be DCE-compliant).

The new UStudent element is generated by the wizard because it has been identified as a referencing element in the SmartGuide Customizer, in the Properties pane for the Student//xmi.id attribute.

#### RELATED CONCEPTS

“XML wizards” on page 503

“Attribute identity in XML” on page 520

“XML references”

“XML references with customized targets” on page 525

#### RELATED TASKS

“Creating an XML wizard” on page 504

“Defining XML wizard macros” on page 507

#### RELATED REFERENCES

“Attribute identity options” on page 533

## XML references

When you create an XML wizard using the SmartGuide Customizer for XML, you can specify elements as repeatable. If the element has an attribute that is a reference to another element (using the other element’s ID attribute), then you need to specify how to handle the reference when the wizard user creates new element instances.

The first step is identifying the attribute as a reference. In the SmartGuide Customizer, click on the attribute in the Contents pane to display its properties. Then in the Properties pane, click the **Reference to ID** option.



Once an attribute is identified as a reference, its default value is used to associate it with the ID it references (which contains a matching default value). When the reference is part of a repeatable element, you need to specify how to resolve the reference when new instances are added. You have two options:

- **Point to existing target**

The new instance will reference the same ID as the original.

- **Create new target**

The new instance will point to a new target that is created to match. If you select this option, then you can customize the way in which the creation takes effect, as described in “XML references with customized targets” on page 525.

### **Example: pointing to existing target**

You create an XML template file that contains the following:

```
<Tenant xmi.id="tenantKey1" name="Simpson">
 <TenantHome xmi.idref="houseKey1">
</Tenant>
<House xmi.id="houseKey1" address="some_address"/>
```

In the SmartGuide Customizer, you specify:

- Tenant as a repeatable element.
- Tenant//name as an editable attribute.
- Tenant//xmi.id as the ID for the family.
- TenantHome as a hidden element.
- TenantHome//xmi.idref as a reference to House//xmi.id, with **Point to existing target** set.
- House as a hidden element.
- House//xmi.id as the ID for the house.

You run the resulting XML wizard. You are prompted to add families. The default entry is “Simpson” (the ID is not shown). You add a second family named “Johnson”, and click **Finish**. The final result is an XML file that contains the following:

```
<Tenant xmi.id="tenantKey1" name="Simpson">
 <TenantHome xmi.idref="houseKey1">
</Tenant>
<Tenant xmi.id="tenantKey2" name="Johnson">
 <TenantHome xmi.idref="houseKey1">
</Tenant>
<House xmi.id="houseKey1" address="some_address"/>
```

The new Tenant element `<Tenant xmi.id="tenantKey2" name="Johnson">` has a name specified by the user, and an ID generated by the wizard (unlike the example the ID would be DCE-compliant).

The new tenant home element `<TenantHome xmi.idref="houseKey1">` has been automatically created for the new Tenant element. Because it was identified as a reference to ID, and had the option **Point to existing target** selected, it points to the existing house: both tenants are living at the same address.

### Example: creating new target

You create an XML template file that contains the following:

```
<Family xmi.id="familyKey1" name="Simpson">
 <FamilyHome xmi.idref="houseKey1">
</Family>
<House xmi.id="houseKey1" address="some address"/>
```

In the SmartGuide Customizer, you specify:

- Family as a repeatable element.
- Family//name as an editable attribute.
- Family//xmi.id as the ID for the family.
- FamilyHome as a hidden element.
- FamilyHome//xmi.idref as a reference to House//xmi.id, with **Create new target** set.
- House as a hidden element.
- House//xmi.id as the ID for the house.

You run the resulting XML wizard. You are prompted to add families. The default entry is “Simpson” (the ID is not shown). You add a second family named “Johnson”, and click **Finish**. The final result is an XML file that contains the following:

```
<Family xmi.id="familyKey1" name="Simpson">
 <FamilyHome xmi.idref="houseKey1">
</Family>
<Family xmi.id="familyKey2" name="Johnson">
 <FamilyHome xmi.idref="houseKey2">
</Family>
<House xmi.id="houseKey1" address="some_address"/>
<House xmi.id="houseKey2" address="some_address"/>
```

The new Family element `<Family xmi.id="familyKey2" name="Johnson">` has a name specified by the user, and an ID generated by the wizard (unlike the example the ID would be DCE-compliant).

The new FamilyHome element `<FamilyHome xmi.idref="houseKey2">` has been automatically created for the new Family element. Because it was identified as a reference to ID, and had the option **Create new target** selected, a new target House element has been created to match. The new target has an ID generated by the wizard (again, the ID would be DCE-compliant, which is not shown here).

The new House element `<House xmi.id="houseKey2" address="some_address"/>` has been generated by the wizard to resolve the reference from FamilyHome, which in turn was added because the user added a new Family element.

#### RELATED CONCEPTS

“XML wizards” on page 503  
“Attribute identity in XML” on page 520  
“XML ID attributes” on page 521  
“XML references with customized targets”

#### RELATED TASKS

“Creating an XML wizard” on page 504  
“Defining XML wizard macros” on page 507

#### RELATED REFERENCES

“Attribute identity options” on page 533

## XML references with customized targets

When you create an XML wizard using the SmartGuide Customizer for XML, you can specify elements as repeatable. If the element has an attribute that is a reference to another element (using the other element’s ID attribute), then you can specify that the referenced element should be created along with the referencing element.

The first step is identifying the attribute as a reference. In the SmartGuide Customizer, click on the attribute in the Contents pane to display its properties. Then in the Properties pane, click the **Reference to ID** option.

Once an attribute is identified as a reference, its default value is used to associate it with the ID it references (which contains a matching default value). Select **Create new target** to specify that a new instance of the referenced element should be created when a new instance of the current element is created. In other words, when a new referencing instance is created, a new referenced instance will be created to match. The new referenced instance will have a generated ID, and otherwise default content.

Once you have selected **Create new target**, you can customize the way in which the new target is created. You have two choices, each of which allow you to customize one part of the creation process:

- **Select Owner**

By default, the new target element is created when the current (referencing) element is created. This is appropriate for simple cases, when the current element is created directly. However, if the current element is created

indirectly (for example, as part of the content of a higher-level element, or as part of a chain of references), then you should specify the directly created element as the owner of the reference. This ensures that the reference target will be created when the current element is created indirectly.

- **Select Target**

By default, the owner of the referenced ID is the target element. This is appropriate for simple cases, when the referenced element has no other dependencies. However, if the referenced element has a context that needs to be created with it (for example, a higher-level element that always contains just one instance of the referenced element), then you should select the appropriate target (for example, the higher-level or parent element).

**Example: selected owner**

You create an XML template file that contains the following:

```
<TenantFamily xmi.id="familyKey1" name="Simpson">
 <Child xmi.id="childKey1" name="Julia">
<Child.Toy xmi.idref="tbearKey1"/>
 </Child>
 <TenantHome xmi.idref="houseKey1"/>
</TenantFamily>
<House xmi.id="houseKey1" name="House">
 <Toy.Bear xmi.id="tbearKey1" name="Bear"/>
 <Toy.Car xmi.id="tcarKey1" name="Car"/>
</House>
```

The important reference for this example is from Child.Toy to Toy.Bear. In the SmartGuide Customizer, you specify:

- TenantFamily as a repeatable element.
- TenantFamily//name as an editable attribute.
- TenantFamily//xmi.id as the ID for the family.
- Child//xmi.id as the ID for the child.
- Child.Toy//xmi.idref as a reference to Toy.bear//xmi.id, with **Create new target** set.
- TenantHome as a hidden element.
- TenantHome//xmi.idref as a reference to House//xmi.id, with **Point to existing target** set.
- House as a hidden element.
- House//xmi.id as the ID for the house.
- Toy.Bear//xmi.id is the ID for the toy bear.
- Toy.Car//xmi.id is the ID for the toy car.

With the settings so far, a new family would automatically point to the same address (based on the setting for TenantHome//xmi.idref). Each new family has a child, which owns a reference to a toy bear. This reference does not

resolve. The reference has been set to **Create new target**, which works fine when the owning element is created directly, or with only one level of indirection. In other words, if Child.Toy were created directly, or Child were created directly, then the **Create new target** option would take effect, and a new toy would be created. But in this case, the only element that is created directly is the TenantFamily. It needs to be explicitly identified as the owner of the reference (Child.Toy//xmi.idref(), for the purpose of creating the new target.

Return to the SmartGuide Customizer, and the Properties pane for Child.Toy//xmi.idref . The **Create new target** option is already set, and there are two buttons available: **Select Owner** and **Select Target**. Click on **Select Owner** to display the Select Owner window. Within the window, select Family, and click **OK**.

You run the resulting XML wizard. You are prompted to add families. The default entry is "Simpson" (the ID is not shown). You add a second family named "Johnson", and click **Finish**. With the revised settings, a new family automatically gets a new house, and the new family's child automatically gets a new toy.

The final result is an XML file that contains the following:

```
<TenantFamily xmi.id="familyKey1" name="Simpson">
 <Child xmi.id="childKey1" name="Julia">
<Child.Toy xmi.idref="tbearKey1"/>
 </Child>
 <TenantHome xmi.idref="houseKey1"/>
</TenantFamily>
<TenantFamily xmi.id="familyKey2" name="Johnson">
 <Child xmi.id="childKey2" name="Julia">
<Child.Toy xmi.idref="tbearKey2"/>
 </Child>
 <TenantHome xmi.idref="houseKey2"/>
</TenantFamily>
<House xmi.id="houseKey1" name="House">
 <Toy.Bear xmi.id="tbearKey1" name="Bear"/>
 <Toy.Bear xmi.id="tbearKey2" name="Bear"/>
 <Toy.Car xmi.id="tcarKey1" name="Car"/>
</House>
```

The new TenantFamily element <TenantFamily xmi.id="familyKey2" name="Johnson"> has a name specified by the user, and an ID generated by the wizard (unlike the example the ID would be DCE-compliant).

The new TenantHome element <TenantHome xmi.idref="houseKey2"> has been automatically created for the new TenantFamily element. Because it was

identified as a reference to ID, and had the option **Point to existing target** selected, it points to the existing house: both tenants are living at the same address.

The new Child element `<Child xmi.id="childKey2" name="Julia"/>` has also been automatically created for the new TenantFamily element. It includes a new Child.Toy element `<Child.Toy xmi.idref="tbearKey2"/>`. Because it was identified as a reference to ID, had the option **Create new target** selected, and had the owner of the reference set to be TenantFamily (the only element being created directly), a new Toy.Bear element has been added to the House element.

### Example: selected target

You create an XML template file that contains the following:

```
<Family xmi.id="familyKey1" name="Simpson">
 <FamilyHome xmi.idref="houseKey1"/>
</Family>
<Property>
 <House xmi.id="houseKey1" name="House"/>
 <Pool xmi.id="poolKey1" name="Pool"/>
 <Garden xmi.id="gardenKey1" name="Garden"/>
</Property>
```

In the SmartGuide Customizer, you specify:

- Family as a repeatable element.
- Family//name as an editable attribute.
- Family//xmi.id as the ID for the family.
- FamilyHome as a hidden element.
- FamilyHome//xmi.idref as a reference to House//xmi.id, with **Create new target** set.
- House as a hidden element.
- House//xmi.id as the ID for the house.
- Pool//xmi.id as the ID for the pool.
- Garden//xmi.id as the ID for the garden.

With the settings so far, a new house would be created for each new family, but all the houses would be on the same property (the FamilyHome reference points to the House element within Property, not to the parent Property element).

To specify that a new property should be created for each family, you can modify the target of the FamilyHome reference (for the purposes of this creation action).

Return to the SmartGuide Customizer, and the Properties pane for FamilyHome//xmi.idref . The **Create new target** option is already set, and there are two buttons available: **Select Owner** and **Select Target**. Click on **Select Target** to display the Select Target window. Within the window, select Property, and click **OK**.

You run the resulting XML wizard. You are prompted to add families. The default entry is "Simpson" (the ID is not shown). You add a second family named "Johnson", and click **Finish**. The final result is an XML file that contains the following:

```
<Family xmi.id="familyKey1" name="Simpson">
 <FamilyHome xmi.idref="houseKey1">
</Family>
<Family xmi.id="familyKey2" name="Johnson">
 <FamilyHome xmi.idref="houseKey2">
</Family>
<Property>
 <House xmi.id="houseKey1" name="House"/>
 <Pool xmi.id="poolKey1" name="Pool"/>
 <Garden xmi.id="gardenKey1" name="Garden"/>
</Property>
<Property>
 <House xmi.id="houseKey2" name="House"/>
 <Pool xmi.id="poolKey2" name="Pool"/>
 <Garden xmi.id="gardenKey2" name="Garden"/>
</Property>
```

The new Family element `<Family xmi.id="familyKey2" name="Johnson">` has a name specified by the user, and an ID generated by the wizard (unlike the example the ID would be DCE-compliant).

The new FamilyHome element `<FamilyHome xmi.idref="houseKey2">` has been automatically created for the new Family element. Because it was identified as a reference to ID, and had the option **Create new target** selected, a new target House element, along with a new parent Property element and sibling Pool and Garden topics, have been created to match. The new target and its sibling elements have IDs generated by the wizard (again, the IDs would be DCE-compliant, which is not shown here).

#### RELATED CONCEPTS

- "XML wizards" on page 503
- "Attribute identity in XML" on page 520
- "XML ID attributes" on page 521
- "XML references" on page 522

#### RELATED TASKS

- "Creating an XML wizard" on page 504
- "Defining XML wizard macros" on page 507

#### RELATED REFERENCES

“Attribute identity options” on page 533

---

## XML wizard properties

### Macro setting

In the SmartGuide Customizer properties pane, you can define values and constraints for the currently selected content (text) or attribute node. The macro setting does not affect elements.

The changes you make to the properties of an attribute or element content are applied according to the setting of the **Macro** control:

- **None**  
Any changes are ignored.
- **Hidden**  
Your changes are applied by the wizard script, but are not exposed to the wizard user.
- **Read-only**  
The value for the content or attribute is exposed in the wizard interface as read-only text.
- **Editable**  
The value for the content or attribute is editable in the wizard interface.

#### RELATED CONCEPTS

“XML wizards” on page 503

#### RELATED TASKS

“Creating an XML wizard” on page 504

### Element properties

In the SmartGuide Customizer for XML, you can define the layout of an XML wizard by setting the properties of selected element nodes in the Contents pane.

The element properties:

- Define whether to start a new page with the controls associated with the selected element, in the wizard interface
- Define whether the attributes and content for this element are a repeatable set (the user can add more than one set of values)

The macro settings for an element definition are ignored. Only the macro settings for the element’s content or attributes are applied.



You have the following controls:

- **Repeatable**  
If you make an element repeatable, then the wizard will display a tree view for the element, to which the user can add instances of the element. Each element instance will have its own set of values, as defined in the SmartGuide Customizer. Values marked as **Editable** are editable by the user, on a per-instance basis.
- **Start new page**  
Click this option to start a new page with the content and attributes exposed under this element. Once you select this option, you can provide a title and description for the page, and associate a URL for the **Help** button on that page.
- **Title**  
Type a title for the page.
- **Description**  
Type a description for the page. The description appears directly below the title, on the page.
- **Help URL**  
Type a URL for a help file that describes the controls on this page. The URL can be absolute (for example, `http://myserver/help1.htm` ) or relative to the location of the XML wizard script (for example, `help/help1.htm`).

#### **RELATED CONCEPTS**

“XML wizards” on page 503

#### **RELATED TASKS**

“Creating an XML wizard” on page 504

“Defining the layout of an XML wizard” on page 516

## **Attribute properties**

In the SmartGuide Customizer for XML, you can set the following properties for an attribute. Click on an attribute or text node in the Contents pane to display and set its options in the Properties pane.

The options you set are applied according to the current selection in the “Macro setting” on page 530 list.

- **Derived value**  
Indicates that the value for the current content or attribute is derived in part from another value elsewhere in the XML structure. The derived value option is checked automatically when you derive a value for the content or attribute from its pop-up menu in the Contents pane, or when you propagate a value to the content or attribute from another node’s pop-up menu. Once the option is checked, you can uncheck it to break the derivation or propagation relationship.

- **Label**  
Type a label for the element content or attribute. If the **Macro** setting is **Read-only** or **Editable**, this becomes the label for the field in the wizard user interface.
- **Default**  
Type a default value for the content or attribute. You can split the value into prefix, value, and suffix to create derivation or propagation relationships with other values in the structure. An existing value may be provided based on the content of the source XML file being used as a template.
- **Attribute identity**  
Select the type of identity the attribute represents:
  - None (page 533)
  - ID (page 533)
  - Reference to ID (page 534)

When an element is specified as repeatable, the user can add multiple instances of the element in an XML wizard. You can use the attribute identity options to specify how the new element instances will relate to other elements (that is, how references involving the new instance will be resolved).

- **Constraints**  
Select a constraint to apply to the value a user can enter for the content or attribute. Only applies when the **Macro** setting is **Editable**. You can set one of the following constraints:
  - NoSpace
  - C++
  - CORBA
  - SQL
  - LongFile
  - File83
  - File8
  - Any
  - Action
- **Fly-over Help**  
Provide a fly-over description for the label and field in the wizard user interface. Only applies when the **Macro** setting is **Editable**.
- **Values**  
Displays a dialog in which you can specify or modify the set of acceptable values for the current content or attribute. When you specify more than one acceptable value, the wizard user is presented with a drop-down list to choose from, rather than an editable field. For each value, you can specify

the term to display in the drop-down list, and the associated value to generate in the XML. Only applies when the **Macro** setting is **Editable**.

#### **RELATED CONCEPTS**

“XML wizards” on page 503

“Attribute identity in XML” on page 520

#### **RELATED TASKS**

“Creating an XML wizard” on page 504

“Defining XML wizard macros” on page 507

“Customizing value lists in an XML wizard” on page 510

“Deriving values in an XML wizard” on page 511

“Propagating values in an XML wizard” on page 513

“Constraining values in an XML wizard” on page 515

#### **RELATED REFERENCES**

“Attribute identity options”

## **Attribute identity options**

In the SmartGuide Customizer for XML, an attribute can be assigned the following types of identity:

- None
- ID
- Reference to ID

Attribute identity can have implications for elements that are repeatable. When an element is specified as repeatable, the user can add multiple instances of the element in an XML wizard. You can use the attribute identity options to specify how the new element instances will relate to other elements (that is, how references involving the new instance will be resolved).

The following sections describe the implications of each option, and include descriptions of the additional choices available when **ID** or **Reference to ID** is selected.

### **None**

The attribute is neither an ID nor a reference to an ID. It is simply data. There are no special implications for this choice.

### **ID**

The attribute is the ID for its element. If the element is specified as repeatable (that is, the user can add multiple instances of this element), then the additional instances will be assigned new IDs by the XML wizard. The new IDs are generated using a DCE-compliant algorithm and are unique.

When you specify that an attribute is an ID, the **Referencing Elements** choice is enabled. This allows you to tie the creation of specific referencing elements to the creation of the current element. When a new instance of the current element is created, new instances of the selected referencing elements will be created to match.

For more information on this option, including an example, see “XML ID attributes” on page 521.

### **Reference to ID**

The attribute is a reference to the ID of another element. If the element is specified as repeatable (that is, the user can add multiple instances of this element), then you can specify how to resolve the reference. You have two options:

- **Point to existing target**  
The new instance will reference the same ID as the original.
- **Create new target**  
The new instance will point to a new target that is created to match.

For more information on these options, including examples, see “XML references” on page 522.

If you select **Create new target**, you can customize the way in which the target element is created:

- **Select Owner**  
By default, the new target element is created when the current (referencing) element is created. This is appropriate for simple cases, when the current element is created directly. However, if the current element is created indirectly (for example, as part of the content of a higher-level element, or as part of a chain of references), then you should specify the directly created element as the owner of the reference. This ensures that the reference target will be created when the current element is created indirectly.
- **Select Target**  
By default, the owner of the referenced ID is selected as the target element. This is appropriate for simple cases, when the referenced element has no other dependencies. However, if the referenced element has a context that needs to be created with it (for example, a higher-level element that always contains just one instance of the referenced element), then you should select the appropriate target (for example, the higher-level or parent element).

For more information on customizing target creation, including examples, see “XML references with customized targets” on page 525.

### **RELATED CONCEPTS**

“XML wizards” on page 503

- “Attribute identity in XML” on page 520
- “XML ID attributes” on page 521
- “XML references” on page 522
- “XML references with customized targets” on page 525

#### **RELATED TASKS**

- “Creating an XML wizard” on page 504
- “Defining XML wizard macros” on page 507
- “Constraining values in an XML wizard” on page 515

## **Constraints**

When you create a wizard, or SmartGuide, with the SmartGuide Customizer for XML, you set which elements will be editable in the wizard. These elements are exposed in the wizard as fields, in which the wizard user can enter values.

When you set the element as editable, you can also set constraints that will be applied to the field, to limit the user to certain value types or formats.

You can set one of the following constraints:

- **NoSpace**  
No spaces are allowed in the value.
- **C++**  
The value must be a valid C++ type.
- **CORBA**  
The value must be a valid CORBA type.
- **SQL**  
The value must be a valid SQL type.
- **LongFile**  
The value must be a valid file name, for a system that supports long file names.
- **File83**  
The value must be a valid file name, for systems that have a maximum file name length of 8, with a maximum file extension length of 3.
- **File8**  
The value must be a valid file name, for systems that have a maximum file name length of 8, and the value must not include a file extension.
- **Any**  
There are no constraints on the value the user enters.
- **Action**  
Replace the selection with the name of a Java class that provides a constraint you defined. The class must implement the Constraint interface, provided with the SmartGuide Customizer for XML. When the XML wizard

runs, it will look for a Java class with that name, and call its test() function with the value the user entered as a parameter.

**RELATED CONCEPTS**

“XML wizards” on page 503

**RELATED TASKS**

“Creating an XML wizard” on page 504

“Defining XML wizard macros” on page 507

“Constraining values in an XML wizard” on page 515

---

## Filters

You can use filters in Object Builder to exclude information from the Tasks and Objects pane. You can use the filters provided with Object Builder, or define your own.

Object Builder provides the following filters or views:

- **Business Objects View**

Displays the User-Defined Business Objects folder, the Non-IDL Types folder, and the Build Configuration folder. Use this view if you are creating or working with components for new data, and do not need to work with data objects separately or import existing DB or PA schemas.

- **Data Objects View**

Displays the User-Defined Data Objects folder, the Non-IDL Types folder, and the Build Configuration folder. Use this view if you are working primarily with data objects, and do not need to create business objects or import existing DB or PA schemas.

- **Schema View**

Displays the DBA-Defined Schemas folder, the Non-IDL Types folder, and the Build Configuration folder. Use this view if you are working primarily with existing DB schemas, and do not need to work with data objects separately or create business objects.

The views hide other information, but do not prevent reference to it. For example, while using the data objects view, you can still define a data object attribute with the type of a hidden business object interface.

You can customize these views, or add new ones. For example, you could create a new view for working primarily with PA schemas, or for packaging (showing only the Build Configuration and Application Configuration folders).

#### RELATED CONCEPTS

“Object Builder” on page 1  
Component assembly (*Programming Guide*)

#### RELATED TASKS

“Filtering the Tasks and Objects pane”  
“Creating a filter for the Tasks and Objects pane”  
“Searching the Tasks and Objects pane” on page 30

## Filtering the Tasks and Objects pane

You can apply a filter to the Tasks and Objects pane in Object Builder, to show only the tasks that you are doing or the objects that you are using.

To apply a filter, follow these steps:

1. From Object Builder’s menu bar, select **View > Set Filter**.
2. Select a filter from the cascade of available views. Your selection is indicated with a checkmark.
3. Select **View > Turn Filter On**.

The filter you selected is applied to the Tasks and Objects pane. You can switch filters at any time, and turn the currently selected filter off and on.

You can customize the existing filters, or add new filters as required.

While a filter is in effect, some menu items may be unavailable. This product’s task documentation assumes an unfiltered view.

#### RELATED CONCEPTS

“Object Builder” on page 1  
“Filters” on page 536

#### RELATED TASKS

“Creating a filter for the Tasks and Objects pane”

## Creating a filter for the Tasks and Objects pane

You can create your own filter for the Tasks and Objects pane, to hide folders or objects that you are not using. Once you create a filter, it is available from the **View > Set Filter** menu, and you can turn it on or off with the **View > Turn Filter On/Off** menu choice.

To create a filter, follow these steps:

1. From Object Builder’s menu bar, select **View > Set Filter > Create New**.  
The Filter Tree window opens.

2. Select one of the existing filters from the filter list, to use as a starting point.
3. Select the folders and objects you want included in the new view. Folders or objects that are **not** selected will be excluded by the filter.
4. Click **Save As**. The Save Filter Scheme window opens.
5. Name the filter.
6. Click **OK**.
7. Click **Finish**.

The new filter now appears in the **View > Set Filter** menu, and is automatically selected and applied.

#### **RELATED CONCEPTS**

“Object Builder” on page 1

“Filters” on page 536

#### **RELATED TASKS**

“Filtering the Tasks and Objects pane” on page 537

---


## **XML browsing with XSL**

When you export an XML file from Object Builder, you can open and read the file in any text editor. However, the XML format is packed with information and can be overwhelming. You can make the XML files generated by Object Builder more readable by applying XSL style sheets to transform them into HTML files, which you can view in a browser.

An XSL style sheet can:


- Filter or reorder an XML document (arranging content)
- Apply fonts and HTML markup to an XML document (formatting content)

A sample XSL style sheet is included with Object Builder. It takes an XML file (model.xml) as input, filters out everything but business object and data object interface information (arranges content), and creates HTML files (formats content).

 In order to use XSL style sheets, you will need an XSL processor. Some web browsers have their own XSL processing capability, but if you do not want to change web browsers, or you want to make the results available to team members with other web browsers, then you can use a stand-alone XSL processor such as the LotusXSL Processor available from IBM's alphaWorks site (<http://www.alphaworks.ibm.com/tech/LotusXSL>). The



LotusXSL processor applies XSL style sheets to XML files and produces HTML files which can be viewed by any web browser.

 The documentation for XSL browsing applies primarily to Windows NT. The samples are available on AIX, but you will need to do your own investigation and setup for processing XSL.

The sample XSL style sheet provided by Object Builder creates HTML files that require a frames-capable browser. When you create your own style sheet, you can avoid frames if you prefer.

For more information about XSL and style sheets:

- The XSL specification: <http://www.w3.org/TR/WD-xsl/>
- XSL resources (links to tutorials, browser and tool resources)  
<http://www.w3.org/Style/XSL/>
- alphaWorks site (XML developer resources):  
<http://www.alphaworks.ibm.com/>

#### **RELATED CONCEPTS**

“The XSL sample” on page 541

#### **RELATED TASKS**


“Browsing XML files”

## **Browsing XML files**

You can use XSL style sheets to filter and transform complex XML files into simple HTML documents. You can use the XSL sample provided with Object Builder as a starting point, and then define your own XSL style sheet for your own browsing needs.

The following tasks describe the steps involved in using the XSL sample and creating your own XSL style sheet:

1. “Setting up for XSL” on page 540
2. “Viewing a sample XSL-based document” on page 543
3. “Applying the sample XSL style sheet” on page 544
4. “Creating your own XSL style sheet” on page 546
5. “Applying an XSL style sheet” on page 547

 The documentation for XSL browsing applies primarily to Windows NT. The samples are still available on AIX, but you will need to your own investigation and setup for processing XSL.

#### RELATED CONCEPTS

“XML browsing with XSL” on page 538  
“The XSL sample” on page 541

#### RELATED TASKS

“Chapter 11. Customizing Object Builder” on page 503

## Setting up for XSL

### ► CHANGED

The XSL process documented for the Object Builder XML format assumes that you are using IBM’s XML for Java, and the LotusXSL processor. These are both installed along with Object Builder.

In order to use the LotusXSL processor, you must install JDK 1.2 or JRE 1.2, or a later compatible version. This is available from <http://java.sun.com/products/jdk/1.2/index.html>

**Note:**The LotusXSL Processor is not dependent on a certain version of JDK as long as it is not earlier than version 1.2.

You must also set up your class path to include the JAR files provided by the LotusXSL package ( v2.0.15 of xml4j.jar, lotusxsl.jar, compat.jar and xalan.jar ).

**Note the following points about the LotusXSL JAR files:**

- These JAR files are installed along with Object Builder, and are located in the <CB installation directory>\lib.
- xml4j.jar is renamed to be ivbxml.jar.
- lotusXSL.jar must be placed before xalan.jar in the class path.

For example, if you have installed Component Broker to e:\CBroker, then modify your class path as follows:

```
CLASSPATH=e:\CBroker\lib\ivbxml.jar;e:\CBroker\lib\lotusxsl.jar;
e:\CBroker\lib\compat.jar;e:\CBroker\lib\xalan.jar;%CLASSPATH%
```

### ► AIX

The documentation for XSL browsing applies primarily to Windows NT. The samples are still available on AIX, but you will need to your own investigation and setup for processing XSL.

#### RELATED CONCEPTS

“XML browsing with XSL” on page 538

#### RELATED TASKS

“Browsing XML files” on page 539

## The XSL sample

►CHANGED

The XSL sample demonstrates how an XSL style sheet can be used to browse XML files exported from Object Builder. The style sheet sample is located in:

<CB installation directory>\xsl

The sample consists of style sheets, a frameset file to present the output of the style sheets, batch files to simplify the process of applying the style sheets, and sample output in the xsl\AccountFileSample directory, based on a sample XML file.

### Style sheets

This sample uses the following style sheets:

#### **tree1.xsl, tree2.xsl**

Each creates an HTML table of contents for the source XML file (for example, model.xml). Indexes business object interfaces (files, modules, interfaces), data object interfaces (files, modules, interfaces) and relationships (1-n). The table of contents links to anchors in another HTML document, created by the second XSL file.

#### **view.xsl**

Creates an HTML page that documents business object interfaces and data object interfaces. Each interface is documented in both a summary table format (listing its members) and a detailed format (providing a full signature and details for each member).

When you apply the style sheets one pair at a time (tree1.xsl with view.xsl using render1.bat; and tree2.xsl with view.xsl using render2.bat) to the source XML file (for example, model.xml), it creates two HTML files based on the XML (the table of contents, and the documentation file), which you can view using a third HTML file that defines a frameset for the two files (so they can be read side-by-side).

### Frameset file

The frameset HTML file (frames.html) is in the same directory as the XSL files. You can use the frameset file to view HTML files you create with the sample style sheets.

The stylesheets were developed with LotusXSL 1.0.0 and are based on the November 1999 Recommendation.

### Batch file

The batch files (render1.bat and render2.bat) are in the same directory as the XSL files. You can use the batch file to apply the pair of style sheets to an XML file in a single step.

## HTML files

Sample HTML files created by the style sheets are in <CB installation directory>\xsl\AccountFileSample:

### AccountFile\_tree.html

The output from tree.xsl, which provides links to AccountFile\_view.html within a frameset

### AccountFile\_view.html

The output from view.xsl

### AccountFile\_frames.html

A modified version of frame.html, which points to the other two files.

## Directory structure

The sample is now consolidated under one directory (xsl\AccountFileSample). model.xml, which contains the exported XML for the model, exists in this directory. This directory has two subdirectories view\_with\_tree1 and view\_with\_tree2, containing the HTML output rendered by tree1.xsl and tree2.xsl respectively. These two subdirectories each contain the AccountFile\_frames.html file, which is a generic HTML file with a frameset for two frames that correctly load AccountFile\_tree.html on the left, and AccountFile\_view.html on the right.

This is the directory structure:

```
xsl\tree1.xsl
xsl\tree2.xsl
xsl\view.xsl
xsl\frames.xsl
xsl\render1.bat
xsl\render2.bat
xsl\AccountFileSample\model.xml
xsl\AccountFileSample\view_with_tree1\AccountFile_frames.html
xsl\AccountFileSample\view_with_tree1\AccountFile_tree.html
xsl\AccountFileSample\view_with_tree1\AccountFile_view.html
xsl\AccountFileSample\view_with_tree2\AccountFile_frames.html
xsl\AccountFileSample\view_with_tree2\AccountFile_tree.html
xsl\AccountFileSample\view_with_tree2\AccountFile_view.html
```

The XSL style sheets were run against the XML file for the Composite Business Object sample to produce two files: tree.html and view.html. These files were renamed and their linking modified to use the names AccountFile\_tree.html and AccountFile\_view.html.

The two files are viewable through a third file, **AccountFile\_frames.html**, in the same directory. This is a modified version of the basic frameset file which points to the renamed tree and view html files, and organizes the two files into a two-frame layout.

### XML file

The HTML files provided in the sample are based on a file model.xml, which was created by exporting the model for the Composite Business Object sample project. The sample model is in the same directory as its associated sample output: <CB installation directory>\xsl\AccountFileSample\

#### RELATED CONCEPTS

“XML browsing with XSL” on page 538

#### RELATED TASKS

“Browsing XML files” on page 539

“Viewing a sample XSL-based document”

“Applying the sample XSL style sheet” on page 544

## Viewing a sample XSL-based document

### ► CHANGED

You can make the XML files generated by Object Builder more readable by applying XSL style sheets to transform them into HTML files, which you can view in a browser. The XSL sample provided with Object Builder filters XML files and produces HTML files that include a subset of the original information, organized for readability. The provided sample includes some sample output. To view the sample output, follow these steps:

1. Start your web browser.

You must use a frames-compatible browser.

2. Browse the following file:

<CB installation  
directory>\xsl\AccountFileSample\AccountFile\_frames.html  
**Note:**The *AccountFileSample* is a model in Component Broker’s  
Tutorial\CBOB\Accounts directory.

If you know the path to the file, you can type it in the URL field in your browser. Otherwise, use the **File** menu to locate and open the file.

The file AccountFile\_frames.html organizes two other files into a frameset: AccountFile\_tree.html, on the left, displays a table of contents for AccountFile\_view.html, on the right. Click on the entries in the table of contents (on the left) to jump to a particular topic in the main view (on the right), or scroll through the view.

#### RELATED CONCEPTS

“XML browsing with XSL” on page 538  
“The XSL sample” on page 541

#### RELATED TASKS

“Browsing XML files” on page 539

## Applying the sample XSL style sheet



You can apply the XSL sample style sheet to your own XML files to produce a simple set of documentation for your project. For more complete documentation, or customized documentation for your own or your team members' needs, you should create your own XSL style sheet.

To generate an XML file for use with the sample style sheet:

1. Open a project that contains business objects and data objects.
2. Click **File > Export Model** to open the Export Model wizard.
3. Check the **Export the model in one file** option.
4. Click **Finish**.

The XML file for the model is placed in the current project's Working\Export directory.

You are provided with two sample style sheets in the xsl subdirectory within Component Broker's install directory: tree1.xsl and tree2.xsl.

The new stylesheet, tree2.xsl renders a different view from the one tree1.xsl does. This new view presents a navigation pane that is based on the file hierarchy of the model. This resembles to a great extent the way Object Builder organizes its components, rather than just grouping the objects by their type, as in version 3.0.

view.xsl is another sample stylesheet that renders the right-hand content pane. The new sample does not include a stylesheet for rendering the content pane. So, it uses the same file view.xsl as does the first sample.

The sample is now consolidated under one directory (xsl\AccountFileSample). model.xml, which contains the exported XML for the model exists in this directory. This directory has two subdirectories view\_with\_tree1 and view\_with\_tree2, containing the HTML output rendered by tree1.xsl and tree2.xsl respectively. These two subdirectories each contain the AccountFile\_frames.html file, which is a generic HTML file with a frameset for two frames that correctly load AccountFile\_tree.html on the left, and AccountFile\_view.html on the right.

This is the directory structure:

```
xsl\tree1.xsl
xsl\tree2.xsl
xsl\view.xsl
xsl\frames.xsl
xsl\render1.bat
xsl\render2.bat
xsl\AccountFileSample\model.xml
xsl\AccountFileSample\view_with_tree1\AccountFile_frames.html
xsl\AccountFileSample\view_with_tree1\AccountFile_tree.html
xsl\AccountFileSample\view_with_tree1\AccountFile_view.html
xsl\AccountFileSample\view_with_tree2\AccountFile_frames.html
xsl\AccountFileSample\view_with_tree2\AccountFile_tree.html
xsl\AccountFileSample\view_with_tree2\AccountFile_view.html
```

To apply the tree1.xsl sample style sheet to your XML file, use the batch file (render1.bat) that comes with the sample:

```
render1 model
```

The render1.bat batch file contains the following commands:

```
java com.lotus.xsl.Process -PARSER
com.lotus.xml.xml4j2dom.XML4JLiaison4dom -in %1.xml -xsl tree1.xsl -out
tree.html - VALIDATE
```

```
java com.lotus.xsl.Process -PARSER
com.lotus.xml.xml4j2dom.XML4JLiaison4dom -in %1.xml -xsl view.xsl -out
view.html - VALIDATE
```

To apply the tree2.xsl sample style sheet to your XML file, use the batch file (render2.bat) that comes with the sample:

```
render2 model
```

The render2.bat batch file contains the following commands:

```
java com.lotus.xsl.Process -PARSER
com.lotus.xml.xml4j2dom.XML4JLiaison4dom -in %1.xml -xsl tree2.xsl -out
tree.html - VALIDATE
```

```
java com.lotus.xsl.Process -PARSER
com.lotus.xml.xml4j2dom.XML4JLiaison4dom -in %1.xml -xsl view.xsl -out
view.html - VALIDATE
```

Object Builder's XML DTD (eom.dtd) must be in the same directory as your input file. (eom.dtd is shipped with the install.)

The sample style sheet assumes that your input XML file contains definitions of business object interfaces and data object interfaces, for example the model.xml file created when you export XML for an entire project. The sample style sheet is not appropriate for XML files that define other elements (for example, a udcontainer.ContainerOfAgents.xml file that defines a container).

**▶ AIX** The documentation for XSL browsing applies primarily to Windows NT. The samples are still available on AIX, but you will need to your own investigation and setup for processing XSL.

#### **RELATED CONCEPTS**

“XML browsing with XSL” on page 538

“The XSL sample” on page 541

#### **RELATED TASKS**

“Browsing XML files” on page 539

## **Creating your own XSL style sheet**

### **▶ CHANGED**

You can create your own XSL style sheet for viewing XML files. You can use the existing XSL sample file tree.xml as a starting point. It includes a number of comments to help you understand and extend it for your needs. Once you are familiar with tree.xml, see view.xml for a more advanced example of XSL usage.

The source XML format uses xmi.id and xmi.idref attributes to identify and reference elements. They can be used to create hypertext links in the output HTML, as demonstrated in both tree.xml and view.xml.

Limitations of the sample style sheet:

- The example style sheet does not resolve links between models; any unresolved links are formatted in red in the HTML output.
- It supports only basic members of typedef, union, struct, and exception.
- It does not support nested modules.

While XSL is still an emerging standard, there are already a number of resources to help with the task of writing XSL style sheets. Many of them are organized and linked to from: <http://www.w3.org/Style/XSL/>

#### **Note the following points when you create your own XSL stylesheets:**

- When an xsl:attribute contains a text node with a newline, the XML output must contain a character reference. If it does not, a character reference, which for version 1.0 of LotusXSL is a sequence of '%20's, is placed between the two lines. To prevent this, you must explicitly create an xsl:text node. An example of this can be found in line 575 and 607 of view.xml. For



more information refer to the XSLT 1.0 W3C Recommendation of 16 November 1999, Section 7.1.3 Creating Attributes with `xsl:attribute`.

- You may find it easier to examine the HTML output from `view.xml` by turning on the indent function in the `xsl:output` directive. To do this, change the attribute `indent` to "yes". (`indent="yes"`). The generated HTML will then be indented and neatly organized. This function had been turned off in `view.xml` because automatic indenting puts some relative positioning out of place: for example, the arrow in the Relationship Details section for each business object interface. If indentation is turned on, the arrows will not align.

#### RELATED CONCEPTS

"XML browsing with XSL" on page 538

"The XSL sample" on page 541

#### RELATED TASKS

"Browsing XML files" on page 539

"Applying an XSL style sheet"

## Applying an XSL style sheet

▶CHANGED

Once you have written an XSL style sheet, you can apply it to XML files exported from Object Builder.

To apply an XSL style sheet to your XML file, use the Java class that is named `Process`:

```
java com.lotus.xml.Process -PARSER
com.lotus.xml.xml4j2dom.XML4JLiaison4dom -in myfile.xml -xsl
mystylesheet.xml -out myoutput.html - VALIDATE
```

For example, as in the XSL sample, `myfile` can be `model`, `mystylesheet` can be `tree1`, `myoutput` can be `tree`.

Object Builder's XML DTD (`eom.dtd`) must be in the same directory as your input file.

▶AIX

The documentation for XSL browsing applies primarily to Windows NT. The samples are still available on AIX, but you will need to your own investigation and setup for processing XSL.

#### RELATED CONCEPTS

"XML browsing with XSL" on page 538

The XSL sample

**RELATED TASKS**

“Browsing XML files” on page 539

“Creating your own XSL style sheet” on page 546

---

## Chapter 12. Configuration

---

### Configuring builds

Once you have defined your components in Object Builder, you are ready to build the components into client and server dynamic link libraries (DLLs, also known as shared library files). The client DLLs contain the component interfaces, and helper classes, which allow your client applications to locate and use the components on the server. The server DLLs contain the implementations and data objects for the component.

To configure and build your DLLs, complete the following steps:

1. Specifying a build location
2. "Generating code" on page 551
3. "Defining a client DLL" on page 552
4. "Defining a server DLL" on page 554
5. "Generating a makefile" on page 556
6. "Building the DLLs" on page 558
7. "Building the JAR files" on page 561
8. "Building for ActiveX clients" on page 562
9. "Building for Java clients" on page 563
10. "Building for QuickTest" on page 564
11. "Rebuilding DLLs" on page 565
12. "Launching a remote OS/390 build" on page 572

For an example of how to set up a remote OS/390 build, see "Tutorial: Launching a remote OS/390 build" on page 573.

Once you have built the DLLs, you can debug them, or package them as part of an application.

#### **RELATED TASKS**

- "Developing in Object Builder" on page 19
- "Building DLLs in a team environment" on page 487
- "Packaging applications" on page 574
- "Chapter 13. Testing applications with QuickTest" on page 611

#### RELATED REFERENCES

“Internationalization of data” on page 132

“Naming objects” on page 128

## Specifying a build location

NEW

Before you generate code, you can specify the absolute path for the working directory, for each of the build platforms. It is this path that is emitted to the generated make files. You can then move the generated code to another location (for example, another machine or another platform, by copying the entire working directory), without having to modify the paths in the generated files.

To specify a build location, follow these steps:

1. Select the Build Configuration folder in the Tasks and Objects pane.
2. From its pop-up menu, select **Properties**. The Build Configuration wizard opens to the Contents Ordering page.
3. Click **Next**. The Build Location page opens.
4. For each of the platforms, you can type in the absolute path.
5. Click **Finish**.

The paths that you specify will be used in the files that are generated to the target build platforms.

#### Note the following points:

- If you do not specify an absolute path for the working directory, Object Builder emits the project paths of the models into `prjdefs.mk`, `qt.bat`, `QTjar.txt`, and the DDL files. If you specify a path, Object Builder emits the contents of the entry field instead.
- The Build Location page allows you to specify absolute paths only for the currently loaded model. You cannot specify your own paths for dependent models. You must do this by opening Object Builder on each of the dependent models directly.

#### RELATED CONCEPTS

“Cross-platform development” on page 426

#### RELATED TASKS

“Configuring builds” on page 549

“Generating code” on page 551

“Setting up a team development environment” on page 467

#### RELATED REFERENCES

“Build targets” on page 567

## Generating code

Before building an application, you must generate source code for the objects you have created. By selecting **Generate > Selected** or **Generate > All** from an object's pop-up menu, you can generate code for that object only, or for that object *and* all objects below it in the tree. You can also generate code for a project from a command line, using the obgen command.

Until you generate code, all information for your objects is maintained in an Object Builder model (for example, MyProject/Model/\*.uni). When you generate, the resulting files are placed in the /Working/platform subdirectory of the project directory you specified, ready to be compiled (for example, MyProject\Working\NT\*.idl, \*.cpp, \*.java). Java versions of the key and copy helper, for use by Java client applications, are generated into subdirectories with names based on the module names of the key or copy helper (for example, MyProject\Working\NT\ClaimModuleCopy\ClaimCopyHelper.java).

Before you generate code, you can specify the absolute path for the working directory, for each of the build platforms. See the task Specifying a build location.

You can select which platforms you generate code for using the **Platforms > Generate** menu on the Object Builder main menu bar. You can also select which platform to view information for, and constrain your development options to a particular set of platforms. You can only view one platform at a time, but you can generate code for multiple platforms at a time.

You can generate source code for any object in the **User-Defined Business Objects** folder, **User-Defined Data Objects** folder, **DBA-Defined Schemas** folder, and **User-Defined Compositions** folder. To generate code for an object, follow these steps:

1. Select an object.
2. From the object's pop-up menu, click **Generate > Selected > All Files**. The appropriate code for the object is generated into the working directory. You can also select to generate only a particular type of code, from the **Selected** choices. These choices display the list of file types that can be generated for the selected object (for example, .ih, .cpp, .java)

**Note:** Because a business object interface is physically contained in a business object file, you generate the code for the interface by generating the code for the file (from the business object file's pop-up menu, click **Generate > Selected**). The same applies to data object interfaces in the User-Defined Data Objects folder.

You can generate the code for all the objects in a folder by selecting **Generate > All** from the folder's pop-up menu.

The generation process is tracked by a progress indicator, and may take some time. The more platforms you are generating code for, the longer the generation process will take.

You may also generate code from the command line, using the `obgen` command.

To view the source code for any of the objects you defined, select **View Source** from the object's pop-up menu. The `.idl`, `.ih`, and `.cpp` or `.java` files for the object are loaded in the Source pane. Click the drop-down arrow on the right end of the editor pane's title bar to access a list of currently loaded files and switch between them. You cannot edit the source code directly: if you want to change the source code, do so by changing the selections in the wizards, or editing the code associated with your methods in the Methods pane. The next time you generate the source code, your changes are applied.

**Note:** Outside of Object Builder, you can edit the source code with the editor of your choice. Changes to method bodies should be imported back into Object Builder, or your changes will be over-written the next time code is generated.

You can now generate the makefiles that will set your build options and define your target DLLs.

#### **RELATED CONCEPTS**

"Multi-platform development" on page 419

#### **RELATED TASKS**

"Importing edited source files" on page 385

"Generating a makefile" on page 556

"Building the DLLs" on page 558

"Generating code from the command line" on page 684

#### **RELATED REFERENCES**

"Objects to source files mapping" on page 258

"obgen" on page 685

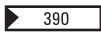
## **Defining a client DLL**



Your application will typically consist of both client and server shared libraries, or dynamic link libraries (DLLs). To define a client DLL, follow these steps:

1. Under Tasks and Objects, select the **Build Configuration** folder.
2. From its pop-up menu, select **Add Client DLL**. The Client DLL wizard opens to the Name and Options page.
3. Type a name for the dynamic link library or shared library file, without the file extension. If you want, you can also type a description of the configuration.
4. Set the platforms for which you want to build DLLs (**Deployment Platforms**).
5. Set the options for each platform:
  - a. Select a platform from the **Deployment Platforms** list. All the options you enter below will apply to the DLL built for this platform.
  - b. Type a name for the library (DLL), without the file extension.

**Note the following points:**

- You cannot have spaces in the DLL file name. When you click **Finish** to close the wizard, the program strips out any spaces. It also removes the file extension, if you happened to include it.
  - If you do not specify a file name, the name of the configuration will be used (with the extension .dll).
  -  The file name cannot exceed eight characters.
- c. In the **Make Options** field, type any options you want to call the DLL's makefile with. The options are added to the **all.mak** file that calls the DLL makefile.

There are several options specific to Component Broker that you can enter in this field. These options supplement or override the selection you make for your "Default Configuration" on page 566. They do not affect the output directory set by the default configuration.

- IVB\_TRACE=1 (page 691)(not appropriate for a client DLL)
- IVB\_TRACE\_DEBUG=1 (page 691) (not appropriate for a client DLL)
- IVB\_UNOPTIMIZE=1 (page 692)
- IVB\_DYNAMIC\_LINK=1 (page 690)
- IVB\_BUILD\_VERBOSE=1 (page 689)
- IVB\_OPTIMIZE=1 (page 690)
- activex (page 692)
- all (page 692)
- cleandll (page 693)
- cpp (page 693)
- java (page 693)
- jcb (page 693)

- quicktest (page 693)
- d. In the **IDL Compile Options**, **IOM Java Compile Options**, **JCB Java Compile Options**, and **CPP Compile Options** fields, specify any options you want passed to the IDL, Java, and C++ compilers by the makefile (in addition to those set by any macros specified in the “Default Configuration” on page 566 or in the **Make Options** field).
  - e. In the **Link Options** field, specify any linker options you want to build the DLL with. Also enter any non-Object Builder user-defined libraries for any DLLs that are referenced by this DLL.
6. Click **Next**. The Libraries to Link With page opens.
  7. Select the names of the import libraries for any other DLLs you have defined in Object Builder that are referenced by this DLL.  
For example, if this DLL contains a child interface whose parent is defined in another DLL, you need to select the import library for the parent’s DLL here.
  8. Click **Next**. The Client Source Files page opens.
  9. Select the files you want to use as source for the DLL. Only files that are candidates for a client DLL (for example, key and copy interfaces) are available for selection.  
If you are building a composition or a composite component, you need to include the client interface files of the member components in the composition (business object file, key file, and copy helper file).
  10. Click **Finish**. The client DLL object appears in the Build Configuration folder and you are ready to generate the makefile that will build it.

#### RELATED TASKS

- “Configuring builds” on page 549
- “Defining a server DLL”
- “Generating a makefile” on page 556

## Defining a server DLL

Your application will typically consist of both client and server shared libraries, or dynamic link libraries (DLLs). To define a server DLL, follow these steps:


1. Under Tasks and Objects, select the **Build Configuration** folder.
2. From the pop-up menu of the folder, select **Add Server DLL**. The Server DLL wizard opens to the Name and Options page.
3. Type a name for the configuration. This is a unique identifier for the build configuration that creates the DLL. If you want, you can also type a description of the configuration.
4. Set the platforms for which you want to build DLLs (**Deployment Platforms**).



5. Set the options for each platform:
  - a. Select a platform from the **Deployment Platforms** list. All the options you enter below will apply to the DLL built for this platform.
  - b. Type a name for the library (DLL), without the file extension.

**Notes:**

- You cannot have spaces in the DLL file name. When you click **Finish** to close the wizard, the program strips out any spaces. It also removes the file extension, if you happened to include it.
- If you do not specify a file name, the name of the configuration will be used (with a .dll extension).

-  The file name cannot exceed 8 characters.

- c. In the **Make Options** field, type any options you want to call the DLL's makefile with. The options are added to the all.mak file that calls the DLL makefiles.

There are several options specific to Component Broker that you can enter in this field. These options supplement, or override, the selection you make for your "Default Configuration" on page 566. They do not affect the output directory set by the default configuration.

- IVB\_TRACE=1 (page 691)
- IVB\_TRACE\_DEBUG=1 (page 691)
- IVB\_UNOPTIMIZE=1 (page 692)
- IVB\_DYNAMIC\_LINK=1 (page 690)
- IVB\_BUILD\_VERBOSE=1 (page 689)
- OPTIMIZE=1
- activex (page 692)
- all (page 692)
- cleandll (page 693)
- cpp (page 693)
- java (page 693)
- jcb (page 693)
- quicktest (page 693)

- d. In the **IDL Compile Options**, **IOM Java Compile Options**, **JCB Java Compile Options**, and **CPP Compile Options** fields, specify any options you want passed to the IDL, Java, and C++ compilers by the makefile.
- e. In the **Link Options** field, specify any linker options you want to build the DLL with. Also enter any non-Object Builder user-defined libraries for any DLLs that are referenced by this DLL.

6. Click **Next**. The Libraries to Link With page opens.
7. Select the name of the import library (.lib file) for the corresponding client DLL. Also select the names of the import libraries for any other DLLs you have defined in Object Builder that are referenced by this DLL.  
For example, if this DLL contains a child interface whose parent is defined in another DLL, you need to select the import library for the parent's DLL here.
8. Click **Next**. The Server Source Files page opens.
9. Select the IDL files you want to use as source for the DLL. Only files that are candidates for a server DLL (for example, business object implementations and managed objects) are available for selection.  
When you select the source file for a data object implementation, the source files for its associated persistent objects are automatically included.
10. Click **Finish**. The server DLL object appears in the Build Configuration folder. You are now ready to generate the makefile that will build it.

#### RELATED TASKS

- “Configuring builds” on page 549
- “Defining a client DLL” on page 552
- “Generating a makefile”

## Generating a makefile

To generate the makefiles that will build the shared libraries or dynamic link libraries (DLLs) in the Build Configuration folder, follow these steps:

1. Select the Build Configuration folder.
2. From the folder's pop-up menu, select one of the options under **Generate > All** (as described below). The makefiles (all.mak, local.mak, and the makefiles for the DLLs in the folder) are generated into your working directory.

This builds the targets that are configured in the build configuration for the current project. Once the makefiles have been generated, you can view them by clicking **View Source** from the folder's pop-up menu.

When you generate from the folder's pop-up menu, the option you select under **Generate > All** determines what is included in the makefiles as their default targets. You can select from the following options:

- **C++ Default Targets**  
The makefile will build all C++ DLLs.
- **Java Default Targets**  
The makefile will build all Java JAR files.

- **Java Client Bindings Default Targets**

The makefile will build all Java client bindings.

▶ 390 Not available on OS/390.

- ▶ WIN **ActiveX Interfaces Default Targets**

The makefile will build all ActiveX client DLLs.

- ▶ WIN ▶ AIX ▶ SOLARIS ▶ HP-UX **QuickTest Default Targets**

Compiles QuickTest client files, in the

Working\platform\config\QTCLS\DLLname\ directory.

Builds the QuickTest client .jar file in the

Working\platform\config\QT\QTDLLname.jar.

Places the qt QuickTest script file that is used to run the QuickTest client application, in Working\platform\config\

- **All Targets**

The makefile will build all DLLs and associated objects available for a particular platform, except for ActiveX and QuickTest, which are excluded by default. You can include them with selections in the Preferences notebook, on the Makefile Generation page.

For more details see “Build targets” on page 567.

The menu item you select determines what the **Build > Default Targets** action will build. The makefile you generate can still be used to build other targets, through the folder pop-up menu’s **Build** actions.

The makefile for each DLL includes any IDL compile, Java compile, CPP compile, and link options you specified for the DLL. Do *not* use these files directly. Use the all.mak file or local.mak file, which call the makefiles for each DLL, and include any make options you specified for each DLL. The all.mak file builds all targets; local.mak builds only the targets configured in the build configuration for the current project. Using these .mak files ensures that the DLLs are built in the correct order.

**Note the following points:**

- If an interface defined using an Object Builder wizard or imported from an .idl file ‘includes’ other interfaces, the ‘included’ interface or header files *does not appear in the makefile as dependencies* of the ‘including’ interface. Prior to rebuilding the generated source, you must either manually edit the makefile to add the missing dependencies, or clean and rebuild all targets.
- You must build the DLLs on a server development machine (typically, the one on which you are using Object Builder). If you move the makefiles to another machine without the server SDK installed, the DLLs may not compile.

#### RELATED TASKS

- “Configuring builds” on page 549
- “Defining a client DLL” on page 552
- “Defining a server DLL” on page 554

## Specifying the order of a build



You can design your build system in such a way that the projects get built in an order, which is compatible with the logical layering of the system.

When you work with multiple Object Builder subprojects, some lower-level projects that provide services for higher-level projects must be built first (before the higher-level ones). This is especially important if you have modules within your projects.

To specify the order of a build, follow these steps:

1. From the pop-up menu of the Build Configuration folder, select **Properties**. The Build Configuration wizard opens to the Contents Ordering page. The DLLs are listed in a tree view in the order in which they were added to the project. This tree view represents the order in which the DLLs will be built when you either select the **Build** action from within Object Builder, or use either `make` or `nmake` from the command line.
2. Select a DLL from the tree view. This action enables the **Order by Dependency** button, and you can use it to have Object Builder order the DLLs so that dependencies are resolved. The new order will closely resemble the old one.
3. When you have a DLL selected, you can also use the **Move Up** and **Move Down** buttons to manually change the order of the DLLs within the tree view.

#### RELATED CONCEPTS

- “Cross-project dependencies” on page 462

#### RELATED TASKS

- “Building the DLLs”/“Configuring builds” on page 549
- “Launching a remote OS/390 build” on page 572
- “Generating a makefile” on page 556
- “Packaging applications” on page 574

## Building the DLLs



Before packaging an application, you must build your client and server DLLs (that is, compile and link the generated code). You can do this by running either the **all.mak** makefile, or the **local.mak** makefile, which you generated. `local.mak` calls only the client and server makefiles in the current project, unlike `all.mak`, which calls the other DLL make files as well.

Do **not** run the makefiles for the individual DLLs directly. Using all.mak ensures that the DLLs are built in the correct order.

You must build the DLLs on a server development machine (typically, the one on which you are using Object Builder). If you move the makefiles to another machine without the server SDK installed, the DLLs may not compile.

▶ 390 If you are building for OS/390, you can use the OS/390 remote build process to build on a specified remote host.

To run all.mak, follow these steps:


1. Under Tasks and Objects, select the **Build Configuration** folder.
2. From the folder's pop-up menu, select **Build**, and then one of the following options:
  - **Out-of-Date Targets**  
You can select the type of out-of-date targets to build:
    - **C++**  
Builds C++ client and server DLLs.
    - **Java**  
Builds JAR files for Java business objects and for components with PA-based persistence.
    - **Java Client Bindings**  
Builds Java client bindings that allow a Java client application to access the equivalent components in the server application.
    - **Default**  
Builds whatever was selected when the makefile was generated (for example, if **Generate > All > C++ Default Targets** was used to generate the makefiles, then selecting **Build > Out-of-Date Targets > Default** will build the C++ targets).
  - **All Targets**  
Builds all targets.
  - **Rebuild All Targets**  
Performs a build clean, followed by a build all targets.
  - **Clean**  
Performs a build clean, but does not perform a build.
3. When the build has finished, you can review the record of the build in the command window.

You can also make all.mak from a command line, with the following flags:

- IVB\_TRACE=1 (page 691)
- IVB\_TRACE\_DEBUG=1 (page 691)
- IVB\_UNOPTIMIZE=1 (page 692) (or IVB\_OPTIMIZE=0)

- IVB\_DYNAMIC\_LINK=1 (page 690)
- IVB\_BUILD\_VERBOSE=1 (page 689)
- IVB\_OPTIMIZE=1 (or IVB\_UNOPTIMIZE=0)
- IVB\_COMBINE\_SOURCE=1 (page 692)
- activex (page 692)
- all (page 692)
- cleandll (page 693)
- cpp (page 693)
- java (page 693)
- jcb (page 693)
- quicktest (page 693)

If you make all.mak from the command line, you can override your “Default Configuration” on page 566 selection in Object Builder.

 **Note:** The options that enable you to build for the QuickTest target are not available both from the Object Builder interface, and the command line, if the view platform (**Platform > View**) is OS/390.

Once you have built all applicable targets, your DLLs and JAR files exist on your hard drive, in one of the following project working directories, as defined by your macro selection or by your default configuration setting.

Configuration setting	Directory
<b>Unoptimized</b> configuration (IVB_UNOPTIMIZE=1 (page 692))	Working\ <i>platform</i> \NOOPT
<b>Production</b> configuration (IVB_OPTIMIZE=1) (This is the default.)	Working\ <i>platform</i> \PRODUCTION
<b>Trace</b> configuration (IVB_TRACE=1 (page 691))	Working\ <i>platform</i> \TRACE
<b>Trace and debug</b> configuration (IVB_TRACE_DEBUG=1 (page 691))	Working\ <i>platform</i> \TRACE_DEBUG

For C++ components, the DLLs (*MyClientDLL.dll* and *MyServerDLL.dll*) are built with the file name that you specify, and are placed in the project working directory. If you have Java components in the same application, then you will need a JAR file for each C++ component that provides Java components on the server with access to the C++ components:  
*MyClientDLL.jar*.

If you are supporting a Java client application, then the Java client bindings file: *jcbMyClientDLL.jar* must also be built in the `working\platform\config\JCB\` directory.

The DLLs and JAR files are automatically pulled into an application when you configure the managed object with the application.

#### RELATED CONCEPTS

“Troubleshooting” on page 905

#### RELATED TASKS

“Configuring builds” on page 549

“Launching a remote OS/390 build” on page 572

“Generating a makefile” on page 556

“Packaging applications” on page 574

## Building the JAR files

If some of your components include Java business objects or use PA-based persistence, you need to build JAR files as well as DLLs. The process is equivalent to building DLLs, and the JAR files are created according to their definitions in the Client and Server DLL wizards.

To build Java targets:

1. Under Tasks and Objects, select the **Build Configuration** folder.
2. From the folder’s pop-up menu, select **Build > Out-of-Date Targets > Java**
3. When the build is finished, review the results of the build in the command window.

You can also build from the command line by making `all.mak` directly. If you build with the `java` (page 693) flag (`nmake all.mak java`), only Java targets will be built.

For Java components, three `.jar` files are created:

- *MyClientDLL.jar*  
Supports access to the component by other components on the server.
  - Source: `\Working\platform\`
  - Compiled classes: `\Working\platform\config\JAVACLS\MyClientDLL\`
  - JAR file location: `\Working\platform\config\`
- *MyServerDLL.jar*  
Supports and implements the Java business object on the server.
  - Source: `\Working\platform\`
  - Compiled classes: `\Working\platform\config\JAVACLS\MyServerDLL\`
  - JAR file location: `\Working\platform\config\`

- JCB\jcbMyClientDLL.jar  
Java client bindings that support access to the component by the client application.
  - Source: \Working\platform\JCB\
  - Compiled classes: \Working\platform\config\JCBCLS\MyClientDLL\
  - JAR file location: \Working\platform\config\JCB\

#### RELATED CONCEPTS

“Troubleshooting” on page 905

#### RELATED TASKS

“Configuring builds” on page 549

“Building the DLLs” on page 558

“Launching a remote OS/390 build” on page 572

“Generating a makefile” on page 556

“Packaging applications” on page 574

## Building for ActiveX clients

When you build ActiveX interfaces in Object Builder, a set of source files, and a makefile that you can use to build them into DLLs, are placed in the current project’s Working\NT\config\ACTIVEX directory. You still need to build the DLLs outside of Object Builder before you can use them with your client application.

To generate the interfaces that ActiveX clients can use to access your server components, follow these steps:

1. In Object Builder, click **File > Preferences** to open the Preferences notebook.
2. Click **Tasks and Objects > Makefile Generation** to turn to the Makefile Generation page.
3. Check the **Include ActiveX in “all” target** option
4. Click **OK** to apply your changes to Object Builder.

From now on, whenever you select to generate all targets or build all targets the result will include ActiveX client interfaces. You can also build specifically for ActiveX without changing this default, as described in the next few steps.

5. Under Tasks and Objects, select the Build Configuration folder.
6. From the folder’s pop-up menu, click **Generate -ActiveX Interfaces Default Targets**.
7. From the same pop-up menu, select **Build > Out-of-Date Targets > ActiveX Interfaces**



8. When the build is finished, review the results of the build in the command window.

You can also build from the command line by making `all.mak` directly. If you build with the `activex` (page 692) flag (`nmake all.mak activex`), only ActiveX interfaces will be generated.

The interfaces are generated into the current platform's `Working\NT\config\ACTIVEX` directory.

#### RELATED CONCEPTS

ActiveX client programming model (*Programming Guide*)

#### RELATED TASKS

"Configuring builds" on page 549

Developing the Component Broker ActiveX client (*Programming Guide*)

## Building for Java clients

To build interfaces that Java clients can use to access your server components, follow these steps:

1. Under Tasks and Objects, select the **Build Configuration** folder.
2. From the folder's pop-up menu, select **Build > Out-of-Date Targets > Java Client Bindings**
3. When the build is finished, review the results of the build in the command window.

You can also build from the command line by making `all.mak` directly. If you build with the `jcb` (page 693) flag (`nmake all.mak jcb`), only Java client bindings will be built.

The client bindings are placed in JAR files with names based on the names of your configured client DLLs in the Build Configuration folder.

The Java client bindings are built as follows:

- File: `jcbMyClientDLL.jar`
- Source: `\Working\platform\JCB\`
- Compiled classes: `\Working\platform\config\JCBCLS\MyClientDLL\`
- JAR file location: `\Working\platform\config\JCB\`

#### RELATED CONCEPTS

Java client programming model (*Programming Guide*)

#### RELATED TASKS

- “Configuring builds” on page 549
- “Defining a client DLL” on page 552
- “Generating a makefile” on page 556

## Building for QuickTest

To build a QuickTest client to test your application with, follow these steps:




1. In Object Builder, click **File > Preferences** to open the Preferences notebook.
2. Click **Tasks and Objects > Makefile Generation** to turn to the Makefile Generation page.
3. Check the **Add QuickTest target to 'all' target** option
4. Click **OK** to apply your changes to Object Builder.

From now on, whenever you select to generate all targets or build all targets the result will include a refresh of your QuickTest client. You can also build QuickTest specifically without changing this default, as described in the next few steps.

5. Under Tasks and Objects, select the **Build Configuration** folder.
6. From the folder's pop-up menu, select **Build > Out-of-Date Targets > QuickTest**
7. When the build is finished, review the results of the build in the command window.




You can also build from the command line by making all.mak directly. If you build with the quicktest (page 693) flag (nmake all.mak quicktest), only QuickTest files will be built.

The QuickTest files are built as follows:

- Source: `\Working\platform\QT\`
- Compiled classes: `\Working\platform\config\QTCLS\DLLname\`
- JAR files: `\Working\platform\config\QT\QTDLLname.jar`
-  QuickTest executable location: `\Working\platform\config\qt.bat`
-  QuickTest executable location: `/Working/platform/config/qt.ksh`  
Object Builder does not automatically make the qt.ksh file executable. To change its mode, follow these steps:
  1. Change directory to the location of qt.ksh.
  2. Type the following command to change qt.ksh to be an executable file:  
`chmod +x qt.ksh`
-  QuickTest executable location: `/Working/platform/config/qt.ksh`

-  QuickTest executable location: `/Working/platform/config/qt.ksh`

Once your QuickTest client is built, you can run it from the Build Configuration folder's pop-up menu. Click **Build > Run QuickTest** to run the QuickTest client.

   **Note:** If you have debug and trace enabled, you must make sure that the JAVA\_HOME environment variable is set to the directory in which your JDK is installed. You must do this before you run Object Builder (if you will launch QuickTest from Object Builder), or before you start QuickTest (that is, before you can invoke the qt.bat, or the qt.ksh file from the command line).

#### RELATED CONCEPTS

"QuickTest" on page 611

#### RELATED TASKS

"Configuring builds" on page 549

"Defining a client DLL" on page 552

"Generating a makefile" on page 556

"Chapter 13. Testing applications with QuickTest" on page 611

## Rebuilding DLLs

### 

If you have made changes to your build configuration, and want to rebuild your DLLs without recompiling any code, you can use the command-line option `cleandll` with the `nmake` or `make` command.

For example:

```
obgen -pe:\myproject -aMake
```

```
nmake -f all.mak cleandll all
```

will generate the makefiles for your updated build configuration, then delete the existing DLLs and supporting files (such as `.def`). Finally, it will rebuild the DLLs according to the updated makefiles.

**Note:** If your compile command fails due to an incorrect DB2 user ID and password error, run the following command before you run the `make` (AIX) or `nmake` (NT) command:

```
> export IVB_DB2AUTH="USER test USING password"
 set IVB_DB2AUTH=USER test USING password
```

You can also run `nmake` in two separate steps:

1. Run `nmake` without the `all` option:  
`nmake -f all.mak cleandll`  
This will only delete the existing DLLs and supporting files, but not rebuild the DLLs.
2. Next, run `nmake` without the `cleandll` option:  
`nmake -f all.mak`  
This will rebuild the DLLs.

**Note:** It is advisable to run the Consistency Checker before you generate makefiles, or start building, to ensure that the configuration for your client DLL is correct. This is crucial particularly if your client DLL does not have at least one IDL file selected.

#### RELATED TASKS

- “Configuring builds” on page 549
- “Building the DLLs” on page 558
- “Generating code from the command line” on page 684

#### RELATED REFERENCES

- “obgen” on page 685
- “make options” on page 688

---

## Build configuration behavior

### Default Configuration

You can configure your builds to produce output for a particular use, such as production or debugging. Globally, you can set a default, that will be in effect unless explicitly overridden in a particular DLL configuration. You can set one of four default configurations: debug, production, trace, or trace and debug. The global configuration determines where all DLLs are built, even if a particular DLL is built with a different configuration. For example, if you are building for production use, but have one DLL configured for debugging, all output, including that DLL, is placed in the `Working\platform\PRODUCTION` directory.

Set the configuration you want as your default in the Object Builder Preferences notebook. Click **File > Preferences** to open the notebook, then click on the Tasks and Objects - Makefile Generation node.

You can select one of the following configurations:

- **Unoptimized**  
Creates unoptimized output. This configuration gives shorter build times at the expense of run-time performance. Output goes in the project’s

Working\platform\NOOPT directory. Equivalent to setting the make macro IVB\_UNOPTIMIZE=1 (page 692) (or IVB\_OPTIMIZE=0).

- **Production**

This is the default. Optimizes output for production use. Output goes in the project's Working\platform\PRODUCTION directory. Equivalent to setting the make macro IVB\_OPTIMIZE=1 (page 690) (or IVB\_UNOPTIMIZE=0).

- **Trace**

Enables output for tracing. Output goes in the project's Working\platform\TRACE directory. Equivalent to setting the make macro IVB\_TRACE=1 (page 691).

- **Trace and debug**

Enables output for tracing and distributed debugging. Output goes in the project's Working\platform\TRACE\_DEBUG directory. Equivalent to setting the make macro IVB\_TRACE\_DEBUG=1 (page 691).

If you are building all.mak from the command line, you can select a default configuration by defining the equivalent macro.

#### **RELATED TASKS**

"Configuring builds" on page 549

Specifying a build location

"Defining a client DLL" on page 552

"Defining a server DLL" on page 554

"Generating a makefile" on page 556

"Building the DLLs" on page 558

#### **RELATED REFERENCES**

"Build options" on page 568

"make options" on page 688

## **Build targets**

When you generate the makefiles for your DLLs, you can set the default targets they will build.

You can generate for the following default targets:

- **C++ Default Targets**

Builds C++ client and server DLLs, in the Working\platform\config directory.

- **Java Default Targets**

Compiles Java class files, in the

Working\platform\config\JAVACLS\DLLname directory.

Builds Java server JAR files for server DLL configurations that contain Java business objects, in the Working\platform\config\ directory.

Builds Java client JAR files (that run on the server) for all client DLL

configurations (to allow access to components on the server using the Java language). JAR files are placed in the `Working\platform\config` directory.

- **Java Client Bindings Default Targets**

Compiles Java client binding files, in the

`Working\platform\config\JCBCLS\clientDLLname` directory.

Builds Java client binding JAR files for all client DLL configurations in the `Working\platform\config\JCB` directory.

-  **ActiveX Interfaces Default Targets**

Generates source files and a makefile into the

`Working\NT\config\ACTIVEX\` directory. You will still need to call the

makefile from outside of Object Builder in order to build the ActiveX client DLLs.

-     **QuickTest Default Targets**

Compiles QuickTest client files, in the

`Working\platform\config\QTCLS\DLLname\` directory.

Builds the QuickTest client .jar file in the

`Working\platform\config\QT\QTDLLname.jar`.

Places the qt QuickTest script file that is used to run the QuickTest client application, in `Working\platform\config\`

- **All Targets**

Builds all of the above.

The default target that you select determines the default behavior of the `all.mak` makefile. This behavior can be overridden when you build, either from the command line (using a make option to specify a different target), or from within Object Builder (selecting a specific target from the Build Configuration folder's **Build > Out-of-Date Targets** menu).

#### RELATED TASKS

"Configuring builds" on page 549

"Generating a makefile" on page 556

Specifying the order of a build

"Building the DLLs" on page 558

"Building the JAR files" on page 561

"Building for ActiveX clients" on page 562

"Building for Java clients" on page 563

"Building for QuickTest" on page 564

## Build options

When you build DLLs from within Object Builder, you have a number of options that define how and where your DLLs and JAR files will be built.

These options correspond to make options that you can set on the command line when making `all.mak`.

## Configuration options








You can set a default configuration within Object Builder, in Object Builder's Preferences notebook. Click **File > Preferences**, then navigate to the **Tasks and Objects > MakeFile Generation** page, to select which configuration to use when building within Object Builder. The option you select in the notebook corresponds to a command-line option you can use with all.mak, and also sets which configuration is set as the default within all.mak (when you call all.mak from the command line).

Default Configuration	make option
Unoptimized	IVB_NOOPT=1
Production(default)	IVB_OPTIMIZE=1
Trace	IVB_TRACE=1 (page 691)
Trace Debug	IVB_TRACE_DEBUG=1 (page 691)

## Target options

The following options determine which targets get built by the make program. When you build within Object Builder, the appropriate make option is defined based on your choice in the Build Configuration folder's pop-up menu. For example, if you click **Build > Out-of-Date Targets > Java**, then make is called with the java option.

When you generate a makefile, you can set a default option. From the Build Configuration folder's pop-up menu, click **Generate > Selected > Targetto** generate only all.mak, and set the default, or **Generate > All > Targetto** regenerate all the DLL makefiles, as well as local.mak (which is an all.mak that filters out client and server makefiles from the current project's dependencies), and to set the default. The build target that you define during generation sets a corresponding make option as the default, as shown in the following table:

Platforms	Build Target	make option
All	C++ Default Targets	cpp (page 693)
 	Java Default Targets	java (page 693)
 	Java Client Bindings Default Targets	jcb (page 693)
	ActiveX Interfaces Default Targets	activex (page 692)
 	QuickTest Default Targets	quicktest (page 693)
All*	All Default Targets*	all (page 692)*

\*Builds all targets available for a particular platform, except for ActiveX and QuickTest, which are excluded by default, but can be included with selections in the Preferences notebook, on the MakeFile Generation page.

### **DLL options**

The following options can be set within Object Builder on a per-DLL basis (within a DLL configuration's wizard). They are stored within all.mak, so you will need to re-generate all.mak before your next build for the options to take effect.

You can also specify these options on the command line when you make all.mak (as described in make options). Options specified on the command line can override the default configuration selected within Object Builder. They do not override any DLL-specific options set within Object Builder.

DLL-specific options do not affect the output directory. For example, if you set IVB\_NOOPT=1 for ClaimS.DLL, and then build with the default configuration setting Production (or build from the command line with IVB\_OPTIMIZE=1), all DLLs will be built into Working\platform\PRODUCTION, including the unoptimized ClaimS.DLL

- IVB\_NOOPT=1
- IVB\_OPTIMIZE=1
- IVB\_TRACE=1 (page 691)
- IVB\_TRACE\_DEBUG=1 (page 691)
- IVB\_DYNAMIC\_LINK=1 (page 690)
- IVB\_BUILD\_VERBOSE=1 (page 689)

#### **RELATED TASKS**

- “Using Object Builder from the command line” on page 659
- “Generating code from the command line” on page 684
- “Setting Object Builder preferences” on page 27

#### **RELATED REFERENCES**

- “Default Configuration” on page 566
- “Build targets” on page 567
- “make options” on page 688

---

## **Remote build configuration**

### **Remote build**

A build that is activated on another computer that is distant from a central site, usually over a network connection. The remote computer may be stationary and non-portable, or it may be portable.



**RELATED CONCEPTS**

“Profile”

“Pass ticket”

“Cross-platform development” on page 426

**RELATED TASKS**

“Launching a remote OS/390 build” on page 572

“Tutorial: Launching a remote OS/390 build” on page 573

## Pass ticket

In Resource Access Control Facility (RACF) secured sign-on, and for the OS/390 secure server, a pass ticket is a dynamically generated, random, one-time-use, password substitute that a workstation or other client can use to sign on to the host rather than sending a RACF password across the network.

This pass ticket is composed of eight characters, which can be any of the letters A to Z, and the digits 0 to 9.

A pass ticket can be used only once in the ten-minute period from its generation. It acts as a secure bridge from legacy applications to the modern world, though it is not as secure as digital certificates.

**RELATED CONCEPTS**

“Remote build” on page 570

“Profile”

**RELATED TASKS**

“Launching a remote OS/390 build” on page 572

“Tutorial: Launching a remote OS/390 build” on page 573

## Profile

In Object Builder, when you are specifying the options for a remote OS/390 build, you can optionally specify the name of a profile file. This is a shell file that contains initializations of the OS/390 environment variables.

**RELATED CONCEPTS**

“Remote build” on page 570

“Pass ticket”

**RELATED TASKS**

“Launching a remote OS/390 build” on page 572

“Tutorial: Launching a remote OS/390 build” on page 573

## Launching a remote OS/390 build

Preliminary steps:

- Ensure that the rexec daemon is running on the OS/390 host machine.
- Have Object Builder for Windows NT up and running.
- Change the platform view to OS/390. Select **Platform > View > 390** from Object Builder's main menu.

To launch a remote build, follow these steps:

1. Select the Build Configuration folder in the Tasks and Objects pane.
2. From its pop-up menu, select **Remote OS/390 Options**. The Remote Build wizard opens to the OS/390 Options page.
3. Specify the name of the OS/390 machine on which you want to run the remote build in the **Host Name** field.
4. Type the user ID and password by which you will access the host machine.
5. Type the full directory path on the OS/390 host machine, which is to contain the files generated by Object Builder in the **Host Directory** field. This is the directory that will contain the files and directories that are normally contained in Object Builder's Working\390 directory after file generation.
6. You can optionally specify the name of a shell profile file, to be used to initialize environment variables.
7. Indicate the format in which data is to be returned by the host. You can choose between the American Standard Code for Information Interchange (ASCII) and the Extended Binary Coded Decimal Interchange Code (EBCDIC). Your selection determines the data translations, if any, that are required.
8. Click **Finish**.  
Your user ID and password are stored in memory, but the host name, host directory, and profile name (if you provide it) are saved.
9. Select the Build Configuration folder again, and from its pop-up menu, select **Build**.  
The remote build will be activated if the host name, user ID, password, and host directory are properly set.  
**Note:** As long you use the same Object Builder session, you do not have to retype your user ID and password each time you want to do a remote build; you can just execute step 9. You will have to follow the preliminary steps, and steps 1 to 4, 6, 8 and 9, if you close Object Builder, and restart it.

### RELATED CONCEPTS

"Remote build" on page 570

## RELATED TASKS

“Tutorial: Launching a remote OS/390 build”

### Tutorial: Launching a remote OS/390 build

Preliminary steps:

- You must ensure that the rexec daemon is running on the OS/390 host machine.
- Optionally, create an NFS read/write mount of your OS/390 host directory. You can then generate code directly into the NFS mounted directory. Instead, you can generate the files onto your local file system, and then use the File Transfer Protocol (FTP) to transfer them over to an Open Edition for OS/390 system.
- Set up Object Builder for Windows NT.
- Once it is running, change the platform view to OS/390. Select **Platform > View > 390** from Object Builder’s main menu.

To launch a remote build, follow these steps:

1. Select the Build Configuration folder in the Tasks and Objects pane.
2. From its pop-up menu, select **Remote OS/390 Options**. The Remote Build wizard opens to the OS/390 Options page.
3. Specify machine.host.com as the name of the OS/390 machine on which you want to run the remote build in the **Host Name** field.
4. Type the user ID and password by which you will access the host machine.
5. Type .../Working/390 as the full directory path on the OS/390 host machine, which is to contain the files generated by Object Builder in the Host Directory field. This is the directory that will contain the files and directories that are normally contained in Object Builder’s Working\390 directory after file generation.
6. If you maintain the settings of the Component Broker Toolkit environment variables such as CLASSPATH, PATH, and so on in a shell profile file similar to .profile in AIX, specify its name.
7. Accept the default, which is the ASCII (American Standard Characters for Information Interchange) format in which data is to be returned by the host. The other choice is the Extended Binary Character Digital Interchange Code (EBCDIC). Your selection determines the data translations, if any, that are required.
8. Click **Finish**.  
Your user ID and password are stored in memory, but the host name, host directory, and profile name are saved.
9. Select the Build Configuration folder again, and from its pop-up menu, select **Build**.

The remote build will be activated if the host name, user ID, password, and host directory are properly set.

**Note:** As long you use the same Object Builder session, you do not have to retype your user ID and password each time you want to do a remote build; you can just execute step 9. You will have to follow the preliminary steps, and steps 1 to 4, 6, 8 and 9, if you close Object Builder, and restart it.

You can now make incremental changes to your files, and you will not have to use the File Transfer Protocol.

**RELATED CONCEPTS**

“Remote build” on page 570

**RELATED TASKS**

“Launching a remote OS/390 build” on page 572

---

## Packaging applications

When your application is ready to ship, you can package it for easy installation at a customer site. Applications consist of managed object configurations, which define the component objects and DLLs you want installed on the server.

To package an application, follow these steps:

1. “Creating an application family” on page 575
2. “Adding an application” on page 576
3. “Creating a container instance” on page 578
4. “Configuring a managed object” on page 588
5. “Generating the DDL files” on page 593
6. “Documenting applications” on page 595

**RELATED CONCEPTS**

“DDL” on page 137

**RELATED TASKS**

“Developing in Object Builder” on page 19

**RELATED REFERENCES**

“Internationalization of data” on page 132

“Naming objects” on page 128

## Creating an application family

▶CHANGED

An application family consists of one or more applications that are packaged together on a CD and need to run at the same code level. There is a single installation process for each application family you define. You can group applications in a family to ensure version compatibility. The installation checks each application's version, and at the end of the installation ensures that all applications in the family are at the same version.

When you install an application family, you cannot select which applications you want to install. You must install all or none of the applications in the family.

Application families consist of applications, which include all the necessary code (DLLs, JAR files, and so on) necessary to support the deployment of configured components on the server.

To create an application family, follow these steps:

1. Under Tasks and Objects, select the Application Configuration folder.
2. From the folder's pop-up menu, select **Add Application Family**. The Add Application Family wizard opens to the Name page.
3. Type a name, description, and version number for the application family.
4. Select the platforms on which your application is to be deployed.
5. Optionally, type the name of the DDL file that will be generated. If you do not specify a name, Object Builder uses the name of the application family for the generated DDL.

▶ 390 When OS/390 is one of the deployment platforms, the DDL file name must not exceed eight characters in length. The first character must be alphabetic, and the other characters (positions 2 to 8) must be alphanumeric.

6. Click **Finish**. The application family is added to the Application Configuration folder.

You can now add applications to the application family.

### Note the following points:

- ▶ WIN ▶ AIX ▶ 390 ▶ SOLARIS ▶ HP-UX Object Builder generates the DDL file on each deployed platform. (Object Builder is capable of generating the file on all platforms - Windows NT, AIX, OS/390, Solaris, and HP-UX). You will need to load this file into System Management when you deploy your applications.
- Your DDL file is generated along with a backup version. For example, if the name of your application family is MyAppFam, and

you have specified the name of the DDL file as AppFam, the DDL emitter will generate the following files:

Working\platform\output\_directory\MyAppFam\AppFam.ddl

Working\platform\output\_directory\MyAppFam\backup\AppFam.auto.ddl

platform is one or more of NT, AIX, Solaris, 390, or HPUX (the platforms on which you are deploying your application).

output\_directory is each of NOOPT, PRODUCTION, TRACE,

TRACE\_DEBUG (that is, for a given platform, you get four sets of DDL files generated).

#### RELATED TASKS

“Packaging applications” on page 574

“Adding an application”

#### RELATED CONCEPTS

“Application DDL files” on page 602

#### RELATED REFERENCES

“Internationalization of data” on page 132

“Naming objects” on page 128

## Adding an application

An application consists of components, which encapsulate distributed data and resources for the use of a client application.

To add a server application to your application family, follow these steps:


1. From the Application Configuration folder, select your application family.
2. From the family’s pop-up menu, select **Add Application**. The Add Application wizard opens to the Name and Environment page.
3. Enter a name, description, and version number for the application.  
You can also specify a Java virtual machine name.
4. Click **Next**. The Additional Executables page opens.
5. Browse for and select the following files:
  - Any client DLL or .jar files (defined in Object Builder) for components in other application families that are referenced by your application.  
DLLs and .jar files for components in this application are automatically included when you configure the component managed objects with the application.
  - Any additional DLLs or .jar files (not defined in Object Builder) that contain code required by your application.
  - Any bind files for components that use embedded SQL (that is, the component’s data object implementation has the **Embedded SQL** option set on the Behavior page of its wizard).


Bind files are the compiled form of a persistent object .sqx file (for example, ClaimPO.sqx becomes ClaimPO.bnd).

- Any SQL files for components that connect to new (as opposed to pre-existing) database tables.

When the server application is installed, the SQL files can be used to configure the database for use by the application's components.

- Any JAR files that exist on your system, and have to be available when the application server runs (if the application is backed by a procedural application adaptor). These JAR files include the one that contains the PA bean that you imported into Object Builder (and are going to use with this application), and those that contain classes that are used by the bean. JAR files are platform-independent, and you must add them for all of the platforms on which you have the server running.

 The JAR file is copied into the x:\Cbroker\ntApps\

 The JAR file is copied into the x:\Cbroker\aixApps\

When the application server is activated for the first time, the JAR file is added to the class path of the server, following which the application is associated with the server. The classes in the JAR file are made available to the server at run time. So, you do not have to add the JAR file to the classpath environment variable.

**Note the following points:**

- The JAR files that you add as additional executables take precedence over the directory into which you import the JAR files (when you import a PA bean) since the JAR files are added to the front of the CLASSPATH when the Component Broker server is started.
- If the DLL or JAR file that you want to add does not appear in the Object Builder file dialog, ensure that the file is not hidden. Using Windows NT Explorer, select **View > Options**. In the Options dialog box, click the View tab and set the **Show All Files** check box. Close the Object Builder file dialog box, close the wizard, save your work, and shut down and restart Object Builder. When you return to the wizard, turn to this page and click the Browse button again, the file should be displayed.

6. Click **Finish**. The application appears under your application family in the Application Configuration folder.

You can now configure managed objects with your application and, if you want, create a container that handles object services for the managed objects.

**RELATED TASKS**

“Packaging applications” on page 574

“Creating a container instance”

“Configuring a managed object” on page 588

“Creating a PA schema by importing a PA bean” on page 862

**RELATED REFERENCES**

“Internationalization of data” on page 132

“Naming objects” on page 128

## Container

A container is a configured version of a particular application adaptor that represents a physical boundary around objects. It can be thought of as where the objects exist. A container can provide some level of isolation between it and other containers. A container can also provide some isolation among objects within the container.

When you add a managed object to an application, you configure it with a container that will be responsible for handling object services for the managed object.

A container is a part of the runtime environment that provides components with transaction and security management, network distribution of clients, scalable management of resources, and other services that are generally required as part of a manageable server platform. A container forms a logical boundary around components and can be thought of as where the components exist.

When you add a managed object to an application, you configure it with a container that will be responsible for providing the required quality of service and that also defines management properties used by the application adaptor.

**RELATED CONCEPTS**

Managed object (*Programming Guide*)

**RELATED TASKS**

“Configuring a managed object” on page 588

“Creating a container instance”

## Creating a container instance

The Component Broker frameworks provide a number of default containers, which are appropriate for components with transient data (that is, without persistent objects) on Windows NT, AIX, Solaris, and HP-UX. If your component has data that you want to be persistent, or you are targeting

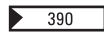


different or additional platforms, you must define a container for the particular needs of the component. You can define new containers in the Container Definition folder.

You can also define new containers as part of the process of configuring a managed object. If you select the **Create a new container** option, in the Managed Object Configuration wizard, Container page, a default container instance will be created that provides behaviors that are appropriate for the selected managed object and its data object.

To add a container instance in the Container Definition folder (without basing the container on an existing managed object), follow these steps:

1. Under Tasks and Objects, select the Container Definition folder.
2. From the folder's pop-up menu, click **Add Container Instance**. The Container wizard opens to the Name page.
3. Type a name and description for the container.

 If you are developing an application intended for deployment on OS/390 (the **Platform > Constrain > 390** menu choice is checked), then you are now done. The rest of the container definition is handled through the System Management user interface.

4. In the **"Deployment platforms" on page 423** section, select the platform or platforms on which the container will be deployed.
5. In the **Number of Components** field, type an estimate for the number of managed objects this container will hold. This sets a lower limit on the size of the container's hash table; additional space will be allocated when it is needed.
6. Click **Next**. The Workload Management page opens.
7. Specify whether the container is workload managing. If you check this option, you must also specify the policy group it will be configured with. For new policy groups, accept the default <New> entry.
8. Click **Next**. The Service page opens. If you select not to provide Transaction Services, select how the container should handle object data when the server stops running (the **Save all data when server is about to stop (page 597)** policy).
9. Select whether the container should passivate objects not in use, or keep objects in memory at all times (the **Passivate a component after its data is saved (page 597)** policy).
10. If you select not to provide any object services (**Use no Object Services**), select whether to **Enable persistent references** (page 598).
11. Select the **Container-managed entity beans only** check box if you want this container to be available for selection when you are configuring the managed object that is associated with a CMP entity bean.

**Note:** This check box is enabled only if either **Use RDB Transaction Service**, **Use PAA Transaction Service**, **Use PAA Session Service**, or **Use MQAA Transaction Service** is selected. In other cases, it is disabled.

12. Click **Next**.
13. If you selected **Use RDB Transaction Service (page 596)**, **Use PAA Transaction Service (page 596)**, **Use PAA Session Service (page 596)**, or **Use MQAA Transaction Service (page 596)**, then the Services Details page now opens. Specify the behavior you want for methods called outside the scope of a transaction or session (the **“Behavior for Methods Called Outside a Transaction” on page 599** and **“Behavior for Methods Called Outside a Session” on page 600** policies), and for sessions specify the type of session. For PAA Session Services, specify the name of the connection (**“Connection” on page 601**).
14. Click **Next** when you are done.
15. The Data Access Patterns page opens. Select the options on this page according to the options set for the objects the container will hold.
16. Under **Business Object**, click **Delegating** or **Caching** according to the option selected for the business object implementation’s **Pattern for Handling State Data** (Business Object Implementation wizard, Name and Data Access Pattern page).
17. Under **Data Object**, click **Delegating** or **Local copy** according to the option selected for the data object implementation’s **Data Access Pattern** (Data Object Implementation wizard, Behavior page).
18. If you select **Delegating**, then you need to indicate whether or not the data object uses the Cache Service. Select the **Cache Service** check box if the data objects have their **Type of Persistence** set to **Cache Service** (Data Object Implementation wizard, Behavior page). Otherwise, click **No**.
19. Click **Finish**. The new container is added to the Container Definition folder.

#### **RELATED CONCEPTS**

“Container” on page 578

Transaction Service (*Advanced Programming Guide*)

Session Service (*Advanced Programming Guide*)

Cache Service (*Advanced Programming Guide*)

Workload management (Using Object Builder) (*Advanced Programming Guide*)

#### **RELATED TASKS**

“Packaging applications” on page 574

“Working with container instances” on page 883

“Configuring a managed object” on page 588’

#### RELATED REFERENCES

“Internationalization of data” on page 132

“Naming objects” on page 128

Container configuration parameters (*Programming Guide*)

Typical settings for container configuration parameters (*Programming Guide*)

Summary of supported container configurations (*Programming Guide*)

## Home

A home is the birthplace of managed objects. It serves as both a factory and a collection for managed objects. It is like a factory designed to manufacture only objects of a specific type.

Component Broker provides some default instances of homes, and most managed objects will use home instances based on these default ones. However, you can create a specialized home in Object Builder if your managed objects require home instances with additional or specific behaviors.

When you add a managed object to an application, you select the type of home that will be used to create it on the Add Managed Object wizard, Home page. When you generate the DDL files for the application, the generated DDL defines the home instance that will be responsible for finding and creating instances of the managed object.

#### RELATED CONCEPTS

Managed object (*Programming Guide*)

“DDL” on page 137

Creating specialized homes (*Programming Guide*)

#### RELATED TASKS

“Creating a specialized home” on page 876

“Configuring a managed object” on page 588

## Polymorphic homes



A polymorphic home is one that supports polymorphic behavior within a set of classes that inherit from one another. It can have objects of different types and subtypes configured into it.

A polymorphic component has `IPolymorphicHome` as the parent of its business object interface; `ISpecializedPolymorphicHome` as the parent of its business object implementation; and `ISpecializedPolymorphicHomeManagedObject` as the parent of its managed object.

Polymorphic homes provide functional support that is referred to as run-time inheritance support because the run time provides interfaces and behavior that takes inheritance relationships into account. This support which includes query support and relationship (association) support.

### **Query support**

You can execute a query over a home, and return types as well as subtypes of the objects that are supported by that home, for all types that are configured into a polymorphic home.

### **Relationship support**

This support will also include polymorphic support for 1-1 and 1-n relationships. Support for 1-n support is provided by query support. Support for 1-1 support is provided by a new polymorphic find interface.

Component Broker currently supports polymorphic homes only with the single table inheritance pattern.

A managed object is considered appropriate for configuration into a polymorphic home when two conditions, both related to the data object implementation are met.



- A discriminator predicate is defined on the data object implementation
- The data object implementation is persisted according to the *single table (or datastore) pattern*. This can occur in one of the following ways:
  - The data object implementation is at the top of the hierarchy (it inherits from the framework), and persists to its own persistent objects (one or more).
  - The data object implementation inherits from another data object implementation, and persists only to the persistent objects (one or more) of the data object implementation at the top of the hierarchy.

So, for polymorphism, an entire hierarchy of data object implementations is mapped to a single table.

### **Note the following points:**

- When you create a business object that implements a specialized, polymorphic home, Object Builder selects `IManagedAdvancedServer` `IManagedAdvancedServer::ISpecializedPolymorphicHome` as the default class to inherit from. This is because a polymorphic home requires a fully backed business object interface (complete with business object implementation inheritance); business object interface inheritance from the polymorphic home is not sufficient.
- Only if data object implementations are part of the single table mapping can they be configured into polymorphic homes.

### Restrictions:

- Abstract classes must have queryable homes
- The same primary key class must be used for all types and subtypes
- All data for the types can be stored only in the same database type
- All polymorphic homes must exist in the same server
- Multiple parent inheritance is not supported by Query for polymorphic homes in this version of Object Builder
- Component Broker does not support polymorphism over configured managed objects using either the inheritance with attributes duplication pattern, or the inheritance with key duplication pattern. Object Builder enforces these restrictions.
- Object Builder does not support container configuration of atomic transactions for query (start a new transaction)
- Object Builder does not support typed tables (this support is specific to DB2 Version 6)
-   If a business object interface is deployable to either OS/390 or HP-UX, you cannot have a polymorphic home class as the parent for either the business object interface, or the implementation. That is, you cannot select the IPolymorphicHome class as a parent class for the business object interface, and you cannot select IPolymorphicHome class as a parent for the business object implementation. The same is true for a managed object that is deployable to OS/390 or HP-UX: it cannot inherit from the ISpecializedPolymorphicHomeManagedObject class.

### RELATED CONCEPTS

“Home” on page 581

“Container” on page 578

Polymorphic behavior for extended business objects (*Programming Guide*)

Managed object (*Programming Guide*)

Data object (*Programming Guide*)

### RELATED TASKS

“Adding a business object interface” on page 777

“Adding a business object implementation and data object interface” on page 780

“Adding a managed object” on page 871

“Adding a data object implementation” on page 807

“Configuring a managed object” on page 588

“Creating a specialized polymorphic home” on page 880

“Packaging applications” on page 574

“Tutorial: Inheritance with views” on page 362

#### RELATED REFERENCES

IPolymorphicHome interface (*Programming Reference*)  
ISpecializedPolymorphicHome Interface (*Programming Reference*)  
ISpecializedPolymorphicHomeManagedObject Interface (*Programming Reference*)  
Keywords for query support

## Managed object configuration behavior

Managed object configurations include the definition of a home instance to be used when creating or locating instances of the component.

You can set the following options for a managed object configuration's home instance:

- "Home Name"
- "Home Options" on page 587

#### RELATED CONCEPTS

Managed object (*Programming Guide*)  
"Home" on page 581

#### RELATED TASKS

"Working with managed objects" on page 869

## Home Name



When you create or edit a managed object configuration, you can specify the home instance that will be used to create instances of the component.

On the Home page of the Managed Object Configuration wizard, you can select whether you are using a default home (provided by Component Broker) or a specialized home (provided by you or another Component Broker user). The home must be the same as the home to query that you specified.


The **Home Name** field lists two different types of homes depending on your selection on the Home page.

### Default homes

If you select the **Default Home** button, the **Home Name** field lists names of appropriate, predefined home instances, which can create a default home (either any instance of a home class that is provided by Component Broker, or the predefined home instances, which can be used to create such homes).

Home instances that can appear in the list:

- **BOIMHomeOfRegHomes**  
This entry is only available when you select **Default Home**. If you want your home instance to be registered with the Life Cycle and Naming services, select this entry, or select a specialized home with equivalent behavior. The managed object's home instance will be created by this home instance.
- **BOIMHomeOfNotRegHomes**  
This entry is only available when you select **Default Home**. If your home instance does not require the Life Cycle or Naming services, select this entry, or select a specialized home with equivalent behavior. The managed object's home instance will be created by this home instance.
- **BOIMHomeOfRegQIHomes**  
This entry is only available if the managed object is queryable (as set on the associated business object interface).
- **BOIMHomeOfRegWLMHomes**  
This entry is automatically selected if you selected a workload-managing container on the Container page, and the managed object is *not* queryable (as set on the associated business object interface).  
You can still select a workload-managed (WLM) home even if your component is not workload-managed.

 When the constraint platform is OS/390 (**Platform > Constrain > 390**), this option is not available for selection. These default WLM homes do not exist on OS/390.

- **BOIMHomeOfRegWLMQIHomes**  
This entry is automatically selected if you selected a workload-managing container on the Container page, and the managed object *is* queryable (as set on the associated business object interface). If your component is *not* workload managed, you can still select a workload managed home, although this will not make the component workload managed: the choice of container is what makes a component workload managed.

### Specialized homes

If you select the **Specialized Home** button, the **Home Name** field lists class names.

- **Specialized home classes (business object interface names)**  
These entries are only available when you select **Specialized Home**. It lists the specialized homes you have defined in Object Builder. You *must* configure the selected specialized home with the current managed object: specialized homes, and the managed objects that use them, cannot be in separate applications. The managed object's home will be an instance of the selected specialized home class.

   **Polymorphic homes**

The **Home Name** list also contains two new default (or system) homes as

appropriate on the Windows NT, AIX and Solaris platforms only for configured managed objects that are candidates for polymorphism. These homes are available (and can also be viewed in the Framework Interfaces folder in the Tasks and Objects pane, along with their methods) only when the **Platform > View** menu does not include either OS/390 or HP-UX.

A managed object is a candidate for polymorphism only when its data object implementation follows the single table pattern of inheritance. That is, it satisfies both these conditions:

- Its data object implementation has no parents (it is at the top of the inheritance hierarchy, and inherits directly from the framework), and it is persisted to one or more of its own persistent objects
- Its data object implementation inherits from a parent, and is persisted to one or more of its parent's persistent objects only.  
A data object implementation, which inherits from another implementation, and also persists to its own persistent object, violates the single table pattern of inheritance.
- Its business object inherits from IPolymorphicHome.

The following homes can also appear in the Home name list:

- BOIMHomeOfPolymorphicHomes  
This entry is available for selection for any managed object whose business object is queryable, and whose container is not workload-managed.
- BOIMHomeOfPolymorphicWLMHomes  
This entry is available for selection for any managed object whose business object is queryable, and whose container is workload-managed.

**Note:**If the business object is queryable, the list will also contain any specialized, polymorphic homes from the Object Builder models. Specialized, polymorphic homes are not available for selection unless the same conditions are met.

### Message queuing-specific homes

- IMessageHomeMO\_OutboundMessageQueueMO  
This entry is available only when you select **Default Home**. It is the OutboundMessages home, designed for use with message queues. It provides a put method, instead of the usual createFromXxx method, for the creation of the message objects.
- IMessageHomeMO\_InboundMessageQueueMO  
This entry is available only when you select **Default Home**. It is the InboundMessages home, designed for use with message queues. It provides a get method, instead of the usual findByXxx method, for the retrieval of the message objects.



**Note:** Make sure that the home uses a different container instance than the managed object. If necessary, create a separate container instance for the managed object. If a managed object and its home are configured with the same container, the server will not activate.

#### RELATED CONCEPTS

“Home” on page 581

Workload management (*Advanced Programming Guide*)

Query Service for Windows NT and AIX (*Advanced Programming Guide*)

Query Service for OS/390 and Solaris (*Advanced Programming Guide*)

#### RELATED TASKS

“Configuring a managed object” on page 588

“Creating a specialized home” on page 876

## Home Options



When you create or edit a managed object configuration, you can provide information about the home instance that will be used to create instances of the component.

Home options can be set on the Home page of the Managed Object Configuration wizard. The home options you select define how the home instance will be registered with services, and the state of the home. Home instances have the following options:

- **Name in Factory Finding Service Registry**  
Define the name used to register the managed object’s home with the life cycle factory finding service.  
**Note:** This must be the same as the name of the factory name, if you provided one, on the Attributes Mapping page of the data object implementation.
- **Name in Naming Service Registry**  
Define the name used to register the managed object’s home with the naming service.
- **Name Contexts**  
If this managed object is a specialized home, then this field determines where instances of the specialized home will be registered in the system. In most cases, you should accept the default for this field.  
If this managed object is **not** a specialized home, then this field is ignored.
- **Locatable by Cell**  
Makes the home visible in the cell name tree, which allows the life cycle service to locate the home or factory by its cell.

- **Locatable by Workgroup**

Makes the home visible in the workgroup name tree, which allows the life cycle service to locate the home or factory by its workgroup.

- **State of Home**

Defines the state you expect the home to be in when the managed object is installed. You can select from **exists** (the home is defined but has not been used) or **created**(the home has been used at least once). Select **exists** if the home is new or has never been used. Select **created** if the home is part of an existing, installed application.

**RELATED CONCEPTS**

“Home” on page 581

**RELATED TASKS**

“Configuring a managed object”

“Creating a specialized home” on page 876

## Configuring a managed object

Once you have defined a server application, you can add and configure the managed objects you want your application to consist of.

To add a managed object to an application, follow these steps:

1. From the Application Configuration folder, select your application.
2. From the application’s pop-up menu, select **Add Managed Object**. The Configure Managed Object wizard opens to the Selection page.
3. Select the managed object from the drop-down list.  
If the managed object has been added to a DLL, and is associated with a key and a copy helper, then the primary key, copy helper, and DLL fields are filled in for you. You can type over these automatic selections, or make alternative selections from the drop-down lists.
4. You can also select a configured managed object parent interface for the managed object that you selected from the **Parent** field.  
**Note:** If the managed object has no parents (yet, or, at all), this field is empty, and disabled. If the managed object has a single parent, this field displays the parent, and is disabled. If the managed object has two or more parents, they are listed in ascending lexical order, and the first one is selected by default. You can change the selection.
5.  WIN  AIX  SOLARIS If you are configuring the application for polymorphism, and if the **Parent Interface for Polymorphism** field is enabled, select a parent interface to act as the parent for polymorphic queries and findBy’s.

**Note:** This box lists all parent managed objects that are configured within application families. It is enabled only if there is more than one parent configured managed object.

6. Click **Next**. The Data Object Implementations page opens.
7. From the Implementations pop-up menu, select **Add**.
8. Select the data object implementations that will be available to the application, and associated DLLs. Note that this is a packaging statement, and not a configuration statement.



You can only select data object implementations whose type of persistence matches the service provided by the managed object (transaction service for DB persistence, session service for PA persistence). If you select a data object implementation that is associated with a PA bean, and if you have not included all the JAR files that are associated with the bean, and that have to be present when the application server runs, you must first add them on the Additional Executables page of the Add Application wizard. (Select the application in the Tasks and Objects pane, and from its pop-up menu, select **Properties**.) You must add them separately for each platform on which you have the server running.

9. Click **Next**. The Container page opens.
10. Specify whether you want to use a workload managing container. If you select this option, then only workload-managing containers are available in the Container list.
11. Select the container to use with this managed object. The container determines the quality of service (that is, how objects are instantiated, terminated, and so on). If you select a workload managing container, then the component will be workload managed.

**Note the following points:**

- Make sure that the managed object is configured with a different container than that used by its home. If necessary, create a separate container instance for the managed object. If a managed object and its home are configured with the same container, the server will not activate.
- If a container is designated 'not valid', you can click the '?' button to see why it is not valid.
- Valid containers must be deployable to at least all of the family's deployment platforms. Also, the DLLs containing the business object or managed object, key, copy helper, data object implementation and specialized home must also be tagged for at least the same platforms as the family.
- that are selected for the business object, and the managed object DLLs.
- If there is an enterprise bean that is associated with this managed object, and if it is a CMP entity bean (that is, it is not a session bean, and the associated data object implementation is neither

IXOInterface.DO\_IMPL\_DETAILS\_NoDetails nor IXOInterface.DO\_IMPL\_DETAILS\_Transient), then only containers that have the **Container-managed entity beans only** check box selected on the Service page of the Container Definition wizard are valid. Non-CMP beans and configured managed objects are not allowed to use containers that are meant for CMP beans.

-  If you are developing an application intended for deployment on OS/390 (the **Platform > Constrain > 390** menu choice is selected), then all containers are listed, and you need to make an appropriate choice based on the kind of managed object you are configuring, and the services it requires. The rest of the container definition is handled through the System Management user interface.
-  OS/390 does not support the default containers. The default containers are only set for Windows NT, AIX, Solaris, and HP-UX. If you are creating an application for 390 (you have the constraints set for 390), when you are building the application object, the Container page will not show any of the default containers: you must create your own container, and this container must have the 390 constraint set.

If no containers are listed, or you want to create a new one, follow these steps:

- a. Select the **Create a new container for this configuration** check box. Object Builder creates a default container into which you can configure the managed object. An extra page (the New Container page) will be added to the end of the wizard.
- b. On the New Container page, name the container, and set any additional behaviors that are not defined by default. By default, the container will have behaviors that are appropriate for the managed object and selected data object implementations, with the exception that all deployment platforms will be set as constraints for this new container, even if only a subset of platforms is all that is required based on the platform constraints for the associated managed object and data object implementations. If there is a CMP entity bean that is associated with this managed object, you can indicate whether this new container can be used for the configuration by selecting the **Container-managed entity beans only** check box.

**Note the following points:**

- This check box can be selected only if you had selected either **Use RDB Transaction Service**, **Use PAA Transaction Service**, **Use PAA Session Service**, or **Use MQAA Transaction Service** on the Service page.

- Containers that are available for use in configuring managed objects that are associated with CMP entity beans are not restricted to only those that have this check box selected.
- Containers that have this check box selected (are marked for use by only CMP entity beans) cannot be used to configure managed objects that are associated with either BMP entity beans, or session beans.

You can use one of these methods to change the platform constraints for the new container, based on the objects in the model:

- From the **Platform** menu, select **Constrain > Apply Constraints to Model**. All files, DLLs, and containers will be updated with the constraint platforms that are selected on this menu.
- From the pop-up menu of the container, select **Properties**, and clear the deployment platforms that are not required.

12. Click **Next**. The Home page appears.
13. Define the home to use with this managed object. The home can either be an instance of a default (system) home provided with Component Broker, or an instance of a specialized home which you created. If you specify a specialized home, you must also specify which DLL contains it. The home that you specify here (either a system home, or a specialized home) determines whether the home of the configured managed object in Systems Management will be polymorphic. If it is a specialized home, it inherits from the `IManagedAdvancedClient::IPolymorphicHome` interface. Only those homes that are appropriate for the current managed object are shown. For example, if the managed object is workload-managed (because you previously selected a workload-managing container on the Container page), then you must select a workload-managed home, and so only workload-managed homes are shown.
14. Select any other configuration options for the home
15. Click **Finish**. You have configured the managed object by choosing a copy helper and a key for it to work with, data object implementations for it to use, a container, a home, and the DLLs that contain it and the other objects. The managed object now appears in the Application Configuration folder, underneath the application you configured it for.

Once you have finished adding managed objects to your server applications, and have completed the configuration of the applications in your application family, you can generate the installation image for your application family.

**Exception:** You can use a data object to connect to a business object implementation even if the set of its deployment platforms is a superset of the platforms that you selected for the business object implementation.

#### RELATED CONCEPTS

“Home” on page 581  
“Container” on page 578  
Managed object (*Programming Guide*)  
Data object (*Programming Guide*)  
Naming Service (*Advanced Programming Guide*)  
LifeCycle Service (*Advanced Programming Guide*)  
Workload management (*Advanced Programming Guide*)

#### RELATED TASKS

“Packaging applications” on page 574  
“Working with managed objects” on page 869  
“Creating a container instance” on page 578  
“Creating a specialized home” on page 876  
“Adding an application” on page 576  
“Generating the DDL files” on page 593

## Parent Interface for Polymorphism



WIN  AIX  SOLARIS Polymorphism is supported only on the Windows NT, AIX and Solaris run times. This field is disabled for managed objects being deployed to either OS/390 or HP-UX.

This box lists all parent managed objects that are configured within application families. It is enabled only if there is more than one parent configured managed object. All these configured managed objects are listed in the form: *application\_name managed\_object\_name*. The list is ordered based on these names.

You can select only one of two or more parent interfaces to act as the parent for polymorphic queries and `findBy`'s.

#### Note the following points:

- If one or more parent managed objects are not yet configured, they will not appear in this list. So, it is important that managed objects are configured top-down with respect to inheritance. That is, you must configure all parent managed objects into a family before configuring the managed object that inherits from them.
- If there are no parents, this control is disabled, and empty.
- If the managed object interface inherits from exactly one parent, this control is disabled, but will display that parent configured managed object. This parent will be used for polymorphic query and `findBy`.
- If there are two or more parents, this control is enabled.

- For new configured managed objects, or for those that are not listed due to their deletion from the application family, the first entry in the list will be selected by default.
- The business object interfaces that are associated with these configured managed objects are the immediate parents of the business object interface that is associated with the managed object that is selected in the **Managed Object** field.

#### RELATED CONCEPTS

Managed object (*Programming Guide*)

Key (*Programming Guide*)

Copy helper (*Programming Guide*)

#### RELATED TASKS

“Configuring a managed object” on page 588

“Creating a specialized polymorphic home” on page 880

## Generating the DDL files

►CHANGED

Once you have created an application family, added applications to the family and configured your managed objects, you can generate the DDL file that defines your data object to the server.

To generate the DDL files, follow these steps:

1. From the Application Configuration folder, select your application family.
2. From the family’s pop-up menu, select **Generate**.

If you have multiple application families, you can generate the DDL files for all of them at once. Select **Generate > All** from the pop-up menu of the Application Configuration folder to generate images for all the families in the folder. You will still need to build the image for each application family individually.

Files are generated to each of the current project’s configuration directories:

- *project\Working\platform\NOOPT\AppFamilyName\\*.ddl*  
Used to load the unoptimized versions of the application family DLLs and JAR files.
- *project\Working\platform\PRODUCTION\AppFamilyName\\*.ddl*  
Used to load the production (optimized) versions of the application family DLLs and JAR files.
- *project\Working\platform\TRACE\AppFamilyName\\*.ddl*  
Used to load the trace-enabled versions of the application family DLLs and JAR files.

- `project\Working\platform\TRACE_DEBUG\AppFamilyName\*.ddl`  
Used to load the trace- and debug-enabled versions of the application family DLLs and JAR files.

▶ WIN ▶ AIX ▶ 390 ▶ SOLARIS ▶ HP-UX Each directory contains the DDL file for the application family (*AppFamily.ddl*). You will need to load this file into System Management when you deploy your applications.

**Note:** Your DDL file is generated along with a backup version (*AppFamily.auto.ddl*) . The DDL emitter thus generates the following files:  
`Working\platform\output_directory\ AppFamilyName \ AppFamily .ddl`  
`Working\platform\output_directory\ AppFamilyName \backup\ AppFamily .auto.ddl`  
*platform* is one or more of NT, AIX, Solaris, 390, or HP-UX (the platforms on which you are deploying your application).  
*output\_directory* is each of NOOPT, PRODUCTION, TRACE, TRACE\_DEBUG (that is, for a given platform, you get four sets of DDL files generated).

▶ 390 If you have developed your code for OS/390 , the generated DDL for the application family includes the statement:  
`targetplatform="390"`

This statement prevents the application family from being accidentally installed on an incompatible System Management platform.

You can now load the DDL into System Manager, and configure your application on the server.

Once the application is installed and configured with system management, you can run it to make the components available to client applications.

**Note:** Before you run a Java client application, you need to add the following JAR files to the beginning of your classpath:

- `somojor.zip`  
Contains classes to support the client-side Java ORB. If this is not at the beginning of the classpath, the wrong classes will be found, and your application will not run.
- The JAR files that contain your Java client bindings (located in the JCB subdirectory, with the naming convention `jcbMyObjectC.jar`). These contain classes to support a client application accessing the equivalent Java component on the server.



**RELATED CONCEPTS**

“DDL” on page 137

**RELATED TASKS**

“Creating an application family” on page 575

“Adding an application” on page 576

“Configuring a managed object” on page 588

Installing and configuring a new application (*System Administration Guide*)

## Documenting applications

You can document applications on a per-project or object basis by applying XSL style sheets to exported XML files. An XSL style sheet can filter and format the XML file into a browsable HTML document.

To get you started, there is an XSL style sheet sample. It does not cover all aspects of an application, nor is it targetted at a client programmer. You can extend or replace the sample with your own style sheet.

**RELATED CONCEPTS**

“XML browsing with XSL” on page 538

“The XSL sample” on page 541

**RELATED TASKS**

“Packaging applications” on page 574

“Browsing XML files” on page 539

---

## Container behavior

The following properties can be set for a container developed in Object Builder:

- “Deployment platforms” on page 423
- “Container service” on page 596
- “Container policies” on page 597
- “Behavior for Methods Called Outside a Transaction” on page 599
- “Behavior for Methods Called Outside a Session” on page 600
- “Connection” on page 601

**RELATED CONCEPTS**

“Container” on page 578

**RELATED TASKS**

“Creating a container instance” on page 578

## Container service

When you create a container, you can define the type of service it provides to its components. The service is defined in the Container Definition wizard, on the Service page. You have the following choices:

- Use no Object Services (page 596)
- Use Home Service (page 596)
- Use RDB Transaction Service (page 596)
- Use PAA Transaction Service (page 596)
- Use PAA Session Service (page 596)
- Use MQAA Transaction Service (page 596)

### **Use no Object Services**

Select this option if the container's components will have transient data only. Components with transient data can also use the RDB or PAA Transaction Service.

### **Use Home Service**

Select this option if the container will hold specialized homes.

### **Use RDB Transaction Service**

Select this option if the container's components have database persistence, and require the Transaction Services. If you do not select this option, then either the objects do not connect to a database, or the transaction support (commit and rollback) is provided in some other manner, for example by a client process, or by embedded SQL calls in the data object. You can also select this option to provide transaction support for components with transient data.

### **Use PAA Transaction Service**

Select this option if the container's components have persistence provided by a procedural adaptor, using the transaction services to provide secure persistence. You can also select this option to provide transaction support for components with transient data.

### **Use PAA Session Service**

Select this option if the container's components have persistence provided by a procedural adaptor, using session service to provide secure persistence.

### **Use MQAA Transaction Service**

Select this option if the container's components have persistence provided by a MQSeries application adaptor, using transaction service to provide secure persistence.

#### RELATED CONCEPTS

“Container” on page 578

Workload management (*Advanced Programming Guide*)

Transaction Service (*Advanced Programming Guide*)

Session Service (*Advanced Programming Guide*)

#### RELATED TASKS

“Creating a container instance” on page 578

“Editing a container instance” on page 883

## Container policies

When you create a container, you can define the termination and memory management policies it will implement for its components. The policies available depend on the type of container service defined. Both the service and the policies are set in the Container Definition wizard, on the Service page.

You have the following choices:

- Save all data when server is about to stop (page 597)
- Passivate a component after its data is saved (page 597)
- Enable persistent references (page 598)

### Save all data when the server is about to stop

This option is required if you selected **Use Home Service**, is optional if you selected **Use no Object Services**, and is not available in any other case.

If you check this option, the container stores all its in-memory objects when the server is about to stop. The container stores objects by calling their `checkpointToDatastore` framework method.

If you do **not** check this option, the container does not store object data when the server is about to stop. The data of in-memory objects may be lost when the server stops.

### Passivate a component after its data is saved

Select whether the container should passivate components not in use, or keep components in memory at all times.

If you check this option, objects are passivated when they are not in use. Passivation occurs as follows:

- If the container uses the Transaction Services (as specified above), an object is passivated when a transaction that involves the object is committed or rolled back.

- If the container does **not** use the Transaction Services (as specified above), an object is passivated when its framework method `checkpointToDatastore` is called (by the client, or by the container if the server stopped and the **Checkpoint the data** option is selected above).

If you do **not** check this option, objects remain in memory once they are activated. There may be performance benefits for this option: objects can be accessed more quickly, but take up memory even if they are accessed rarely. If the container components have transient data (that is, they have data objects with no datastore), then it may be necessary to select this option to avoid losing the objects' data.

### **Enable persistent references**

This option is only available if you select **Use no Object Services**. By default, object references are not made persistent when there are no object services enabled (the data is assumed to be transient), and instances held by the container are dropped when the server stops. Check this option if your components have persistent data, that you are accessing without the Object Services.

If you check this option, then an attempt to find a component in the container will first try looking in current memory, and if that fails, then try calling the `retrieve` method of the component's data object implementation. If the `retrieve` method does not throw an exception, the `retrieve` is assumed to be successful, and the container returns an object reference.

To force the `retrieve` method to fail for a particular data object (when, for example, there is no datastore to access), you can modify the code of the `retrieve` method to return exception `IBOIMException::IDataKeyNotFound`.

If you do **not** check this option, then an attempt to find a component in the container will succeed only if the component is currently in memory. The component will be in memory if it has been created and added to the container since the server was last started. Check this option if your components have only transient data.

### **RELATED CONCEPTS**

"Container" on page 578

Workload management (*Advanced Programming Guide*)

Transaction Service (*Advanced Programming Guide*)

Session Service (*Advanced Programming Guide*)

### **RELATED TASKS**

"Creating a container instance" on page 578

"Editing a container instance" on page 883

## Behavior for Methods Called Outside a Transaction

When you create a container that uses Transaction Service (by selecting either **Use RDB Transaction Service** (page 596), or **Use PAA Transaction Service** (page 596), or **Use MQAA Transaction Service** (page 596) on the Service page of the Container Definition wizard), you can set the way the container will handle methods called outside a transaction.

You have the following options:

- Start a new transaction and complete the call
- Throw an exception and abandon the call
- Ignore the condition and complete the call

### **Start a new transaction and complete the call**

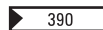
Click this option if components require the Transaction Services at all times. If a method is called outside of the scope of an existing transaction, the Transaction Services start a new transaction for that method, and commit the transaction when the method completes.

### **Throw an exception and abandon the call**

Click this option if components require the Transaction Services at all times, but you do not want transactions to be started automatically. If a method is called outside of the scope of an existing transaction, the object throws the exception `CORBA::TRANSACTION_REQUIRED`. If you select this option, transactions must be explicitly started and committed (for example, by the client that calls the method).

### **Ignore the condition and complete the call**

Click this option if some but not all components require the Transaction Services. If a method is called outside of the scope of an existing transaction, the method will complete without transaction support. While the Transaction Services will support components in this container, transactions must be explicitly started and committed when they are needed.



**Restriction:** When the development platform is OS/390, the **Use PAA Session Service** option is not available on the Service page of the Container wizard.



The **Use MQAA Transaction Service** option is available only on the Windows NT, Solaris and HP-UX deployment platforms.

### **RELATED CONCEPTS**

Application adaptor (*Programming Guide*)  
“Container” on page 578

“DDL” on page 137  
Session Service (*Advanced Programming Guide*)  
Transaction Service (*Advanced Programming Guide*)  
Connections to a tier-3 system (*System Management*)

#### **RELATED TASKS**

“Creating a component for PA data” on page 157  
“Working with container instances” on page 883

## **Behavior for Methods Called Outside a Session**

If you select **Use PAA Session Service** on the Service page of the Container Definition wizard, you can set the way the container will handle methods called outside a session. The options are available on the Service Details page of the Container Definition wizard.

You have the following options:

- Throw an exception and abandon the call
- Ignore the condition and complete the call

### **Throw an exception and abandon the call**

When a method is called outside of the scope of an existing session, and you select this option, the object throws the exception `CORBA::BAD_OPERATION` with a minor code of `0x49420330`, which indicates that a session is required. This is the sessional equivalent of `CORBA::TRANSACTION_REQUIRED`.

### **Ignore the condition and complete the call**

Select this option if some but not all components require Session Service. If a method is called outside of the scope of an existing session, the method will complete without session support. While Session Service will support components in this container, sessions must be explicitly started and committed when they are needed.

#### **RELATED CONCEPTS**

Application adaptor (*Programming Guide*)  
“Container” on page 578  
“DDL” on page 137  
Session Service (*Advanced Programming Guide*)  
Transaction Service (*Advanced Programming Guide*)  
Connections to a tier-3 system (*System Management*)

#### **RELATED TASKS**

“Creating a component for PA data” on page 157  
“Working with container instances” on page 883

## Connection

▶CHANGED

When you create a container that uses PAA services (by selecting either **Use PAA Transaction Service** (page 596) or **Use PAA Session Service** (page 596) on the Service page of the Container Definition wizard), you need to specify the details of the connection. You specify the connection details on the Service Details page of the Container wizard.

You have the following options:

- Connection Name
- Connector type used by session (HOD, ECI, SAP, Generic)

### Connection Name

Type a name for the connection (the machine name) to be used by the session, or the PAA transaction.

### Connector type used by session

You can select options from this section only if you selected **Use PAA Session Service** (page 596) on the previous page (the Services page) of this wizard.

Use this section to specify the connector type to be used by the session. You have one of the following options:

- HOD
- ECI
- SAP
- Generic

### HOD

Select this option if you want to use or reuse TN3270 communications provided by Host On Demand (HOD) to a tier-3 CICS region or IMS system.

### ECI Connection

Select this option if you want to use the External CICS Interface (ECI) communications provided by a CICS Common Client to connect to a tier-3 CICS region or IMS system.

### SAP Connection

Select this option if you want to connect to a 3-tier SAP system or group of SAP systems.

### Generic Connection

A generic connection is used to configure and manage a generic physical connection to a tier-3 system other than that defined by a HOD connection, an ECI connection, a SAP connection, or a RDB connection.

#### RELATED CONCEPTS

Application adaptor (*Programming Guide*)

“Container” on page 578

“DDL” on page 137

Session Service (*Advanced Programming Guide*)

Transaction Service (*Advanced Programming Guide*)

Connections to a tier-3 system (*System Management*)

#### RELATED TASKS

“Creating a component for PA data” on page 157

“Working with container instances” on page 883

---

## Application DDL files

### ▶ CHANGED

The installation package for an application family contains a **DDL file** that describes the contents of the application family. It describes the applications in the family and the objects, attributes, and relationships that make up each application. For example, the DDL file for an application family has the following definitions:

- The applications to run on servers, and their relationships to objects that they provide
- The applications to run on clients, and their relationships to objects that they provide
- The classes, DLLs, homes, containers, and other objects provided by the applications, and appropriate relationships between such objects
- Appropriate attributes of the applications, and other objects in the application family

The application family installation program uses the information in the DDL file to create *Install objects* that the System Manager can use to define and configure the applications.

When you use Object Builder to create an application family, it generates a DDL file for the application family along with a backup version.

▶ WIN ▶ AIX ▶ 390 ▶ SOLARIS ▶ HP-UX Object Builder generates the DDL file on each deployed platform. (Object Builder is capable of generating the file on all platforms - Windows NT, AIX, OS/390, Solaris, and HP-UX). You will need to load this file into System Management when you deploy your applications.

For example, if the name of your application family is MyAppFam, and you have specified the name of the DDL file as AppFam, the DDL emitter will



generate the following files:

`Working\platform\output_directory\MyAppFam\AppFam.ddl`

`Working\platform\output_directory\MyAppFam\backup\AppFam.auto.ddl`

*platform* is one or more of NT, AIX, Solaris, 390, or HP/UX (the platforms on which you are deploying your application).

*output\_directory* is each of NOOPT, PRODUCTION, TRACE, TRACE\_DEBUG (that is, for a given platform, you get four sets of DDL files generated).

Use the DDL files in a particular directory to load the equivalent version of the application libraries. For example, `Working\NT\TRACE\AppFamily\*.ddl` can be used to deploy the trace-enabled version of the application family.



If you have developed your code for OS/390, the generated DDL for the application family includes the statement:

```
targetPlatform="390";
```

This statement prevents the application family from being accidentally installed on an incompatible System Management platform.

You do not normally change DDL files after the application family has been installed into Component Broker. When you load an application family into Component Broker, each application in the DDL file is represented as an **available application** through the System Manager user interface. If you need to customize the application within Component Broker, you normally do so by changing model objects for the application through the System Manager user interface.

#### RELATED CONCEPTS

“Application DDL files” on page 602

“Creating DDL files”

#### RELATED TASKS

Creating an application family

“Packaging applications” on page 574

“Generating the DDL files” on page 593

## Creating DDL files



When you use Object Builder to create an application family, it generates a DDL file for the application family. This generated DDL file is found in a subdirectory of your working directory that has the same name as the application family. That is, in

`Working\platform\output_directory\app_family_name`. A backup DDL file is also created in the backup directory within this directory.

**Note:** *platform* is one or more of NT, AIX, Solaris, 390, or HPUX (the platforms on which you are deploying your application).  
*output\_directory* is each of NOOPT, PRODUCTION, TRACE, TRACE\_DEBUG (that is, for a given platform, you get four sets of DDL files generated).

When you use Object Builder to add objects to the application family package, it adds entries for those objects into the DDL file with appropriate attributes and relationships.

Before you generate the the DDL for an application family, you can add additional files to an application, or change your configured managed objects. Such objects are sometimes needed to configure special application functions; for example, when packaging an application family for a controlled server group, you can add policy groups, bind policies and their associated C++ classes.

You do not normally change DDL files after the application family has been installed into Component Broker. When you load an application family into Component Broker, each application in the DDL file is represented as an **available application** through the System Manager user interface. If you need to customize the application within Component Broker, you normally do so by changing model objects for the application through the System Manager user interface.

#### **RELATED CONCEPTS**

“Application DDL files” on page 602

#### **RELATED TASKS**

“Generating the DDL files” on page 593

## **The structure of a DDL file**



This topic describes the general internal structure of an application DDL file. It is intended as a review aid in case you need to look within a DDL file after it has been generated by Object Builder (using a text editor, in either review or debug mode).

**Recommendation:** You must not directly edit DDL files after the application family has been installed into Component Broker. When you load an application family into Component Broker, each application in the DDL file is represented as an available application through the System Manager user interface. If you need to customize the application within Component Broker, you can change model objects for the application through the System Manager user interface.

The following description of the DDL file structure is based on an extract of the Insurance.ddl file provided with Component Broker. The application

family defined within the DDL file is referred to as 'your application family'. Other application families provided by Component Broker are referred to by name. Some lines have been missed out, and replaced with ellipsis (...), where they do not add any significant value to the description.

### Declaration of objects supplied by Component Broker that are used by your application family

At the top of the DDL file is a set of lines that declare the objects supplied by Component Broker that are used by your application family. Most, if not all, of these objects are defined in the **iDefaultApplications** application family provided by Component Broker. You should not change the definitions for these objects.

```
//*****
// Top of DDL file
//*****
//Pre-Declare objects used by the Application which are Supplied by CB
//*****
ApplicationFamily.iDefaultApplications;
ApplicationFamily.iDefaultApplications/Dll.somib1li; ...
ApplicationFamily.iDefaultApplications/
ManagedObjectClass.IBOIMManagedObject_IViewCollectionImpl;
```

### Declaring objects

Objects within a DDL file are identified by their class and object name, in the following format:

```
object_class.object name;
```

If an object exists in a different DDL file, the name is prefixed with the name of the other application family where the object is defined; for example, *ApplicationFamily.iDefaultApplications/Dll.somib1li*. Note that the DDL file name is joined to the object name by a forward slash character (/) and each declaration line ends in a semi-colon (;). Several declarations, separated by commas, can be grouped on the same line.

### Object names

If the name of an object is to contain embedded blanks or any of the following characters, the name must be enclosed in double quotation marks:

```
{ } , ; . /
```

(Open brace, close brace, comma, semi-colon, period, and forward slash.) Otherwise, the use of double quotation marks is optional.

▶ 390 The OS/390 interpreter does not allow blanks, even when in quotation marks.

The System Manager expands the name of a DLL object into a fully qualified path name when it creates the corresponding DLL Image. The DLL object name is prefixed with the install path for the application family, and has the file type (.dll or .a) appended. For example, for the DLL object **myappinit** and its application family installed in **c:\Cbroker\appfamily\** on Windows NT, the name of the DLL Image becomes **c:\Cbroker\appfamily\myappinit.dll**.

### Definition of your application family

A DDL file defines one application family only. Everything about that application family is contained within the definition of that application family, which is delimited by the **ApplicationFamily.family\_name** statement and its opening and closing braces { ... }. When you use Object Builder to create an application family, it creates this definition. Normally, your application family and all its objects have the same value for the **version** attribute.

```
// Describe the application family named "Insurance".
ApplicationFamily."Insurance"
{ // Set the attributes of the application family.
description = ""; version = "1.0.0"; ... }
//*****
// Bottom of DDL file
//*****
```

### Object attributes

All object attribute statements have the general form **attribute\_name = value;**

Text string values must be enclosed in double quotes. If an attribute has several values (at the same time), the sequence of values is enclosed within braces and each value separated by commas; for example,  
{value1,value2,value3}

When needed, attribute statements are created automatically by Object Builder. Other attributes do not need to be defined in a DDL file, and are left to assume their default values.

### Forward declaration of objects that are needed later

At the top of your application family definition is a set of entries that declare the objects that are defined later within your application family. For each entry, the object class and name must match its later definition.

```
Foward declarations of objects which will be needed later.
Xarm."LifeIns"; MappedType.BCPBO_csClaimBOBO_DO; ...
Container.InsuranceContainer;
```

### Definition of objects within your application family

Within your application family definition there are separate definitions for all the objects of the family, as declared at the top of your application family definition.

All these object definitions are at the same level within your application family definition, and have the same general format, as shown below:

```
// Define the Xarm image
for "LifeIns". Xarm."LifeIns" { openString = "LifeIns";
switchLoadFile = "db2s1f"; }
```

An object definition is delimited by its **class\_name.object\_name** statement and its opening and closing braces { ... }.

### Definition of applications within your application family

An application within an application family is defined like any other object. A server application is delimited by its **Application.application\_name** statement and its opening and closing braces { ... }. A client application is delimited in the same way by its **ClientApplication.application\_name** statement and braces.

Within the braces are statements that define appropriate attributes of the application and the “provides” relationships to objects that the application provides.

```
// Define applications.
Application."LifeInsObjects"
{ // Set the attributes of the application.
description = ""; version = "1.0.0"; runControl = stop;
requiredJavaVMName = ""; ProvidesXarm -> { Xarm."LifeIns" };
ProvidesManagedObjectClass -> { ManagedObjectClass."BCPMO_csClaimMO",
ManagedObjectClass."BCPMO_csPayoutFractionMO", ...
ManagedObjectClass."A_C_ModuleMO_csAgentMO" }; ... }
// End definition of application LifeInsObjects.
```

### Application “provides” relationships

For each non-application object within your application family there should be a “provides” relationship with at least one application. (An object can have “provides” relationships with more than one application.) These relationships are created automatically by Object Builder.

The “provides” relationships have the same form:

```
Providesrelationship_name -> {
 object_class.object_name };
```

The “arrow” (->) indicates a forward relationship to the object that the application provides. If an application provides several objects of the same class, the sequence of object identifiers is enclosed within braces and each identifier separated by a comma.

### Other relationships between objects

Some objects need to contain relationships with other objects. A relationship should exist in only one of the two related objects. (If an object in your

application family needs a relationship to an object in the default application family, it must be defined in the object in your DDL file.) These relationships are created automatically by Object Builder.

The relationships of an object are normally listed after the objects attributes; for example:

```
// Define the MO class 'BCPMO_csClaimMO'.
ManagedObjectClass."BCPMO_csClaimMO"
{ // Define the attributes. description =
 "Description of the class named csClaim."; ... interfaceName =
 "BCP::csClaim";
// Define the relationships. This defines the DLLs containing
//this class's information. ContainsManagedObjectImplementation
<- dll.b_s; containsmanagedobjectkeyimplementation
<- dll.b_c; containsmanagedobjectcopyhelperimplementation
<- dll.b_c; }
```

The direction of the relationship, defined by the arrow (<- or ->), must be appropriate for the object that the relationship is defined in. You can get a clue about the correct direction from the relationship name: relationship names that begin with 'Uses' or 'Provides' are forward relationships (->); relationship names that begin with starting 'Contains' or 'Collects' are backward relationships (<-).

For example, the relationships for a home define the managed object class, data object class, and container used by the home (as forward relationships). It also defines the home provided by Component Broker that 'collects' this home (as a backward relationship) and the home of view objects provided by Component Broker that this home uses (as forward relationships).

```
// Define a home for the "LifeInsObjects_BCPMO
csClaimMO_BCPDOImpl_csClaimDOImpl" class. Home."LifeInsObjects_BCPMO
csClaimMO_BCPDOImpl_csClaimDOImpl"
{
// Define the attributes. ...
// Define the relationships.
UsesManagedObjectClass ->
 ManagedObjectClass.BCPMO_csClaimMO;
UsesDataObjectClass ->
 DataObjectClass."LifeInsObjects_BCPDOImpl_csClaimDOImpl";
UsesContainer ->
 Container.InsuranceContainer;
CollectsHome <-
 applicationfamily.idefaultapplications/home.iboimhomeofregqihomes;
homeofviews ->
 ApplicationFamily.iDefaultApplications/Home.iBOIMViewCollection;
}
```

#### RELATED CONCEPTS

"Application DDL files" on page 602

"Creating DDL files" on page 603

**RELATED TASKS**

“Generating the DDL files” on page 593

**RELATED REFERENCES**

“Internationalization of data” on page 132

“Naming objects” on page 128





---

## Chapter 13. Testing applications with QuickTest

Once you have deployed your application on the server, you can test it with QuickTest. Object Builder can generate a simple QuickTest client program, based on the QuickTest Java framework, which you can use to perform queries and run methods on your components. This enables you to test your business logic without going through the steps of writing a complete client application.

1. "Generating QuickTest client applications" on page 614
2. "Running QuickTest client applications" on page 615
3. "Testing cross-project applications with QuickTest" on page 489
4. "Recording QuickScript" on page 617
5. "Compiling the QuickScript file" on page 618
6. "Running QuickScript" on page 618
7. "Running the QuickTest tutorial" on page 647

### RELATED TASKS

"Chapter 4. Creating a component" on page 127

### RELATED CONCEPTS

"QuickTest"

"QuickScript" on page 616

"Building for QuickTest" on page 564

---

## QuickTest

Component Broker provides a suite of server application development tools that enables you to quickly generate server application code without knowing the programming model. QuickTest is a Java framework for creating simple Java clients that have a standard GUI (graphical user interface). The Java clients can be generated by Object Builder and used to quickly test your server applications. QuickTest requires no technical expertise or knowledge of the Component Broker programming model. You can also use QuickTest to record your test steps in a QuickScript file. Subsequently, you can invoke QuickTest to *playback* the QuickScript for repetitive testing.

QuickTest provides a 100% pure Java application framework. The QuickTest framework Java source is shipped with Component Broker in the samples directory. The generated QuickTest Java application allows you to access the

Component Broker client programming model and exercise the server objects. You can, for example, create managed objects and invoke the methods that have been defined for that object.

Here is a summary of some of the features of QuickTest:

- **Server code testing**  
In large corporations, the individuals responsible for the server business logic implementation may not necessarily develop the client GUI code. Although the server application can be compiled, client code is required to test it. With QuickTest, a visual Java client application can be created by Object Builder to let you test the business logic of the server code.
- **Run-time problem diagnosis**  
QuickTest-generated client code can be saved to help identify run-time problems of client/server applications. Often, it is not clear if a run-time problem is the result of the client or the server implementation because of the complexities of client/server applications. With QuickTest, the generated client code, which was used at application development time can be reused to narrow the scope of the run-time problem.
- **Education**  
The QuickTest framework (which includes source code) together with the generated Java client code demonstrates component reuse and is an example of implementing the Component Broker programming model.
- **Regression testing**  
QuickTest can record every transaction that is initiated with its client application. The transactions are saved as QuickScript files which can be compiled and played back at a later time for regression testing.
- **Long-running tests**  
With QuickScript, tests can be assembled and placed in long-running loops to stress test the system. Since each QuickScript runs in its own process, multiple processes of the same QuickScript can be exercised in combination with other QuickScripts to simulate deployment environments.
- **Visual demonstrations**  
QuickScripts can be run in an active mode to visually display processing as it occurs. This can be used to demonstrate the functionality of server applications.

The following features are supported by QuickTest:

- Transaction and session processing
- Use of attributes of the following types: boolean, char, octet, string, short, unsigned short, long, unsigned long, float and double, enumeration, sequences, structures, *any*, and multi-dimensional arrays
- In, out, and inout parameter support for methods and object types
- Enumeration support with names of enumerated values
- Character length restriction support on data entry

- Key processing to assist with data entry fields that only allow numeric values
- UUID support
- Method invocations with parameter types as described above
- Processing of return values from methods
- Availability of object return values from methods for copy and paste actions
- Pasting of object parameters to methods
- Copying of selected objects as the parameters of methods
- Copying and pasting of iterator return values from methods to the tester iterator
- Automatic copying of iterators to transient reference collections so that they can be used outside the scope of the transaction
- Creation of objects using keys (**Add by Key**)
- Creation of objects using copy helpers (**Add by Copy**)
- Updating objects (**Options > Key with update** from the Test Cases window)
- Deletion of objects
- Iteration on queryable homes
- Evaluation (evaluate()) on queryable homes using OOSQL predicates
- Finding objects by primary key
- Multithreaded transaction processing
- Specialized homes that inherit from IHome or IQueryableIterableHome
- Savings settings for the home, port, and timeout values
- Saving properties for persistence, scoping, window positions, and whether the objects reside in queryable homes
- Inheritance of attributes and methods
- Inclusion of Java code snippets in each generated bean during generation time, in the following forms:
  - Import statements
  - Declarations
  - Method modifications at the beginning and end of the bean definition class
  - New methods
- Invocation of customized scripts in Java by the framework (No knowledge of CORBA or the Component Broker Programming Model is required.)
- Query evaluators with data arrays
- Security using DCE and SSL
- Reference collection support (either transient or relational database-backed)

- Queries over reference collection (**View > Reference Collection** from the Test Cases window)

**Note:** QuickTest does not support the union type.

#### RELATED TASKS

- “Generating QuickTest client applications”
- “Running QuickTest client applications” on page 615
- “Recording QuickScript” on page 617
- “Compiling the QuickScript file” on page 618
- “Running QuickScript” on page 618
- “Running the QuickTest tutorial” on page 647

#### RELATED CONCEPTS

- “QuickScript” on page 616

#### RELATED REFERENCES

- “The QuickTest framework” on page 620
- “QuickTest-generated files” on page 647

## Generating QuickTest client applications

You can generate a QuickTest client application that you can use to test your application code. First define the client and server DLLs for your application, and then build them.

To generate a QuickTest client once you have built your DLLs, follow these steps:

1. Select the Build Configuration folder in the Tasks and Objects pane.
2. From its pop-up menu, select **Generate > All > QuickTest Default Targets**.

This generates the makefile that will be used to build or remove the QuickTest client, and the batch file that will be used to run the QuickTest client:

#### Makefile:

Working\<<platform>\<DLLname>.QT.mak

#### Batch file:

 Working\<<platform>\<config>\qt.bat

   Working/<<platform>/<config>/qt.ksh

#### RELATED TASKS

- “Running QuickTest client applications” on page 615

“Running the QuickTest tutorial” on page 647

#### RELATED CONCEPTS

“QuickTest” on page 611

#### RELATED REFERENCES

“The QuickTest framework” on page 620

“QuickTest-generated files” on page 647

## Running QuickTest client applications

Once you have defined and built your DLLs, and generated the makefile for your QuickTest client, you can build and run the QuickTest client application to test your application code.

To build the QuickTest client, follow these steps:

1. Select the Build Configuration folder
2. From its pop-up menu, select **Build > Out-of-Date Targets > QuickTest**

The makefile for your QuickTest client is built. You can follow the progress of the build in the Command window.

To rebuild your entire application including QuickTest in a single step:

1. Click **File > Preferences** to open the Object Builder Preferences notebook.
2. Expand the Tasks and Objects folder.
3. Click the Makefile Generation node.
4. In the right-hand pane, select the **Add QuickTest target to “all” target** option.
5. Click **OK** to close the notebook and apply your changes.
6. From the pop-up menu of the Build Configuration folder, select **Generate > All > All Targets**.
7. From the pop-up menu of the Build Configuration folder, select **Build > All Targets**.

The source of the QuickTest-generated .java files is in the Working\*<platform>*\QT directory. When these files are compiled, the resultant JAR file is located under the target configuration directory: Working\*<platform>*\*<config>*\QT. For example, if Production is the target configuration directory, then the files are located in Working\*<platform>*\PRODUCTION\QT.

Once the build is complete, you can run the QuickTest client application. You can run the client application in two ways:

- Within Object Builder:
  1. From the Build Configuration folder’s pop-up menu, click **Build > Run QuickTest**.

- Outside Object Builder:
  1. Go to the project's `\Working\<platform>\<config>` directory.
  2. Type the command `qt`.

For example, if the platform is Windows NT with Production configuration, then you must change directory to `Working\NT\PRODUCTION`, before you run the `qt` command.

#### **RELATED TASKS**

"Testing cross-project applications with QuickTest" on page 489

"Generating QuickTest client applications" on page 614

"Recording QuickScript" on page 617

"Running the QuickTest tutorial" on page 647

#### **RELATED CONCEPTS**

"QuickTest" on page 611

#### **RELATED REFERENCES**

"The QuickTest framework" on page 620

"QuickTest-generated files" on page 647

---

## **QuickScript**

When you are running a QuickTest client application, you can record your test steps. When the client application terminates, it saves a QuickScript Java file, which contains the recorded test steps. QuickTest can compile and then playback the QuickScript file for repetitive testing. The saved QuickScript file is `lastQuickTest.java`.

The code in the QuickScript file accesses the classes that were generated in QuickTest's build step. The recorded actions are the events that occurred during the running of the QuickTest client application.

Individual elementary events like keystrokes and mouse movements are not recorded; but the state of the data for Adds, Updates, Deletes, and the invoked query statements are recorded. Replaying QuickScript events does not verify that the data processed during the initial recording and replay session is identical. But, it does verify that the activities successfully completed during replay.

#### **RELATED TASKS**

"Recording QuickScript" on page 617

"Compiling the QuickScript file" on page 618

"Running QuickScript" on page 618

"Running the QuickTest tutorial" on page 647

#### RELATED CONCEPTS

“QuickTest” on page 611

#### RELATED REFERENCES

“The QuickTest framework” on page 620

“QuickTest-generated files” on page 647

## Recording QuickScript

By default, the QuickTest client application records test events in a QuickScript file. This .java file is saved automatically at intervals while the client is running, and finally saved when the client program exits. The script can be cleared at any time. You can verify that the recording is active by clicking on the **File** menu in the QuickTest Main window to see if **Record Script** has been selected. If **Record Script** is not selected, you can restart the recording. Follow these steps:

1. Go to the QuickTest main window.
2. From the **File** menu, select **Record Script**.

Whenever you toggle **Record Script** off, the contents of the QuickScript file are cleared.

By default, the QuickScript file is saved when the QuickTest client application terminates. However, if you wish to record only a part of the test steps, you should save the QuickScript file using these steps:

1. In the QuickTest Main window, select **File > Save Script**.
2. Specify a name, and path for the saved file.

Once you have saved the QuickScript file, you can toggle **Record Script** off to clear the previously recorded events, and stop the recording. If you want to continue to record events in a new QuickScript file, you must clear the contents of the QuickScript file after it has been saved.

When you are recording test steps during the execution of a QuickTest client application, you can restart the recording with a clean file by following this step:

In the QuickTest Main window, select **File > Clear Script**.

All the steps that have been recorded are erased. Recording continues in a clean QuickScript file.

#### RELATED TASKS

“Running QuickTest client applications” on page 615

“Compiling the QuickScript file” on page 618

“Running QuickScript” on page 618

“Running the QuickTest tutorial” on page 647

**RELATED CONCEPTS**

“QuickTest” on page 611

“QuickScript” on page 616

## Compiling the QuickScript file

Once a QuickScript file has been saved, it can be compiled and run at a later time. QuickTest is used to compile the QuickScript .java file, as follows:

Follow these steps in the QuickTest Main window:

1. Start the project the QuickTest client was generated for.
2. From the Build Configuration window, click **Build > Run QuickTest** to start the QuickTest client. The QuickTest Main window opens.
3. From the **File** menu, select **Load Script**.
4. Select the QuickScript file that you want to compile. QuickTest compiles the QuickScript file.

A window with the compilation status appears.

The end result of compilation is a QuickScript file, which is a class file (extension .class). This class inherits from a Script class within the QuickTest framework. The Script class is abstract; the script is concrete. So when the framework tries to execute the Script class, it actually works with the new script class.

If there is an error, an error output file is generated and saved to the same directory as the QuickScript file. You can view this error file and modify the QuickScript file. If there are errors with the QuickScript compilation, you can contact IBM Support to resolve the errors.

**RELATED TASKS**

“Running QuickTest client applications” on page 615

“Recording QuickScript” on page 617

“Running QuickScript”

“Running the QuickTest tutorial” on page 647

**RELATED CONCEPTS**

“QuickTest” on page 611

“QuickScript” on page 616

## Running QuickScript

After you successfully compile the QuickScript .java file, you can run QuickScript to repeat the testing that you recorded when you ran the QuickTest client application.



When you run QuickScript, the Controller window appears, where you can select QuickScript's execution settings.

In the QuickScript Controller window, set the following entries on the right side of the window:

- **Number of Loops**  
Specifies the number of times the QuickScript application will loop. This enables you to stress test the system.
- **Sleep Duration**  
Specifies the sleep duration in milliseconds (5000 ms = 5 seconds). This value can be modified at any time. The new value is used for the next statement which has **Sleep** selected.

The remaining fields at the right side of the window show the status of the QuickScript application as it runs:

- **Current Loop**  
Displays the current loop count when QuickScript runs.
- **Start Time**  
When you click **Start**, this value is updated to show the time the QuickScript execution started.
- **End Time**  
When the number of loops has been completed, this value is updated with the time the execution completed.

The table at the left side of the window lists all the actions that were recorded while running the QuickTest client application. Each action is recorded as a statement (row) in the table. The table has the following columns:

- **Stop**  
When you select this check box for a row, and the value of **After** for that row is greater than the **Current Loop** count, QuickScript stops before executing this statement. To continue its execution, click **Continue**. For example, if the statement at row 15 has **Stop** selected, and the **After** value is 15, then whenever the **Current Loop** count is greater than 15, QuickScript will stop before the execution of this statement. To disable the stop action before this statement, clear the **Stop** check box, or change the value of **After** to a number less than the **Current Loop** count.
- **After**  
This value is compared with the **Current Loop** to determine if QuickScript should pause before the execution of this statement. This value is used only when **Stop** has been selected for the same row.
- **Sleep**  
When selected, QuickScript will pause at that statement for the amount of time specified in the **Sleep Duration** field. The entire **Sleep** column can be toggled on or off by clicking the title of the column: **Sleep**.

- **Row**  
This is the statement number in the QuickScript file. When QuickScript stops at a statement, the command console displays a statement number indicating the row that is currently stopped. You can use this number to find the row which has the active Stop statement. This becomes very useful when you have a large number of statements in this table and several of them have **Stop** selected.
- **Statements**  
These are the QuickScript statements that were recorded when the QuickTest client application was executing.

At the right side of the window, you can use the following buttons to control QuickScript processing:

- **Start**  
To begin the execution of QuickScript. Be sure to start the testing from a known status of your data. For example, you may want to ensure an empty database before you start testing.
- **Continue**  
To continue QuickScript execution, after it has stopped at a statement, which you indicated had to halt its execution (a statement for which the **Stop** check box is selected).

#### **RELATED TASKS**

“Running QuickTest client applications” on page 615

“Recording QuickScript” on page 617

“Compiling the QuickScript file” on page 618

“Running the QuickTest tutorial” on page 647

#### **RELATED CONCEPTS**

“QuickTest” on page 611

“QuickScript” on page 616

---

## **The QuickTest framework**



This section describes the object model for the QuickTest framework. The Java classes that are emitted during the build process inherit from some of these classes. Objects that are defined as Interface inherit from TestBean; those that are identified as Copy inherit from TestBeanCopy; and those identified as Key inherit from TestBeanKey.

The Test Case presents the TestBean derived classes on the left side of the Test Case window, while the Home behavior is presented through the Tester. Each TestBean derived class is *wired* to an instance of a Tester.



“Recording QuickScript” on page 617  
 “Running the QuickTest tutorial” on page 647

**RELATED CONCEPTS**

“QuickTest” on page 611  
 “QuickScript” on page 616

**RELATED REFERENCES**

“QuickTest-generated files” on page 647

**QuickTest with the Component Broker Programming Model**

The following table provides information for assisting in locating code snippets within the QuickTest framework. The Java code for QuickTest is located in `samples\Tutorial\QuickTest\framework\com\ibm\quicktest\framework` (under the Object Builder install directory).

How do I	Answer	Example QuickTest Java Source
Get a queryable, iterable home	Use the <code>factoryFinder</code> to find the factory using the object interface name. Then narrow the returned object using the <code>IQueryableIterableHomeHelper</code> .  Uses package: <code>com.ibm.IManagedAdvancedClient.IQueryableIterableHomeHelper</code>	<code>CBGeneral.get_QI_home</code>
Get a regular home	Use the <code>factoryFinder</code> and find the factory using the object interface name. Then narrow the returned object using the <code>IQueryableIterableHomeHelper</code> .  Uses package: <code>com.ibm.IManagedClient.IHomeHelper</code>	<code>CBGeneral.get_home</code>
Get transaction or session and set the timeout value	The ORB from <code>CBSeriesGlobal</code> has a method, <code>get_current</code> which takes the string parameter for either “ <code>CosTransactions::Current</code> ” or “ <code>ISessions::Current</code> ”. The returned ORB is then narrowed to the current transaction or session, after which the timeout can be set.  Uses packages: <code>com.ibm.CBCUtil.CBSeriesGlobal</code> <code>org.omg.CosTransactions.Current</code> <code>com.ibm.ISessions.Current</code>	<code>CBGeneral.setCurrentTransaction</code>

How do I	Answer	Example QuickTest Java Source
Initialize the ORB as an applet or application	<p>Use a Properties object, or pass arguments to the Initialize method. The Properties object is used when you run the ORB as an application while the args[] parameter is used when you run it as an applet.</p> <p>Uses package: com.ibm.CBCUtil.CBSeriesGlobal.Initialize</p>	<p>CBGeneral.     resolve_with_string</p>
Set DCE Security	<p>When using Properties, use props.put ("com.ibm.CORBA.EnableDCESecurity", "true").</p> <p>When using the args[] parameter, use args[x] = "-ORBEnableDCESecurity"; args[x+1] = "true";</p> <p>Uses package: com.ibm.CBCUtil.CBSeriesGlobal.Initialize</p>	<p>CBGeneral.     resolve_with_string</p>
Start a transaction	<p>Depending on the container, either use the begin() method on the transaction, or the beginSession() operation on the session.</p> <p>Uses package: org.omg.CosTransactions com.ibm.ISessions</p>	<p>CBGeneral.begin</p>
Commit a transaction or session	<p>Depending on the container, use the commit() method on the transaction, and endSession() operation on the session.</p> <p><b>Note the following points:</b></p> <ul style="list-style-type: none"> <li>• If you have an atomic container, you do not have to use transaction semantics. They are automatically done for you.</li> <li>• An EndMode value needs to be set for the session. This enumeration indicates the type of end-session processing that should be performed. For a list of possible values for EndMode, see ISessions in the Sessions Service.</li> </ul> <p>Uses package: org.omg.CosTransactions com.ibm.ISessions</p>	<p>CBGeneral.commit</p>

How do I	Answer	Example QuickTest Java Source
Rollback a transaction or session	<p>Depending on the container, use the rollback() operation on the transaction and endSession() operation on the session.</p> <p><b>Note:</b> An EndMode value must be set for the session.</p> <p>Uses package: org.omg.CosTransactions com.ibm.ISessions</p>	CBGeneral.rollback
Identify the factory for a transient or a RDB reference collection	<p>For a transient reference collection, find the factory or home for:</p> <p>IManagedCollections::   IReferenceCollection.object interface/   TransReferenceCollectionFactory.object home</p> <p>For a relational database (RDB), use:</p> <p>IManagedCollections::   IReferenceCollection.object interface or   DB2ReferenceCollectionFactory.object home.</p> <p>Then narrow the result with ICollectionHomeHelper.narrow().</p> <p>Uses package: com.ibm.IManagedCollections</p>	RefColPanel.createRefCol
Create the collection	<p>Once the factory is found, use the createCollectionFor() method and provide the parameter of the IDL tag for the name that is to be contained within the collection. The form of the IDL tag is "IDL::HomeName/Object:1.0"</p> <p>Uses package: com.ibm.IManagedCollections</p>	RefColPanel.createRefCol

How do I	Answer	Example QuickTest Java Source
Create and process an iterator on the reference collection	<p>The iterator is obtained from the reference collection using the createIterator() method. The iterator is processed using the more() method and the next() method to retrieve an element.</p> <p>The iterator returns IManageable objects, which must be narrowed to the appropriate object.</p> <p>Uses packages:  com.ibm.ICollectionsBase  com.ibm.IManagedCollections  com.ibm.IManagedClient.IManageable</p>	RefColPanel.createRefCol
Query over a reference collection	<p>This is a typical query evaluator implementation except for the use of a ParameterListBuilder (PLB) to identify the reference collection. First create a PLB using the PLBHelper and identify the reference collection using the add_object_parm() method by providing a name and a pointer to the reference collection. Then create a NVPair[] from the PLB with the method get_parm_list().</p> <p>Uses package:  com.ibm.IExtendedQuery  org.omg.CosQueryCollection</p>	RefColPanel.createRefCol
Create a query evaluator	<p>Find the factory for "host/resources/servers/", the server as defined in System Management, and "/query-evaluators/default". Then narrow the object with QueryEvaluatorHelper.narrow.</p> <p>Uses package:  com.ibm.IExtendedQuery</p>	QueryPanel.doQuery

How do I	Answer	Example QuickTest Java Source
Create, remove, and populate a query data array iterator	<p>Once a data array iterator holder (DataArrayIteratorHolder) has been created, and before it reuses the value, a data array iterator (DataArrayIterator) must be removed from the server. The data array iterator is then populated with the evaluate_to_data_array() method on the query evaluator.</p> <p>Uses package: com.ibm.IExtendedQuery</p>	QueryPanel.doQuery
Find out how many fields are contained in the data array iterator	<p>Use the method get_number_of_fields() on the data array iterator.</p> <p>Uses package: com.ibm.IExtendedQuery</p>	QueryPanel.doQuery
Find out the names of the fields in the data array	<p>Use the method get_field_name() for each field in the data array iterator.</p> <p>Uses package: com.ibm.IExtendedQuery</p>	QueryPanel.doQuery
Get access to the field values in the data array	<p>Use the method next_one() on the data array iterator with the parameter being a DataArrayHolder. This is the equivalent of retrieving one row of data. The DataArrayHolder now contains all the columns for the single row. Next process the DataArrayHolder's value (which is an array). Since each element is a CORBA:any, the type().kind() will indicate the org.omg.CORBA.TCKind. Once the kind is determined, the appropriate value can be extracted from the <i>any</i>.</p> <p>Uses package: com.ibm.IExtendedQuery</p>	QueryPanel.doQuery
Process an iterator	<p>The iterator is processed using the more() and next() methods. The object that is returned from the next() method is of type CORBA and must be converted to a managed object.</p> <p>Uses package: com.ibm.ICollectionsBase.IIterator</p>	Tester.doPasteIterator



How do I	Answer	Example QuickTest Java Source
Access attributes from a CORBA object	<p>When the CORBA object is available, it must be narrowed to the class of the particular type of the processed application. For QuickTest, look at the generated code for the display method to see how the object is narrowed appropriately. Then follow the code to the object's display() method to see the attributes retrieved from the narrowed object. With QuickTest, the message is sent from the Tester to the TestBean, and the TestBean extracts the attribute values and sends a message back to the Tester containing the attribute value array for display within the table.</p> <p>Uses package: CORBA.Object_fooBean.display(TesterEvent e)</p>	Tester.doPasteIterator
Process through an iterator multiple times	<p>The iterator is positioned at the end after being processed once. To reprocess the same iterator means to reposition the pointer to the beginning. This is accomplished with the reset() method.</p> <p>Uses package: com.ibm.ICollectionsBase.IIterator</p>	Tester.doPasteIterator
Find a CORBA object	<p>Using the appropriate home, use the byte string that represents the object being searched and then invoke the findByPrimaryKeyString method.</p> <p><b>Note:</b>QuickTest responds to either the button or the script by first sending a message to the TestBean so that a byte string can be created.</p> <p>The TestBean sends a message back to Tester and the result is stored in the variable keyString. See the Tester.testResponseEvents method for information on where the keyString is set.</p> <p>Uses package: com.ibm.IManagedAdvancedClient com.ibm.IManagedClient</p>	Tester.find

How do I	Answer	Example QuickTest Java Source
Determine whether two CORBA objects are identical	<p>Use the <code>_is_equivalent()</code> method on one object with the other as a parameter.</p> <p>Uses package: <code>org.omg.CORBA.Object</code></p>	Tester.haveIdentical
Create a managed object using a primary key	<p>Use the <code>keyString</code> provided by the <code>fooKeyHelper</code> to invoke the <code>createFromPrimaryKeyString</code> against the appropriate home.</p> <p>Uses package: <code>com.ibm.IManagedAdvancedClient</code> <code>com.ibm.IManagedClient</code></p>	Tester.key

How do I	Answer	Example QuickTest Java Source
<p>Create a keyString byte string</p>	<p>When you run <i>make</i> for each object in the project, the Java client bindings are generated, and one of the files is <code>fooKeyHelper.java</code>. The use of <code>fooKeyHelper</code> is demonstrated in the QuickTest-emitted code in the <code>_fooKeyBean</code> class where the <code>_create()</code> method is invoked to create the <code>IPrimaryKey</code> subclass. Each of the attributes that make up the key are then set, and the <code>_toString()</code> method can be invoked to obtain the byte string.</p> <p><b>Note:</b> QuickTest performs both operations (add by primary key, and update) during a single transaction. At the completion of the primary key creation, the Proxy screen (left side of the screen) responds to an update message.</p> <p>The Proxy screen holds only one instance of a class. But, the right side of the screen, which is called the Tester, holds a collection. The Proxy screen has all the attributes of the object, some of which may be identified as belonging to a key. All the attributes can be entered on the screen.</p> <p>When the <i>add by key</i> operation is performed, only those attributes pertaining to the key are used to build the key. QuickTest immediately requests an update, ensuring that all the attributes of the object are updated with all of the values on the screen.</p> <p>Uses package: <code>com.ibm.IManagedLocal fooKeyHelper</code></p>	<p>Tester.key</p>
<p>Create a UUID primary key</p>	<p>When an object is defined as a BOIM with UUID in an Object Builder data object implementation, the key is created using the <code>IUUIDPrimaryKeyHelper._create</code> method. Then the <code>generate()</code> and <code>_toString()</code> methods produce the required byte string to use against the home with the normal <code>createFromPrimaryKeyString()</code> method.</p> <p>Uses package: <code>com.ibm.IBOIMExtLocal</code></p>	<p>Tester.key</p>

How do I	Answer	Example QuickTest Java Source
<p>Create an object using the copy helper</p>	<p>This is similar to creating the key helper except that all the attributes that make up the copy helper are set to the fooCopy object before the _toString() method is invoked.</p> <p>The copy helper creates fooCopy:</p> <pre>csAgentCopy aCopy = csAgentCopyHelper._create();</pre> <p>All the attributes of csAgentCopy are set:</p> <pre>aCopy.commPercent(((Float) obj0).floatValue());</pre> <p>The byte string is obtained from the fooCopy object:</p> <pre>aCopy._toString()</pre> <p>This byte string is provided to the Component Broker framework to create the object with a call against the appropriate home using the home.createFromCopyString(keyString) method.</p> <p>You can also refer to the QuickTest-emitted code for _fooCopyBean for an example of generating the required byte string.</p> <p>Uses package: com.ibm.IManagedLocal</p>	<p>Tester.copy</p>

How do I	Answer	Example QuickTest Java Source
Evaluate an OOSQL statement against a home	<p>The evaluate method can only be called on a queryable, iterable home. The statement that is passed to the evaluate method uses the WHERE clause. For QuickTest, the OOSQL is derived by examining the Proxy Screen and determining if any of the OOSQL predicates have been selected. These predicates appear to the left of each attribute of the proxy object. The clause is made up of &lt;attributename predicate value&gt;predicates that are joined together with the AND operator. An example would be "customerName = 'Barton' AND customerAge &lt; 55". The evaluate method returns an iterator.</p> <p>Uses package: com.ibm.IManagedAdvancedClient</p>	Tester.iterate
Create an iterator against a home	<p>If the home is a queryable, iterable home (one that has the QueryableIterableHome interface), then the iterator is obtained with the createIterator() method. An iterator will contain all objects within the home. This is an inefficient means of obtaining the data for all these objects. Since QuickTest takes each object returned within the iterator and extracts the attribute values, much network traffic is generated since each call to get the attribute value is another round trip. A much better mechanism to obtain the attribute values with reduced network traffic is to evaluate to a data array, which is demonstrated in the RefColPanel and QueryPanel code.</p> <p><b>Note:</b> RefColPanel and QueryPanel are two panels that perform OOSQL queries. RefColPanel performs against a reference collection (whether it is transient, or RDB-backed). QueryPanel performs against either the Component Broker run time, or DB2 objects.</p> <p>Uses package: com.ibm.IManagedAdvancedClient</p>	Tester.iterate

How do I	Answer	Example QuickTest Java Source
Return a small population of an iterator	<p>An iterator has a method called nextN which takes two parameters, one is the size request of the return set and the other is a member list holder (MemberListHolder) that contains the requested set. The MemberListHolder is a kind of structure, with one of the attributes being an array. This array holds this list of values after the call. Process the array by checking the length.</p> <p>Uses package: com.ibm.ICollectionsBase</p>	Tester.iterate
Update a CORBA object's attributes	<p>Before an object can have its attributes modified, a transaction may need to be started depending upon the container of the home. Once the transaction is started, the subclass can have its attributes set. For QuickTest, the _fooBean object has an update method where the values from the Proxy Screen are retrieved and set into the object. Tester (the right side of the screen) starts the transaction and then sends a message to _fooBean to perform the update. When _fooBean is complete, Tester commits the transaction.</p> <p>Uses package: org.omg.CORBA.Object</p>	Tester.update
Delete a managed object	<p>The managed Proxy object is sent a message to remove itself. The message is sent during the scope of a transaction. The QuickTest-emitted _fooBean.delete() method is invoked, and the remove() method is invoked on the CORBA object.</p> <p>Uses package: org.omg.CORBA.Object</p>	Tester.delete

How do I	Answer	Example QuickTest Java Source
Retain the elements of an iterator for processing outside the transaction that created the iterator	<p>If an iterator is created from a queryable, iterable, home (QueryableIterableHome), or a return value from a method, the contents can be placed into a transient reference collection. From this collection, an iterator can be created that is then made available to other objects for processing. This is how QuickTest takes the iterator from a foreign key pattern method Parent.listChildren, which returns an iterator containing Children.</p> <p>Uses package: com.ibm.IManagedCollections com.ibm.ICollectionsBase</p>	CopyPaste.setObject
Find out if a CORBA object is actually an iterator	<p>Use the _is_a method with the IDL name. For example, foo._is_a("IDL:ICollectionsBase/Iterator:1.0"). The IDL name can be determined using the IR Browser.</p> <p>Uses package: org.omg.CORBA.Object</p>	CopyPaste.setObject
Create a transient reference collection	<p>Find the factory for the reference collection using the string "IManagedCollections:: IReferenceCollection.object interface/ TransReferenceCollectionFactory.object home". This returns a CORBA object. Narrow this object to the specific home (in this case, ICollectionHome), and invoke the createCollection() method.</p> <p>Uses package: com.ibm.IManagedCollections</p>	CopyPaste.setObject
Copy objects from a transactional iterator to a reference collection	<p>Make a duplicate of the IManageable object since the one contained within the transaction iterator will go out of scope when the surrounding transaction commits. Then take this duplicate object and use the addElement method on the reference collection home.</p> <p>Uses package: com.ibm.IManagedCollection org.omg.CORBA.Object</p>	CopyPaste.setObject

How do I	Answer	Example QuickTest Java Source
Process the elements of an iterator	Use the more method to determine if there are additional objects. Then use the next method to obtain the CORBA object.  Uses package: com.ibm.ICollectionsBase.Iterator	CopyPaste.setObject
Remove an iterator	Use the remove method on the Iterator.  Uses package: com.ibm.ICollectionsBase.Iterator	CopyPaste.clearObject

**RELATED TASKS**

“Chapter 13. Testing applications with QuickTest” on page 611

**RELATED CONCEPTS**

“QuickTest” on page 611

**RELATED REFERENCES**

“The QuickTest framework” on page 620

“QuickTest with Java and JFC”

ISessions in the Sessions Service (*Programming Reference*)

## QuickTest with Java and JFC



The following table provides information for locating code snippets within the QuickTest Framework. The Java code for QuickTest is located in the “samples\Tutorial\QuickTest\framework\com\ibm\quicktest\framework” directory.

How do I	Answer	Example QuickTest Java Source
Initialize the Swing look-and-feel	Use the UIManager class, and the setLookAndFeel method to set the look-and-feel to the appropriate selection. For a cross-platform consistent look-and-feel, use getCrossPlatformLookAndFeelClassName().  Uses package: javax.swing	QuickTest.initLF



How do I	Answer	Example QuickTest Java Source
Change the look-and-feel while the system is active	<p>Provide JRadioButtonMenuItems for each look-and-feel. Add an ItemListener to each MenuItem and when the state becomes selected, update the UIManager with the look-and-feel. Then all the existing frames must be validated and repainted to display the new look.</p> <p>Uses package: com.sun.java.swing.plaf</p>	QuickTest.setupOptions
Process a zipped file for input and read serialized objects	<p>Create a FileInputStream and provide it to the GZIPInputStream constructor. Then supply the GZIP class to an ObjectInputStream. GZIP is an internal class that compresses data to reduce the size of the file that is used either for storage, or for transmittal over the wire.</p> <p>Uses package: java.io java.net</p>	QuickTest.restoreIni
Update each frame with a new look-and-feel	<p>For each container, update, invalidate, validate, and repaint ComponentTreeUI using the respective methods.</p> <p>Uses packages: java.awt javax.swing</p>	QuickTest.doLNF
Determine the superclass for a given name	<p>Create a class using Class.forName(). Then use getSuperClass() to obtain the class name.</p> <p>Uses package: java.lang</p>	QuickTest.determineClass
Create an instance of a specific class given a name	<p>Use the Class object and the newInstance() method and typecast to the desired class using the typeCast operator.</p> <p>Uses package: java.lang</p>	QuickTest.getTestBean
Save a file using the FileDialog and show existing files with a specific extension	<p>Set the filenameFilter of a FileDialog to accept files that have the desired extension.</p> <p>Uses package: java.awt</p>	QuickTest.saveScript

How do I	Answer	Example QuickTest Java Source
Compile Java code from within a Java application	<p>Instantiate a javac compiler using the command <code>javac xxx.java</code>, where <code>xxx</code> is the name of the file to be compiled. The input, output, and error streams are monitored in separate threads to obtain the completion status.</p> <p>Uses package: <code>sun.tools.javac</code></p>	QuickTest.loadScript
Determine the center of the screen	<p>Use the <code>getDefaultToolkit()</code> method, and then the <code>getScreenSize()</code> method. Divide the Dimension object's width and height by two.</p> <p>Uses package: <code>java.lang.Toolkit</code></p>	QuickTest.centerScreen
Display a pop-up menu within a cell in a JTable	<p>The QTBool Editor works with any QTBool object. When the TestTable class puts data into its JTable, the data may contain type QTBool. The JTable will render that for display purposes but when you use the pop-up trigger, the Editor can be invoked. QTBool has a mouse listener added to it. When the pop-up trigger is detected, a JPopupMenu is displayed. The menu items have listeners to perform the copy and paste actions. See the inner class of QTBoolTableCellEditor in TestTable to know how the cell is associated with an instance of this class.</p> <p>Uses package: <code>javax.swing</code></p>	QTBoolEditor. QTBoolEditor
Determine the size of a component	<p>Use the <code>getPreferredSize()</code> method on the component.</p> <p>Uses package: <code>java.awt.Component</code></p>	TestBean.TestBean
Determine the greater of two numbers	<p>Use the <code>Math.max(value1, value2)</code>. The return value is the largest value.</p> <p>Uses package: <code>java.lang.Math</code></p>	TestBean.TestBean

How do I	Answer	Example QuickTest Java Source
Fire a property change event	<p>Use the <code>PropertyChangeSupport</code> and <code>firePropertyChange</code> methods, providing the key value, old value, and new value. Those classes that listen for the property change will be notified.</p> <p>Uses package: <code>java.beans</code></p>	<code>TestBean.setQueryable</code>
Change the font of a label	<p>Create a new <code>Font</code> object with the desired type and set the font on the <code>Label</code> object.</p> <p>Uses package: <code>java.awt</code></p>	<code>QTAnyField.isKey</code>
Change the attributes of a file on AIX or Solaris	<p>Get the run time, and execute the command “<code>chmod</code>” with the appropriate permissions. This will create a process. Use the <code>waitFor()</code> method on the process for execution to complete.</p> <p>Uses package: <code>java.lang.Runtime</code></p>	<code>GenModel.setExecutable</code>
Use a <code>JTable</code> , and control cell display and editing	<p>Provide default cell renderers and editors for the tables that are associated with the data types that will be placed into the data model.</p> <p>Uses package: <code>javax.swing.table</code></p>	<code>TestTable</code>
Control the rendering for a specific data type in a <code>JTable</code>	<p>Implement a <code>DefaultTableCellRenderer</code> and set it on the table for the specific class. To keep a consistent display for selection and focus, first obtain the component by invoking the superclass method, and then customize this component for the new look.</p> <p>Uses package: <code>javax.swing</code></p>	<code>TestTable.setDataModel</code>
Associate a customized editor for a specific data type in a <code>JTable</code>	<p>Instantiate a class that extends a <code>DefaultCellEditor</code> and associate it with the desired data type by using the <code>setDefaultEditor()</code> method on the <code>JTable</code>.</p> <p>Uses package: <code>javax.swing</code></p>	<code>TestTable.setEditors</code>

How do I	Answer	Example QuickTest Java Source
<p>Use an interface to avoid use of the instanceof operator</p>	<p>Define an interface that defines a method that needs to be implemented to achieve the desired behavior. In this example, all of the data that is displayed in the JTable can be copied using a pop-up menu. To guarantee that each representation can support the execute statement and that the value obtained from the data model does not require integration using the instanceof operator with a series of if statements (for example, if (foo instanceof bar) . . .), have each object implement the interface. Every object that implements the QTExecute interface has the execute() method.</p> <p>Since each QT data element (object contained in TestTable that participates in the TestTable behavior) is an instance of QTExecute, and implements the QTExecute interface, each of them has the execute() method, and implementation code in that method. As new data types are introduced, if they implement QTExecute, the following code will not require any modification:</p> <pre>if (obj instanceof QTExecute) ((QTExecute) obj).execute()</pre> <p>This allows for polymorphism. If this is not the case, you would have to use code that needs modification for each new object that is introduced into the system. For example:</p> <pre>if (obj instanceof Type1) ((Type1)obj).doSomething(); else if (obj instanceof Type2) ((Type2)obj).somethingElse();</pre> <p>Uses package: java.lang</p>	<p>TestTable. selectAndCopy</p>
<p>Set an icon in the upper left-hand corner of a window</p>	<p>Use the setIconImage method of the frame and use an icon image.</p> <p>Uses package: java.awt</p>	<p>TestFrame.TestFrame</p>

How do I	Answer	Example QuickTest Java Source
Add a toolbar with buttons to a frame	Create a JToolBar and add JButtons to it.  Uses package: javax.swing	TestFrame.TestFrame
Use scroll panes within split panels	Use two scroll panes and setViewportView with the JPanel which contains the objects to scroll. Then take a JSplitPane and set the left and right components.  Uses package: javax.swing	TestFrame.TestFrame
Protect the screen during a long operation	Create a JComponent that covers the screen, and set it as the glass pane for the frame. Whenever protection from user interaction is desired, set the component to be visible. Then, no interaction with data entry fields, buttons, list boxes, or any other interface element will be possible. See the QTGlassPane for a complete sample.  Uses package: javax.swing	TestFrame.TestFrame
Create "CheckBox MenuItems" and know when the user selects it	Create an instance of the JCheckBoxMenuItem and add an anonymous action listener which implements the actionPerformed method. When the user toggles the item, the(ActionEvent) object is a parameter to the actionPerformed method. Use the object with the getSource() method, and then check the state of the check box with the isSelected() method.  Uses package: javax.swing	TestFrame. setOptionsMenu
Create a radio button group with menu items	Create JRadioButtonMenuItems and a ButtonGroup. Add the menu items to the menu for display and selection. Add the menu items to the ButtonGroup so that they behave in a mutually exclusive manner.  Uses package: javax.swing	TestFrame. setOptionsMenu

How do I	Answer	Example QuickTest Java Source
Create a submenu, which contains other menu options	<p>Add to the main JMenu a new JMenu that contains JMenuItem.</p> <p>Uses package: javax.swing</p>	<p>TestFrame. setOptionsMenu</p>
Detect a mouse click on a JTable heading	<p>On the table, use the getTableHeader() method and add a mouse listener that responds to mouseClicked events. When the event occurs, use the getColumnModel() method of the table and determine if the x coordinate is in the desired column.</p> <p>Uses package: javax.swing</p>	<p>Script. addMouseListener ToHeaderInTable</p>
Change the font of a label	<p>Create a new Font object with the desired type and set the font on the Label object.</p> <p>Uses package: java.awt</p>	<p>QTTextField.isKey</p>
Control the data displayed in a JTextField	<p>If a JTextField is edited within a JTable, when editing is complete, the document itself is modified; the JTextField.setText() method is not modified. When you type into a JTextField, the document is actually notified first so that the data can be edited. See the QTNumberDocument class for further information.</p> <p>Uses package: javax.swing.text</p>	<p>QTTextField. createDefaultModel</p>
Edit JTextField data before the data is visible in the text field	<p>Extend a PlainDocument class and provide that class using the JTextField.createDefaultModel() implementation.</p> <p>Uses package: javax.swing.text</p>	<p>QTTextField. QTNumberDocument</p>

How do I	Answer	Example QuickTest Java Source
Prevent erroneous data from entering JTextField by keyboard entry, or by Copy or Paste actions	<p>When data is to be set into the Model for JTextField, the insertString() method of the default Model is called. The string that is to be inserted is available in the parameters. The string will consist of one character if entry is being made using the keyboard. The string will have potentially more characters if a Paste operation is being performed.</p> <p>Uses package: javax.swing.text</p>	<p>QTTextField. QTNumberDocument. insertString</p>
Position the text caret at the left if the string is padded with blanks.	<p>Large strings can have trailing filler blanks and when set into a JTextField, the display will be right-justified. You may not see the initial character values if the display is smaller than the length of the string. To force the display to the left position, use the setCaretPosition() method of the JTextField.</p> <p>Uses package: javax.swing.text</p>	<p>QTTextField. QTNumberDocument. insertString</p>
Implement a custom layout manager	<p>Implement the LayoutManager interface. The QTLayout example shows laying out the components either vertically or horizontally. This layout manager supports the components all occupying the same area. This is used so that TestFrame can lay out multiple panels that contain parallel rows of information and keep them presented in line. Without the same size constraint, the components would assume their preferred size.</p> <p><b>Note:</b>TestFrame is a component within QuickTest. It is a frame with a title bar, complete with its own set of Minimize and Maximize buttons, and so on. All the test cases are contained within a TestFrame.</p> <p>Uses package: java.awt</p>	<p>QTLayout</p>

How do I	Answer	Example QuickTest Java Source
Load an image as an applet from the containing .jar or .zip file	<p>Unzip the somiaqf.zip file to see the .gif files that are contained in the package directory. Determine where the applet is a stream resource and create an InputStream. Use the input stream to read the the image into a byte array. Then initialize an ImageIcon with the byte array. From the ImageIcon, an image can be retrieved.</p> <p>Uses packages:  java.net  java.awt  java.io</p>	Util.loadImage
Load an image as an application from the containing .jar or .zip file	<p>Unzip the somiaqf.zip file to see the .gif files that are contained in the package directory. Get the URL for the resource given the name of the file. Then create a MediaTracker and allow it to get the image. This gets the image without flickering.</p> <p>Uses packages:  java.net  java.awt  java.io</p>	Util.loadImage
Copy a file	<p>Use the FileInputStream class to read each character, and the FileOutputStream class to write each character.</p> <p><b>Note:</b>FileInputStream and FileOutputStream are core Java classes that permit file manipulation (either reading or writing) in a platform-independent manner.</p> <p>Uses package:  java.io</p>	Util.copyFile
Get parameters from applet tags	<p>The HTML contains tags with name and value pairs. To obtain the values, use the getParameter method.</p> <p>Uses package:  javax.swing</p>	QuickTestApplet.init



How do I	Answer	Example QuickTest Java Source
<p>Read a .jar file from within an applet, and determine the contents</p>	<p>Get the document base and build a URL with the name of the .jar file. The .jar file name can be obtained from a tag within the HTML source. Using the URL, open a stream and pass the stream to the constructor of a ZipInputStream. Using this stream, process each ZipEntry.</p> <p>This gives QuickTest the ability to inspect a .jar file at run time, and determine what the contents are. QuickTest needs this ability because it does not know what beans are contained in the .jar file (what test cases are being processed). QuickTest needs to instantiate each class and look at the base classes to determine how to perform the dynamic wiring of the events.</p> <p>Uses package: java.net java.util.zip</p>	<p>QuickTestApplet.init</p>
<p>Send a window message to a specific frame</p>	<p>Create a WindowEvent with a type of event to be processed and then invoke the dispatchEvent().</p> <p>Uses package: javax.swing</p>	<p>QuickTestApplet.stop</p>
<p>Detect a JComboBox selection</p>	<p>A JComboBox will create its own model of data if the constructor has the data provided. When an item is selected, the method itemStateChanged is called with the event that occurred. By extending the JComboBox, the method can be overridden and the getSelectedIndex can be called to determine the current selection.</p> <p>Uses package: javax.swing</p>	<p>TestBean.QTChoice</p>

How do I	Answer	Example QuickTest Java Source
Get the headings for the tables on initialization	<p>A tester (an object within QuickTest) and a subclass of TestBean are identified during initialization so that messages can be sent between them. For the tester to display the headings in the TestTable, Tester sends a message to the TestBean. The TestBean sends a message back to the tester which contains an array of the attribute names that can be used as column headings.</p> <p>Uses package: java.awt.event.ActionEvent</p>	Tester.testFrame
Process PropertyChangeEvent (Beans)	<p>When the Host panel is completed, it sends messages to the listeners to inform them that the properties have changed. The event contains the property name and new value.</p> <p>Uses package: java.beans</p>	Tester.propertyChange
Process elements contained in Vector	<p>Use the Enumeration object obtained from the elements() method. Then process the Enumeration object by checking if there are more elements with the hasMoreElements() method and if true, obtain the next element with nextElement().</p> <p>The returned object must be typecast to the correct type since the Vector contains objects of type java.lang.Object.</p> <p>Uses package: java.util</p>	Tester.haveIdentical
Keep the screen repainted during long-running calls	<p>The QuickTest class Perform starts a thread and calls a Runnable object. A Runnable object implements the run() method. This thread allows the screen to continue to be refreshed and allows for simultaneous execution of commands.</p> <p><b>Note:</b> During QuickTest script playback, threading is not used.</p> <p>Uses package: java.lang.Thread</p>	Tester.actionPerformed

How do I	Answer	Example QuickTest Java Source
Implement a clone	<p>The clone method allows an object to return a copy of itself. This is important, for example, when a copy or paste of the object requires a unique object. When QuickTest copies a CORBA Proxy object, the clone is used so that the original and new clone are edited separately, in the same way as when the clear method is invoked.</p> <p>Uses package: java.lang.Object</p>	CopyPaste.clone
Use a clipboard and transfer a CORBA Proxy using the paste operation	<p>Multiple objects are copied to the clipboard. A static clipboard is used to contain the Transferable object. The Transferable object is of a specific DataFlavor (type such as string, Object, integer, and so on). The DataFlavor determines if the contents of the clipboard are appropriate for the copy. If there is no Transferable object in the clipboard, there is nothing to paste.</p> <p>Uses package: java.awt.datatransfer</p>	CopyPaste.execute
Use a clipboard and copy a CORBA Proxy to it	<p>A static clipboard is used to contain the Transferable object. QuickTest uses the ProxySelection object that implements the Transferable and ClipboardOwner interfaces. When the user selects a Copy, a ProxySelection object is instantiated for the DataFlavor and CORBA object. The clipboard then has the contents set with this ProxySelection.</p> <p>Uses package: java.awt.datatransfer</p>	CopyPaste. copyToClipboard

How do I	Answer	Example QuickTest Java Source
<p>Release system resources when garbage collection occurs</p>	<p>When the Java virtual machine (JVM) recognizes that a specific instance of an object is no longer referenced, and is therefore not necessary to the program, the JVM can free that memory in a process called garbage collection.</p> <p>When an object is garbage-collected, the <code>finalize()</code> method is invoked. Within this method, an action can be performed to release system resources. For QuickTest, the CopyPaste object detects an iterator, transfers the contents to a transient reference collection, and creates a new iterator from the collection. The contents can then be processed outside the scope of the transaction that created the original iterator.</p> <p>The collection of objects that is referenced by the iterator can be accessed without having a transaction. Component Broker's behavior is such that when an iterator of objects is created within a transaction, the objects can only be referenced within the scope of that transaction. For QuickTest, for example, we may return rows of employees and want to scroll through them over time and select some to see their details. We may not want to keep the transaction active during that process. So, we can retrieve them first from the iterator, and store them in a transient iterator. Because it is transient, if the system were to fail, we would lose the collection itself. But mainly, by putting the objects in the transient collection, we can access them without a transaction being required.</p> <p>This means that the transient iterator needs to be removed when the CopyPaste object is garbage-collected.</p> <p>Uses package: java.lang</p>	<p>CopyPaste.finalize</p>

**RELATED TASKS**

“Chapter 13. Testing applications with QuickTest” on page 611

**RELATED CONCEPTS**

“QuickTest” on page 611

**RELATED REFERENCES**





“The QuickTest framework” on page 620

“QuickTest with the Component Broker Programming Model” on page 622

---

**QuickTest-generated files**

Object Builder generates the makefiles which invoke the QuickTest emitter to create the following files for QuickTest:

- Source: `\Working\<platform>\QT\`
- Compiled classes: `\Working\<platform>\<config>\QTCLS\<DLLname>\`
- JAR files: `\Working\<platform>\<config>\QT\QT<DLLname>.jar`
-  QuickTest executable location:  
`\Working\<platform>\<config>\qt.bat`
-    QuickTest executable location:  
`/Working/<platform>/<config>/qt.ksh`

 You cannot build for QuickTest on OS/390.

**RELATED TASKS**

“Generating QuickTest client applications” on page 614

“Running QuickTest client applications” on page 615

“Recording QuickScript” on page 617

“Running the QuickTest tutorial”

**RELATED CONCEPTS**

“QuickTest” on page 611

“QuickScript” on page 616

**RELATED REFERENCES**

“The QuickTest framework” on page 620

---

**Running the QuickTest tutorial**

The QuickTest tutorial uses the ForeignKey sample to demonstrate the use of the QuickTest client application and its QuickScript replay facilities.

## Objective of the QuickTest tutorial

The objective of the QuickTest tutorial is to demonstrate how you can use a QuickTest client application to test your Component Broker server application.

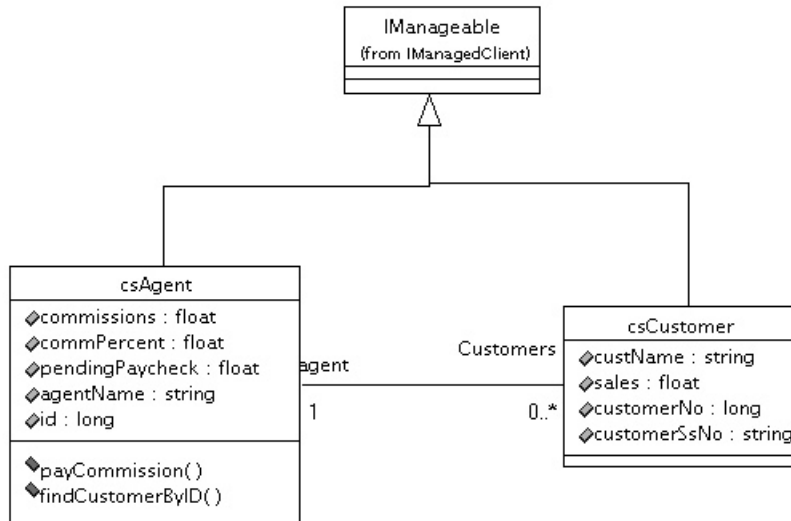
## The QuickTest tutorial's programming model

The QuickTest tutorial uses the Foreign Key sample that is provided with Component Broker. This sample is located in %IVB\_DRIVER\_PATH%\samples\Tutorial\Fundamentals\foreignkey directory.

**Note:** The documentation is written using the Windows NT paths and naming conventions. For a UNIX environment, %IVB\_DRIVER\_PATH% should be \$IVB\_DRIVER\_PATH and the backward slash (\) should be a forward slash (/).

## The Rose model

The following is a Rose model depicting the sample that is used by the QuickTest tutorial:



## Files shipped for the QuickTest tutorial

The QuickTest Framework source code is located in the <Cbroker>\samples\Tutorial\QuickTest directory on a machine that has the samples installed. The emitted code from the *make* process is located in the project's Working\<platform>\<config>\QT directory.

## Description of the QuickTest tutorial

This is a summary of the steps in the QuickTest tutorial:

1. Delete all objects in the csAgent and csCustomer homes.
2. Add some agents by using the Add by Key and Add by Copy methods.
3. Iterate over the csAgent home to observe its contents.
4. Add some customers to the csCustomer home using the Add by Copy method.
5. Perform an evaluate on the csCustomer home.
6. Some customers are assigned to agents using the methods provided with the csAgent object. You can use the Copy and the Paste functions to copy a customer to the parameter of the agent's addCustomers method.
7. Invoke the listCustomers method on a particular agent to return an iterator which contains objects of type Customer. You can copy and paste this iterator to the iterator of the customer's home to display its contents.
8. Use the query evaluator to perform OOSQL queries over the csAgent home.
9. Add some customers to a transient reference collection and proceed to query that reference collection.
10. Save all the recorded test events as a QuickScript file.
11. Replay the QuickScript file.

For detailed descriptions of the options on a particular window, select **Help**, or press F1, from within the window.

## Assumptions

This tutorial assumes that you have performed the following preliminary steps:

1. The ForeignKey sample has been built, and installed on a host machine that is accessible.
2. The client is running on the server machine.
3. All the files that are required to support **QuickTest** are installed on the server (somojj.zip, somshcl.zip, somojor.zip, somiaqe.zip, somiaqf.zip and ivbjfc.jar).

**Note:** If **QuickTest** is running on a client install machine, copy these files from the %IVB\_DRIVER\_PATH%\lib directory from the server machine.

## Starting the QuickTest Tutorial

When we start QuickTest (by clicking **Build > Run QuickTest** from the Build Configuration folder's pop-up menu), we must first establish the host, port, timeout, and security options in the CB Server Host window:

- Type a host name and server name that are appropriate for your environment.

- Set **Timeout** to 180
- Set **Port** to 900
- Set **Iterate Count** to 25

Refer to help from within the window for more details on the contents of the window.

### **Assigning keys and copy helpers for managed clients**

If this is the first time you are running QuickTest, or if this is a subsequent time but you have deleted the configuration files (config.key and config.copy), the Assign Key Helpers and Assign Copy Helpers windows open, and you can associate keys and copy helpers for the managed clients.

By default, QuickTest assigns a default keys and copy helpers for the managed clients.

### **Note the following points:**

- The associations that you create here determine success or failure when you run QuickTest.
- When you are assigning keys, keep in mind that in the case of inheritance, both the parent and child can be mapped to the same key.

### **The QuickTest tour**

Now we are ready for a guided tour of QuickTest by maintaining data, invoking methods and exercising the Foreign Key sample that is installed on Component Broker. We will use the Copy and Paste actions throughout the process, and show how all the test actions are recorded. These QuickScript recordings are then reloaded and replayed.

### **Step 1 - Delete All**

In the QuickTest Main window, from the **Test Cases** menu, select **sample5.\_csAgentBean** to open the Test Case window. Refer to help from within the window for more details on the entries in the window. The default options in the Test Case window are correct for this application. Review the default options in the window's **Options** menu. The defaults are:

- No security
- Queryable
- Persistence is transactional
- Reference collection is transient

**Note:** The same defaults apply to **sample5.\_csCustomerBean**. To start the **csCustomerBean**, select the **Test Cases** menu and then **sample5.\_csCustomerBean** in the QuickTest Main window.



Select **Delete All** from the home toolbar for both test cases to delete all the objects defined in the csAgent and csCustomer homes so that when we replay the QuickScript that is being recorded, we will not have problems with duplicate key adds. If the purpose of the QuickScript replay is to have exactly the same behavior as the initial run, then start the test by deleting all the objects.

### Step 2 - Add Agents

Now that the csAgent home is empty, we will add agents: FirstAgent and SecondAgent. Type the information, and for the first agent, click the **Add by Key** push button from the Proxy toolbar. For the second, select **Add by Copy**. These two types of Adds are available because keys and copy helpers were defined within Object Builder. If a key or copy helper is not defined in the model, the **Add by Key** and **Add by Copy** push buttons are not enabled on the toolbar.

**Note:**As the objects are added, they are displayed in the home table (either for transient objects or for non-queryable homes). When a home is not queryable, the table can be populated by performing either the **Find**, or **Add by Key** and **Add by Copy** functions. Also, when an object is deleted, the row that contains the object will be removed from the table.

Follow these steps using the Proxy pane:

1. Type the data:  
Michael Francis, ID = 10, commPercent = .50
2. Select **Add by Key** on the toolbar.
3. Type the data:  
Charles Hobbes, ID = 20, commPercent = .50
4. Select **Add by Copy** on the toolbar.
5. Select any column from the resulting row in the table of objects in the right pane (the home).

Note that the sample5.csAgent\* label is now cyan which means that it can be copied. Also, the method buttons (lower left) for the Proxy are also enabled, indicating that the methods can be invoked.

### Step 3 - Iterate on Agent Home

Select **Iterate** from the Home's toolbar. After the iteration is complete, you can alter the window in the following ways:

- The columns can be rearranged by selecting the column heading with the mouse and dragging it to a new column position.
- The column headings can be stretched or expanded by dragging the separators with the mouse.

- The panes can be resized by dragging separators of the panes with your mouse. However, each pane cannot be resized to an area that is smaller than the width of its toolbar.
- The entire window can also be resized.

#### Step 4 - Add Customers to Customer Home

In the QuickTest Main window, from **Test Case** menu, select **sample5\_csCustomerBean**. Then in the Proxy pane of the Test Case window, add customers with the following details:

1. Bob Smith, customerNo =11
2. Adam Smith, customerNo = 12
3. John Smith-Jones, customerNo = 13
4. Andrea Andrews, customerNo =21
5. Elijah Aardvark, customerNo = 22
6. John Q. Public, customerNo = 23
7. Marie Forest, customerNo=31

Select **Add by Copy** after specifying the data for each customer.

#### Step 5 - Evaluate on Customer Home

The following steps perform an evaluate on the customer home. This incorporates the OOSQL predicate usage. The evaluate statement on the home is created by using one or more OOSQL predicates with a value supplied for the corresponding attribute. The window is updated after you complete these actions.

1. Select **>** (greater than) from the list box next to **customerNo**.
2. Set the **customerNo** value to 0.
3. Select **Iterate** from the Home toolbar.

As a result, a predicate of **customerNo > 0** is passed to the evaluate method on the csCustomer Home when **Iterate** is selected.

#### Step 6 - Associate a csCustomer to a csAgent

The ForeignKey pattern provides three methods, addFoo, removeFoo and listFoo where Foo is the child object contained by the parent. In this sample, the child is the csCustomer. To establish the relationship between the parent and child (Agent and Customer), the Agent must be identified and the Customer object must be provided as the customer parameter of the addCustomer method.

In this step, you select a customer and copy that customer to the agent window as a parameter to the **addCustomers** method. The parameter name is csCustomer. Before the customer can be copied to the agent, an agent must be selected.

### 1. **Select Agent**

From the `sample5._csAgentBean` window, select the row with **Michael Francis**.

**Note:** `sample5.csAgent*` is cyan, which indicates that there is a selected agent that can be copied and pasted.

### 2. **Select Customer**

From the `sample5._csCustomerBean` window select the row with Bob Smith. Note that the label `sample5.csCustomer` is now cyan, and indicates that the selected customer can be copied and pasted.

### 3. **Copy Customer**

To copy the selected customer, select the label `sample5.csCustomer*` in the `sample5._csCustomerBean` window, and from its pop-up menu, select **Copy**. This copies the CORBA object to the clipboard.

### 4. **Paste customer to agent method's parameter**

In the `sample5._csAgentBean` window, select the label `sample5.csCustomer` (which is a parameter of the `addCustomers` method for the proxy). From its pop-up menu, select **Paste**. The customer object is pasted from the clipboard, and the label changes to `sample5.csCustomer*`.

### 5. **Invoke Agent addCustomers method**

Click the `addCustomers` method to add the customer to this agent.

### 6. Using the same method, add one more customer to this agent by following these steps:

- a. From the `sample5._csCustomerBean` window, select the row on Adam Smith.
- b. In the Proxy pane, select `sample5.csCustomer*`, and from its pop-up menu, select **Copy**.
- c. In the `sample5._csAgentBean` window, right-click on the label, `sample5.csCustomer`, which is a parameter of the `addCustomers` method for the proxy, and select **Paste**.
- d. Click `addCustomers` method to invoke the method.

## **Step 7 - Display Customers associated with Agent**

The objective of this step is to determine which customers are now associated with the selected agent. It also demonstrates how return values from method invocations can be displayed. In this particular case, the return value is of type `Iterator`.

You have to perform the following actions in this step:

- Invoke the `listCustomers` method on `csAgent`
- Copy the return value of `Iterator*`
- Paste the `Iterator*` to the `Iterator` label in the `csCustomerBean` window

Select the **listCustomers** method. The **Return = Iterator\*** now indicates that a CORBA object is available for copying. Select the returned **Iterator\***, and from its pop-up menu, select **Copy**.

In the sample5.\_csCustomerBean window, select the **Iterator**, and from its pop-up menu, select **Paste**.

Once you have pasted **Iterator\***, you see the two customers that belong to agent Michael Francis displayed in the window.

### Step 8 - Query Evaluator

This step demonstrates the use of the query evaluator for the csAgentBean. In the sample5.\_csAgentBean window, from the **Options** menu, select **Query Evaluator**. The Query Evaluator window opens, and you can specify OOSQL statements against the home. Refer to the help from within the window for more details.

Specify the OOSQL statement in the **Query Prompt** field. To initialize the prompt with an OOSQL statement that includes all the attributes and objects that are contained in the home, make sure that the radio button **Select all** is selected, and then click **Set Prompt**. Once the prompt field is populated, click **Query** (which is next to the prompt entry) to perform the query. The result sets are presented in a table similar to the Test Case Home table in the lower part of the window. You can copy attributes from the table to paste into other parts of QuickTest. In this exercise, after initializing the prompt field, add the method **listCustomers** to the select clause. This is the way to invoke methods in OOSQL statements. Click **Query** to run the query.

You can see the results of the query against the home in the window. There is a column for the method **listCustomers**, which shows the return values of type **Iterator**, which contains the customers defined for the csAgent. Note that even though there are no customers defined for Charles Hobbes, the **Iterator** is non-null, but the **Iterator** contains no elements.

**Note:**An **Iterator** will always be non-null. It is supposed to always return an object instance. It may not contain any elements, depending upon how it was created, and the data source. In this example, there were no customers defined. So, the **Iterator** returned empty.

Now select the row with Charles Hobbes and the cell under the column **\_csAgent**. From the pop-up menu of the cell, select **Copy**. This provides the CORBA object for Charles Hobbes, which can be copied for pasting to other windows in QuickTest.

Now, in the sample5.\_csAgentBean window, place the mouse on the Proxy label, right-click and select **Paste**. This displays all the attributes for this object.

### Step 9 - Reference Collection

The reference collection must first be populated with data. To achieve this for the tutorial, perform the following steps:

- Select the sample5.\_csCustomerBean window.
- Click **Iterate** on the home so that all the csCustomers are returned.
- Select the **Add All to Reference Collection** to add all the csCustomers to the transient reference collection.
- Open the Reference Collection window, sample5.\_csCustomerBeanRefCol, by selecting **View > Reference Collection** from the sample5.\_csCustomerBean window. Select **Iterate** in the Reference Collection window.

Refer to the help from within the Reference Collection window for more details on the entries in the window.

Note that the two csCustomers have the agent attribute set with the agent they were assigned to from the previous steps. As with the Query Evaluator, you can select the cell, and copy and paste it to the csAgent window proxy label. Also, the **Query Prompt** field of the Reference Collection window is populated in the same manner as that of the Query Evaluator window. Note that since both the Query Evaluator and Reference Collection tables are only for display, you can only copy from these tables.

When this Reference Collection window is closed, the collection is removed. The collection can always be extended by adding other objects using either the **Add to Reference Collection** from the Proxy pane, or the **Add All to Reference Collection** from the Home pane in the proxy window.

### Step 10 - Save the QuickScript

During this entire tutorial, QuickTest has been recording all the activities that have been exercised. At this time, use the QuickTest Main window and save your actions in a QuickScript file using **File > Save Script**. Save the file in the default directory (where the build and run scripts are located), and use the default name: tmpScript.java.

The tmpScript.java file that is saved contains all the instructions that were executed during the steps of the tutorial.

QuickScript files are written in Java. The scripts access the classes that were emitted in the build step. The actions that are recorded are the events that occurred during execution of QuickTest that involve Component Broker.

Individual, elementary events such as key strokes, or mouse movements are not recorded. Besides, the state of data for Adds, Updates and Deletes, and the query statements that were invoked are also recorded. The replay of QuickScript does not verify that the data processed during the initial recording is the same but rather that the activity can be replayed successfully.

### Step 11 - Loading the Script

Select **File > Load Script** and select the saved file, tmpscript.java. QuickTest will compile the QuickScript for you. A message appears while the compilation takes place. If there is an error, an output file is generated and saved to the directory. This file can be viewed, and corrections made to the QuickScript file.

### Step 12 - Replaying QuickScript

Once the compilation completes successfully, the Controller window gives you the following options, which are available when running the QuickScript file:

These options are found on the right side of the screen:

- **Number of Loops**  
The entire QuickScript file will be executed using as many loops as indicated. This provides the capability to stress the system. For our purposes, type 2.
- **Current Loop**  
While the QuickScript file is running, the current loop is displayed.
- **Sleep Duration**  
This value is in terms of milliseconds (5000 ms = 5 seconds). Type 500. This is equal to half a second.
- **Start Time**  
When you click the **Start** button, this value is updated to identify the time at which the execution started.
- **End Time**  
When the specified number of loops is completed, this field is updated with the time the execution completed.

These columns appear on the left side of the screen:

- **Stop**  
Each cell in this column has a check box, which you can select, or clear to toggle the option on or off. The entire column can be toggled by clicking the title of the column. The statement that is ready to execute first checks if the Stop cell is set. This statement will not execute until you clear the Stop value, or click **Continue**. But first, the Stop value is checked against the value of the corresponding **After** column. The value of the **After** cell determines when the statement will stop executing (if **Stop** is enabled) in relation to the value specified for **Current Loop**. For example, if the After value is set to 1, row 15 will stop executing only when the value of **Current**

**Loop** exceeds 1. This enables the system to execute the loop many times, and provides the capability to stop after, for example, the 53rd loop.

- **After**

This value is compared with that of **Current Loop** to determine if the **Stop** cell at the same row position is set should be activated. When the **After** value is equal to the **Current Loop** value, if the **Stop** cell is set, the statement will be stopped.

For example, if the **After** value is 3, processing starts, and it passes the first and second loops without incident. On the third time through, it encounters this statement. It halts before executing the statement, and will not proceed until you click **Continue**, or you clear the **Stop** check box.

- **Sleep**

Rows that have the **Sleep** cell set will pause for the amount of time specified in **Sleep Duration** before execution. The entire column can be toggled on or off by clicking the column heading **Sleep**. The duration of sleep will be 5000 milliseconds. The value of **Sleep Duration** can be modified at any time, and the next execution of the **Sleep** row will use the new value.

- **Row**

This is the count of the statement within the QuickScript file. When the execution of a statement is stopped, the command console displays a statement indicating the row that is currently stopped. This is to assist the user in scrolling the table to the row with the active Stop statement when they are viewing a large table of statements.

- **Statements**

These are the QuickScript statements that were executed during the recording stage of QuickTest.

Buttons:

- **Start**

Begins the execution of the QuickScript file.

- **Continue**

When a Stop statement is reached, this button will be enabled. If you click the button at this time, the statement will proceed to execute and the Stop setting will remain.

Start the playback by clicking the **Start** button. You will observe that each row that is to execute will be selected within the window. In the command window that started the QuickTest application, a message appears with information on which row is sleeping. After the sleep time is satisfied, the selected row is executed. When a statement with **Stop** selected is encountered, the application will stop until you either click **Continue**, or clear the check box on that row. In either case, the execution of that statement will then proceed. For this example, we will use the **Continue** button so that on the second iteration, the Stop flag will still be in effect for this statement.

While a statement is stopped, it is possible to check the database or server application. For example, during this breakpoint, you could access the table in the database for the application, and select all the rows to determine the state of the contents.

Allow the script to continue through to the end of the statements. Since the loop count was set at two (2), another execution of the script is required.

Again, the breakpoint on the stopped statement will be performed since the check box is still selected. Allow the application to continue. After completion of the two loops, the QuickScript Controller will still be active. Note the new time in the **End Time** field. If desired, the Sleep, Stops and Loop counts can be modified and additional testing performed.

#### **RELATED TASKS**

“Generating QuickTest client applications” on page 614

“Running QuickTest client applications” on page 615

“Recording QuickScript” on page 617

“Compiling the QuickScript file” on page 618

“Running QuickScript” on page 618

#### **RELATED CONCEPTS**

“QuickTest” on page 611

“QuickScript” on page 616

#### **RELATED REFERENCES**

“The QuickTest framework” on page 620

“QuickTest-generated files” on page 647



---

## Chapter 14. Command-line interfaces

---

### Using Object Builder from the command line

The following tasks describe command-line interfaces to Object Builder:

- “Migrating from the command line” on page 35
- “Exporting XML from the command line” on page 660
- “Importing XML from the command line” on page 663
- “Importing IDL from the command line” on page 666
- “Importing edited source files from the command line” on page 682
- “Importing enterprise beans from the command line” on page 670
- “Generating code from the command line” on page 684
- “Rebuilding DLLs” on page 565

#### RELATED CONCEPTS

“Object Builder” on page 1

#### RELATED TASKS

“Developing in Object Builder” on page 19

#### RELATED REFERENCES

“obmigrate”

“obexport” on page 661

“obimport” on page 664

“obgen” on page 685

“make options” on page 688

“importidl” on page 667

“importimpl” on page 683

“cbejb options” on page 673

### obmigrate

The `obmigrate` command can be used from the command line to migrate a project or projects, along with the projects they depend on, from the 3.0 format to the 3.5 format.

The syntax of the `obmigrate` command is:

```
obmigrate -all project1 project2 ...projectn
```

The parameters are:

- **-all**

If the listed projects depend on other projects, these other projects will also be migrated. This is generally the appropriate option when you want to migrate a set of interdependent projects (a team environment) in a single step. Dependencies will be migrated recursively.

For example, if you migrate a project \Integration, which depends directly on \A, which depends in turn on \B, which depends in turn on \C, then even though \Integration only depends directly on \A, it depends indirectly on \B and \C as well, so those projects are also migrated.

If you omit the -all option, and one of the listed projects has dependencies, then you will be prompted on how to handle these dependencies.

- *project1 ...projectn*

The list of projects to migrate. Project directories should be fully qualified and separated by spaces (for example e:\myprojects\projectA e:\myprojects\projectB)

**RELATED CONCEPTS**

“Projects and models” on page 17

**RELATED TASKS**

“Migrating projects from 3.0” on page 33

“Migrating from the command line” on page 35

“Migrating a team environment” on page 36

---

## Exporting XML from the command line

You can export the XML that defines a project’s model either from within Object Builder, or from the command line.

To export from the command line or batch interface, use the “obexport” on page 661 command:

**obexport**

-p *project\_directory*

[-d *target\_file*]

-ALL | [-UDBO -UDDO -UDCB -UDLOCAL -DLL -APPL -NIDL -CONT -UDPAO -UDSCHEMA -UDEJB]

-v

Files are exported from the project directory (-p) to its \Export subdirectory. Optionally, the exported content can be generated into a single target file (-d).

You can either export the files for the main folders in the project (-ALL), or for a specific folder or object type (for example, -UDBO for the User-Defined Business Objects folder).

For example, if a project defines three components (Policy, CarPolicy, and Claim), then the command:

```
obexport -p e:\myproject -UDBO -DLL -APPL
```

generates the following files into e:\myproject\Export\ :

- udbo.PolicyFile.xml, udbo.CarPolicyFile.xml, udbo.ClaimFile.xml (the component business object layers)
- uddl.PolicyS.xml, uddl.PolicyC.xml, uddl.ClaimS.xml, uddl.ClaimC.xml (the contents of the Build Configuration folder)
- udaf.MyApplicationFamily.xml (the application family defined in the Application Configuration folder)

#### RELATED CONCEPTS

“Model interchange with XML” on page 492

#### RELATED TASKS

“Using Object Builder from the command line” on page 659

“Exporting XML” on page 387

#### RELATED REFERENCES

“XML interchange files” on page 493

“obexport”

## obexport

The obexport command can be used from the command line to export XML files for a specified project.

The syntax of the obexport command is:

### obexport

-p *project\_directory*

[-d *target\_file*]

-ALL | [-UDBO -UDDO -UDCB -UDLOCAL -DLL -APPL -NIDL -CONT -UDPAO  
-UDSCHEMA -UDEJB]

-v

The parameters are:

- -p *project\_directory*

The project directory you are exporting from. The exported contents will be

placed in the project directory's \Export subdirectory, unless you are exporting to a target file and you specify a different path. Required.

- **-d *target\_file***  
The name and path of a target file to generate content into. If you do not specify a target file, the contents are exported to separate files based on the content type, as described in "XML interchange files" on page 493.
- **-ALL**  
Exports XML for the contents of the project's folders. Cannot be set with any of the folder- or object-specific export options.
- **-UDBO**  
Exports *udbo.bofile.xml* for each business object file defined in the project's User-Defined Business Objects folder. Cannot be set with -ALL.
- **-UDDO**  
Exports *uddo.dofile.xml* for each data object file defined in the project's User-Defined Data Objects folder. Cannot be set with -ALL.
- **-UDCB**  
Exports *udcb.composition.xml* for each composition file defined in the project's User-Defined Compositions folder. Cannot be set with -ALL.
- **-UDLOCAL**  
Exports *udlocal.localonly.xml* for each local-only object file defined in the project's Local-Only Objects folder. Cannot be set with -ALL.
- **-DLL**  
Exports *uddll.dll.xml* for each DLL defined in the project's Build Configuration folder. Cannot be set with -ALL.
- **-APPL**  
Exports *udaf.appfamily.xml* for each application family in the project's Application Configuration folder. Cannot be set with -ALL.
- **-NIDL**  
Exports *udnidl.nonidltype.xml* for each non-IDL type in the project's Non-IDL Types folder. Cannot be set with -ALL.
- **-CONT**  
Exports *udcontainer.container.xml* for each user-defined container in the project's Container Definition folder. Cannot be set with -ALL.
- **-UDPAO**  
Exports *udpaschema.paschema.xml* for each PA schema in the project's User-Defined PA Schemas folder. Cannot be set with -ALL.
- **-UDSCHEMA**  
Exports *uddb-schema.dbschemagroup.xml* for each DB schema group in the project's DBA-Defined Schemas folder. Cannot be set with -ALL.
- **-UDEJB**  
Exports *udejb.ejbfile.xml* for each enterprise bean in the project's EJB folder. Cannot be set with -ALL.

- **-v**  
Exports in verbose mode, with information on the status of each exported item.

#### **RELATED CONCEPTS**

“Model interchange with XML” on page 492

#### **RELATED TASKS**

“Using Object Builder from the command line” on page 659

“Exporting XML from the command line” on page 660

#### **RELATED REFERENCES**

“XML interchange files” on page 493

---

## Importing XML from the command line

There are two main cases that apply when you import XML into an Object Builder project from the command line:

- Importing files into a single project
- Importing files into multiple projects

### **Importing files into a single project**

To import from the command line into a single project, use the “obimport” on page 664 command:

```
obimport -p project_directory -d source_dir [xmlfile1 xmlfile2 ...
xmlfilen | -ALL] [-COE]
```

Files are imported into the project directory (-p) from the source directory (-d). If you do not specify a source directory, it defaults to the \Export subdirectory of the specified project directory. You can import specific files from the source directory, or all files (-ALL). You can select to stop the import if an error occurs, or continue despite the error (-COE, for continue on error).

For example:

```
obimport -Pe:\myproject -de:\anotherproject\Export -ALL
```

imports all XML files in e:\anotherproject\Export into the e:\myproject project.

When you are importing related files (for example, the files for Agent and the files for Customer, where Agent has a relationship to Customer), import all the files involved in a single step to ensure that the cross-references are correctly resolved. If you import related files in multiple steps, some information may be lost or incorrectly imported.

## Importing files into multiple projects

You can import files into multiple projects with `obimport`, using the `-X` option. This is especially useful for projects that have cross-dependencies, where you would otherwise have to import some sets of XML files multiple times.

Use the following command syntax to import into multiple projects:

```
obimport -x -d source_subdir project1 project2 ... projectn [-newuuid]
[-COE]
```

For each project listed, the XML files in the specified subdirectory are imported, and any cross-project references will be resolved. If the XML files are originally from the same model, specify `-newuuid` to assign new UUIDs as necessary to avoid cross-project identity conflicts. UUIDs are used within the XML file to uniquely identify elements. If you specify `-COE`, then the import process will continue even if errors are encountered in some XML files.

### RELATED CONCEPTS

“Model interchange with XML” on page 492

### RELATED TASKS

“Using Object Builder from the command line” on page 659

“Importing XML” on page 389

### RELATED REFERENCES

“XML interchange files” on page 493

“obimport”

## obimport

The `obimport` command can be used from the command line to import XML files to one or more specified projects.

There are two syntaxes you can use. The first is appropriate for importing into single projects. The second is appropriate for importing into multiple projects.

### Simple syntax

```
obimport -p project_directory -d import_dir [xmlfile1 xmlfile2 ...
xmlfilen | -ALL] [-COE]
```

Use the simple syntax when you are importing files into a single project.

### Cross-project syntax

```
obimport -x -d import_subdirectory project1 project2 ... projectn
[-newuuid] [-COE]
```

Use the cross-project syntax when you are importing files into multiple projects.

### Parameters

- **-p**  
The project directory you are importing into. If you set `-x`, you do not need to set `-p`: any directories you list on the command line will be assumed to be project directories.
  - **-d**  
The directory that contains the XML files you are importing. Defaults to the `\Export` subdirectory of the project directory you specified. If you set `-x`, then you cannot specify an absolute path: you must specify a path that is relative to the project directory, or accept the default.
  - *xmlfile1...xmlfileN*  
The XML files that you want to import. If you set `-x` or `ALL`, you cannot specify particular files: all files in the import directories will be imported.
  - **-ALL**  
Imports all the XML files in the specified import directory. If you set `-x`, or want to import particular XML files, you must not specify `-ALL`.
  - **-x**  
Specifies that you want to use the cross-project import syntax.  
If you set `-x`:
    - You cannot use `-p`. All directories listed on the command line will be assumed to be target project directories.
    - You cannot specify an absolute path with `-d`. You can specify a subdirectory to import from, or accept the default.
    - You cannot list particular files to import. `-ALL` is assumed.
    - You must list at least one target project directory.
- The import process will preserve any cross-dependencies among the files being imported, and create any necessary project dependencies.
- *project1...projectn*  
The projects that you want to import into. You must set `-x` for multiple project directories to be recognized, otherwise you can only set one project directory, with the `-p` option.
  - **-newuuid**  
Assigns new UUIDs for target models as necessary to avoid identity conflicts (for example, to perform model refactoring). Only use with `-x`. Specify this option when the source XML files come from the same project.

Every element within an XML file is identified by a UUID. When you move elements from a single model into multiple models, the UUIDs need to be recalculated to keep them unique.

- **-COE**

Continues the import process even if errors are encountered in the files being imported. If you do not set `-COE`, the import process will stop when it encounters an error.

**RELATED CONCEPTS**

“Model interchange with XML” on page 492

**RELATED TASKS**

“Using Object Builder from the command line” on page 659

“Importing XML from the command line” on page 663

**RELATED REFERENCES**

“XML interchange files” on page 493

---

## Importing IDL from the command line

You can import IDL files into Object Builder directly from the command line to create business object interfaces or data object interfaces in an Object Builder project.

**Note:** The IDL must be CORBA 2.2-compliant without IDL extensions. You can make sure the IDL files you are importing are valid by compiling them first. Object Builder will only import IDL files that are considered valid by the compiler.

To import IDL from the command line, use the `importidl` command:

```
importidl
 -p project_directory
 -d source_directory
 -ALL | -fimport_files
 -i include_directories
 -B0 | -D0 | -LOCAL
 -v]
```

Files are imported into the target project directory (`-p`) from the source directory (`-d`). You can either import all the files in the source directory (`-ALL`), or only specified files (`-f`). If the files you import include other files, you need to specify the directories that contain those include files (`-i`).



By default, imported files becomes business object interfaces (-BO). You can import them as data object interfaces by setting the -DO option.

If you have problems importing, set the -v option to display processing messages and errors, to help you debug the problem files.

The order of the options is not important, except for -ALL, which must come after -d *source\_directory*.

## Examples

**importidl -p E:\myProject -d E:\myIDLfiles -ALL**  
Imports all the IDL files in E:\myIDLfiles into the project E:\myProject as business object interfaces.

**importidl -v -p E:\myProject -d E:\myIDLfiles -ALL**  
Attempts to import all the IDL files in E:\myIDLfiles into the project E:\myProject as business object interfaces, and displays processing messages to help you locate and debug any problem files.

**importidl -p E:\myProject -d E:\myIDLfiles -f claimdata.idl  
policydata.idl -i E:\dependencies -DO**  
Imports E:\myIDLfiles\claimdata.idl and E:\myIDLfiles\policydata.idl, as well as any include files they reference in the E:\dependencies directory, into the project E:\myProject as data object interfaces.

## RELATED CONCEPTS

Interface Definition Language (*Programming Guide*)

## RELATED TASKS

- “Using Object Builder from the command line” on page 659
- “Creating a business object by importing an IDL file” on page 780
- “Creating a data object by importing an IDL file” on page 804
- “Creating a local-only object by importing an IDL file” on page 669

## importidl

The importidl command can be used from the command line to import IDL files into an Object Builder project.

The syntax of the importidl command is:

```
importidl
 -p project_directory
```

```
-d source_directory
-ALL | -f idl_files
-i include_directories
-BO | -DO | -LOCAL
-v
```

importidl has the following options:

- **-p** *project\_directory*  
A target directory (the project directory you are importing into). Required.
- **-d** *source\_directory*  
A source directory (the location of the files you are importing). Required.
- **-i** *include\_dir1 include\_dir2...include\_dirn*  
Any include directories (the location of any include files referenced by the files you are importing). Required if the files you import have references to include files.
- **-ALL**  
Imports all files in the source directory. You must specify this option, or **-f** with a list of files; you cannot specify both options. This option, if it appears, must come after the **-d** option.
- **-f** *file1 file2 ... filen*  
Imports the specified files from the source directory. You must specify this option, or **-ALL**; you cannot specify both options.
- **-BO**  
The contents of the imported files are added to the target project as business object files, business object modules, or business object interfaces. These objects appear in the User-Defined Business Objects folder. This is the default behavior, and you do not have to explicitly set this option. You cannot specify this option with either the **-DO** option, or the **-LOCAL** option.
- **-DO**  
The contents of the imported files are added to the target project as data object files, data object modules, or data object interfaces. These objects appear in the User-Defined Data Objects folder. You cannot specify this option with either the **-BO** option, or the **-LOCAL** option.
- **-LOCAL**  
The contents of the imported files are added to the target project as local-only object files, local-only object modules, or local-only object interfaces. These objects appear in the Local-Only Objects folder. You cannot specify this option with either the **-BO** option, or the **-DO** option.
- **-v**  
Debugs the imported files, and displays processing messages during the import. Specify this option if you experience problems importing files.

#### RELATED CONCEPTS

Interface Definition Language (*Programming Guide*)

#### RELATED TASKS

“Importing IDL from the command line” on page 666

### Creating a local-only object by importing an IDL file



If you have code already in IDL files, you can parse the code into Object Builder, and incorporate the classes, relationships, and code in the IDL files into your Object Builder application. A relationship is imported as methods.

**Note:** The IDL must be CORBA 2.2-compliant without IDL extensions. You can make sure the IDL files you are importing are valid by compiling them first. Object Builder will only import IDL files that are considered valid by the compiler.

To import an existing IDL file, follow these steps:

1. Under Tasks and Objects, select the Local-Only Objects folder.
2. From the folder’s pop-up menu, select **Import IDL**. The Import IDL wizard opens to the File Selection page.
3. Browse for and select the files you want to import. The files you select, and any files they include, will be parsed and imported into Object Builder.
4. Click **Next**. The Search Paths for Nested Files page opens.
5. From the Include directories pop-up menu, select **Add**. Browse for the directories you want searched.

When you import a file that includes other files (that is, a file with nested files), the import process will search for the other files in the directories you specify here.

6. Click **Finish**. The selected files (and files they include) are parsed into Object Builder, and the information in the IDL is added to the current project model as business object files, business object modules, and business object interfaces.

#### RELATED CONCEPTS

Interface Definition Language (*Programming Guide*)

#### RELATED TASKS

“Importing IDL from the command line” on page 666

Creating a business object by importing an IDL file

“Creating a data object by importing an IDL file” on page 804

---

## Importing enterprise beans from the command line

For redeployment, you can bypass the creation of the persistent object, and the mapping of the data object to the persistent object, if these steps are not required (as when you modify an enterprise bean, for example, by adding a method to it) by importing from the command line.

You can import BMP entity beans and session beans from the command line.

To import enterprise beans from the command line, follow these steps:

1. Install the EJB Deployment Tool.
2. Invoke the tool using the “*cbejb options*” on page 673 command: at a command prompt, type:

```
cbejb EJB_JARFile [-ob projDir] [-bean beanNames] -guisg
```

**Note:**The *cbejb* command has many other options that you can specify according to your requirements. If you do not use the *-guisg* option, the deployment process takes place entirely using only the command line, without launching any interface.

If you use the *-guisg* option with the *cbejb* command, Object Builder’s Import EJB JAR wizard opens. Follow these steps:

1. The Import EJB JAR wizard opens to the Enterprise Bean Selection page. It shows the deployment options (the directory in which your project’s models are stored, and the full directory path of the enterprise bean JAR file that you specified for import), and the process options (the code generation, compilation and build options) that you had specified with the command. Only the process options are editable. If you had specified the beans to be deployed with the *cbejb* command, they are listed in the **Enterprise Beans to Be Deployed** panel. You can change your selections.
2. Click **Finish**. The EJB Browser page opens. You can change the set of deployment platforms for the beans, and change the setting of the target build platform. You can select a bean from the **Enterprise Beans to Be Deployed** panel, and click **Properties** to either view or edit its definition.
3. The Import EJB JAR wizard opens to the Deployment Information page. Except for the deployment platforms, you can edit all other information on this page. You can provide a finder helper class name, a database name, and add or delete client and server JAR file dependencies.

**Note:** If you created the beans that you imported using VisualAge for Java, do not use the finder helper class that is generated by the tool. Instead, you must use the **FinderHelperGenerator** utility of the EJB server in CB to implement the finder helper class. For example, to generate a finder helper class for the AccountHome interface, use the command:

```
ejbfhgen com.ibm.ejs.doc.account.AccountHome
```

This command generates the finder helper class named `com.ibm.ejs.doc.account.AccountHome`.

For more information on finder helper classes, see Defining finder methods, and in particular, Creating finder logic in the EJB server (CB) in *Writing Enterprise Beans in WebSphere*.

4. If you are deploying either session or BMP entity beans, you can now click **Finish**. If you are deploying CMP entity beans, click **Next**.
5. The Container-Managed Bean Settings page opens. You set the sentinel values and string behavior for the Java primitive object types of the enterprise bean that you have selected for deployment.
6. Click **Next**. The Container-Managed Bean Field-Level Settings page opens. This page enables you to set the sentinel values and string behavior for each CMP field in the bean.

**Note:** If you do not set a sentinel value for a field of appropriate type, a default value is assumed. This is either the value that you set at the bean level using the previous page (the Container-Managed Bean Settings page), or if you had not explicitly set any values at the bean level, the default values at the bean level that are assumed from the EJB JAR-level settings.

7. Click **Finish**.

The enterprise beans in the EJB JAR file (*EJB\_JARFile*) that you selected for deployment, are imported into Object Builder for deployment.

During deployment, a deployed JAR file is generated from an EJB JAR file. The EJB Deployment Tool is thus used to deploy enterprise beans in the EJB server (CB) environment. The deployed JAR file (which is represented by a node in the Enterprise Beans folder in the Tasks and Objects pane of Object Builder) contains classes required by the EJB server.

#### **For session beans and BMP beans**

The remaining deployment process (code generation and build) continues automatically by means of the command line.

#### **For CMP beans and MQSeries application adaptor-backed session beans**

If `cbejb` is used with `-guisg`, you are presented with a dialog box from which you can select different actions to be taken:

- launch Object Builder if you want to change the default mappings between the data object and the persistent object;
- continue with code generation and build; or
- stop the deployment without generating and building code.

If `cbejb` is used without `-guisg`, you must type (at the command line) a specific character that corresponds to the same actions:

- o, to launch Object Builder (most often if you want to change the default mapping between the data object and the persistent object).
- c, to continue with model check, code generation and build
- x, to cancel out of the deployment process

**Important:** If you are working with MQSeries application adaptor-backed session beans, you **must** select the action to launch Object Builder.

If you selected the action to open Object Builder, you will find the deployed enterprise beans present in the Enterprise Beans folder under the JAR file from which they were imported. Besides, Object Builder creates the following objects for each enterprise bean that you select for deployment:

In the User-Defined Data Objects folder:

- the data object file
- the data object interface
- the data object implementation

**Note:** The IDL attributes of the data object interface correspond to the entity bean's container-managed fields.

In the User-Defined Business Objects folder:

- the associated business object
- the key
- the copy helper
- the managed object

**Warning:** It is recommended that you do not change the business object's implementation language, which is C++.

The following files that correspond to the imported JAR file are created in the Working\<>platform> directory:

- the Tie class
- the IDL files
- the Java files

You can now build and configure your application.

The EJB Deployment Tool also generates the data definition language (DDL) file used during installation of the enterprise bean into the EJB server (CB).

After you have completed these steps, saved the model, and exited from Object Builder, if you are working with either CMP entity beans, or session beans that are associated with an MQSeries application adaptor-backed business object, the **Do All** dialog box appears. You can specify further actions to be taken: run the Model Consistency Checker, generate all code for model,

build targets, or exit Object Builder. You also have the option of changing the default checks that the Consistency Checker will perform (click the **Checker Options** button, and select a different set of options, if you want to). Click **Start** to run the Model Consistency Checker, generate code, and compile it (if you have selected the corresponding check box options).

#### RELATED CONCEPTS

The EJB JAR file (*Writing Enterprise Beans in WebSphere*)

An introduction to enterprise beans (*Writing Enterprise Beans in WebSphere*)

The deployment descriptor (*Writing Enterprise Beans in WebSphere*)

#### RELATED TASKS

“Chapter 8. Working with enterprise beans” on page 391

“Importing enterprise beans into Object Builder” on page 392

“Deploying enterprise beans into a polymorphic home” on page 412

#### RELATED REFERENCES

“cbejb options”

## cbejb options



The EJB deployment tool (cbejb) can be invoked using the following syntax:

```
cbejb EJB_JARFile [options]
```

The tool has the following options:

```
[-rsp responseFile]
[-ob projDir]
[-nm] [-nc] [-ng] [-cc]
[-guisg]
[-platform [NT | AIX | OS390 | Solaris | HPUX]]
[-bean beanNames]
[-dllname DLLName beanName]
[-usecurdopo]
[-nouseraction]
[-dbname DBName [beanName]]
[-queryable beanName[:beanName]]
[-cacheddb2v52|-cacheddb2v61|-db2v61
 | -oracle|-informix|-jdbcaa beanName[:beanName]]
[-ccf | -hod | -eci | -appc | -exci |
 | -otma beanName[:beanName]]
[-finderhelper finderHelperClassName beanName[:beanName]]
[-polymorphichome beanName[:beanName]]
[-usewstringindo [beanNames]]
[-workloadmanaged [beanNames]]
[-family familyName [beanNames]]
[-clientdep deployed-jarFile [beanNames]]
```

```
[-serverdep deployed-jarFile [beanNames]]
[-sentinel [JavaPrimitiveObjectType=] sentinelValue
beanName[+CMFieldName][:beanName[+CMFieldName]]]
[-strbehavior strip|corba
beanName[+CMFieldName][:beanName[+CMFieldName]]]
```

**Note:** The EJB JAR file must be specified soon after the `cbejb` command. You can specify the other options in any order.

### Required option

- **EJB\_JARFile**  
This is a required parameter. It must be the first argument and it must specify a valid EJB JAR file as described in *Creating an EJB JAR file in Writing Enterprise Beans in WebSphere*.

### Substitution option

- **-rsp responseFiles**  
Specifies a file that contains a list of command-line options that you want to use along with the `cbejb` command. This file can contain all options except the EJB JAR file name. This makes the command you have to type at the prompt much shorter and easier. You can also use options that you have not specified in the response file, along with the `-rsp` option on the command line.

### Deployment-specific options

#### Optional options

- **-ob projDir**  
The relative or full path name of the project directory in which the generated files are stored. If this option is not specified, the current working directory is used as the project directory.
- **-bean beanNames**  
Identifies the enterprise beans (in the EJB JAR file) to be deployed. By default, all enterprise beans in the EJB JAR file are deployed. This is a list of one or more fully qualified enterprise bean names delimited by colons (:) (for example, `com.ibm.ejs.doc.transfer.Transfer:com.ibm.ejs.doc.account.Account`). For the enterprise bean name, specify either the bean's remote interface name, or the name of its deployment descriptor. *beanNames* is the class name of the EJB object interface. If you do not specify the bean names, all the enterprise beans in the JAR file for which the option is valid are selected for deployment.

#### Code generation and compilation options

- **-nm**  
Does not generate and import XML.



- **-nc**  
Does not compile and link the code.
- **-ng**  
Does neither XML generation, nor XML importing, nor code generation, but runs make. That is, it compiles and links the code. You can use this option instead of using `make -f all.mak` separately, after running `cbejb`. As an example, you can use the command `cbejb -My EJB.jar -nm -ng` instead of using `cbejb -My EJB.jar`, and then using `make -f all.mak`
- **-cc**  
Cleans the compiled code (runs `nmake` with the `clean` option: `nmake clean -f all.mak`), to remove make-generated files (the compiled and linked code). That is, it generates the makefiles for your updated build configuration, then deletes the existing DLLs and supporting files (such as `.def`). Finally, it rebuilds the DLLs according to the updated makefiles.

**Note the following points:**

- If you specify **-nm**, **-ng** and **-nc** in a command-line invocation, you must use the **-cc** option as well. That is, you must specify **-cc** if you specify either of these combinations:
  - `-ng -nc`
  - `-nm -ng -nc`
- If you do not specify any of these options, the tool generates and imports XML, does the code generation, and runs make, which compiles and links the code.

For changes in the makefiles that are generated, refer to the section Component Broker Support for Enterprise JavaBeans™ in the Component Broker Information's *Release Notes*.

- **-guisg**  
Presents a graphical user interface (GUI) with which you can input tool options, rather than use the command line for their input.
- **-platform**  
Specifies the platform for which code is to be generated. You can specify any one of NT, AIX, OS390, Solaris and HPUNIX, corresponding to the platform on which you want to build.  
**Note:** The platform that you specify will be automatically set in Object Builder's wizard pages (in the Deployment Platforms section). However, this selection will not automatically update Object Builder's **Platform** menu settings. You will have to manually select the platforms for which you want to view and edit code (**Platform > View**), those for which you want Object Builder to generate code (**Platform > Generate**), and those for which you want to apply development constraints according to the platforms for which you are developing your application (**Platform > Constrain**).

- **-dllname** *dllName beanName*  
Specifies the name of the DLL that should contain CB artifacts when the EJB class called *beanName* is deployed.
- **-usecurdopo**  
Uses the current mapping between the data object and the persistent object in the existing model. Does not bring up the Object Builder interface for the mapping.  
**Note:** Use this option when you are redeploying the beans, if you had already created a mapping between the data object and the persistent object, and you either have no need to, or do not want to change it during redeployment. The deployment will proceed automatically, without any interruption.  
When you first deploy CMP beans, you must not use **-usecurdopo**. The default data object to persistent object mapping is created, and Object Builder will be launched if requested, when you are prompted for the next action to be taken.
- **-nouseraction**  
Does not prompt you for any action after the data object to persistent object mapping is done.  
The deployment flow (codegen and build, if specified previously using the respective `cbejb` options) will continue by means of the command line only.  
If you do not use this option, you are prompted for the next action, which can be either `x`, to cancel out of the deployment process; `c`, to continue with model check, code generation and build; or `o`, to launch Object Builder (most often if you want to change the default mapping between the data object and the persistent object).  
If you do not use this option, and you use **-guisg**, you are presented with a dialog box from which you can select different actions to be taken:
  - launch Object Builder if you want to change the default mappings between the data object and the persistent object;
  - continue with code generation and build; or
  - stop the deployment without generating and building code.

### Backend storage type options

If none of the backend storage types are specified, DB2 version 5.2 with embedded SQL is used as the default backend for CMP entity beans.

 390 Note the following points:

If you specify 390 as a platform with the **-platform** option, you can only specify one of the following options for backend storage:

- `-db2v61`
- `-exci`

- -otma

Enterprise bean support is available on the Windows NT, AIX, Solaris, and HP-UX deployment platforms. It is also available for OS/390. However, Component Broker and WebSphere EJB clients will not be able to exchange information with CB OS/390 enterprise beans.

- **-dbname** *DBName [beanName]*  
Specifies the name of the database for CMP beans.
- **-cacheddb2v52** *beanName [:beanName]*  
Specifies CMP beans that use the DB2 version 5.2 backend with the Cache Service.
- **-cacheddb2v61** *beanName [:beanName]*  
Specifies CMP beans that use the DB2 version 6.1 backend with the Cache Service.
- **-db2v61** *beanName [:beanName]*  
Specifies CMP beans that use DB2 version 6.1 backend and embedded SQL.
- **-oracle** *beanName [:beanName]*  
Identifies CMP entity beans that require Oracle to store persistent data.
- **-informix** *beanName [:beanName]*  
Identifies CMP entity beans that require Informix that uses the Cache Service, to store persistent data.  
**Restriction:** A given transaction cannot access more than one Informix database per CB server. To involve two Informix databases in a transaction, you must access each database from a different server.
- **-jdbcaa** *beanName [:beanName]*  
Identifies BMP entity beans that require JDBC (Java Database Connectivity), along with the ability to carry out distributed transactions. -jdbcaa handles distributed transactions by enabling the bean implementation to connect to the CB Transaction Service. BMP beans that do not use the -jdbcaa option handle persistence by themselves: they may or may not use JDBC.  
If you do not specify a bean name, all beans in the JAR file use this option.
- **-hod** *beanName [:beanName]*  
Identifies CMP entity beans that use Host-on Demand (HOD) to store persistent data.

Must not be used for enterprise beans generated from the **PAOToEJB** tool.

- **-eci** *beanName [:beanName]*  
Identifies CMP entity beans that use the external call interface (ECI) to store persistent data.  
Must not be used for enterprise beans generated from the **PAOToEJB** tool.  
**Note:** CMP beans that connect to HOD or ECI backends use Session Service. All other CMP beans use Transaction Service.

- **-appc** *beanName* [:*beanName*]  
Identifies CMP entity beans that use advanced program-to-program communications (APPC) to store persistent data.  
Must not be used for enterprise beans generated from the **PAOToEJB** tool.
- **-exci** *beanName* [:*beanName*]  
Specifies CMP beans that use the EXCI backend.  
Must not be used for enterprise beans generated from the **PAOToEJB** tool.
- **-otma** *beanName* [:*beanName*]  
Specifies CMP beans that use the OTMA backend.  
Must not be used for enterprise beans generated from the **PAOToEJB** tool.
- **-ccf** *beanName* [:*beanName*]  
Specifies CMP beans that use the SAP backend, which is a common connector framework (CCF) backend.  
Can be used only with session beans, and with those that do not use the MQSeries application adaptor backend support.

### Run time-specific options

- **-finderHelper** *finderHelperClassName beanName* [:*beanName*]  
Specifies the finder helper class name for CMP entity beans. If unspecified, it is assumed that no finder helper class is provided by the deployer.  
You can use the **FinderHelperGenerator** utility to implement the finder helper class. For example, to generate a finder helper class for the AccountHome interface, use the command:  

```
ejbfhgen com.ibm.ejs.doc.account.AccountHome
```

  
This command generates the finder helper class named `com.ibm.ejs.doc.account.AccountHome`.  
For more information on finder helper classes, see Defining finder methods, and in particular,  
Creating finder logic in the EJB server (CB) in *Writing Enterprise Beans in WebSphere*.  
For more information on finder helper classes, see Defining finder methods in *Writing Enterprise Beans in WebSphere*.
- **-queryable** *beanName* [:*beanName*]  
Specifies CMP beans that are queryable, and generates a queryable CB home object. You can use this option only for CMP entity beans that store their persistent data in either a DB2, Oracle or Informix relational database. You must use either this option, or the **-polymorphichome** option if the finder helper class, which is used to implement the finder methods in a CMP entity bean, uses the CB Query Service.

#### Note the following points:

- You must use this option if you use either Oracle or Informix Cache Service as the backend storage type.

- You must not use this option if an entity bean uses CICS or IMS to store its persistent data.

By default, the interface definition language (IDL) interface of an enterprise bean's CB home extends the `IManagedClient::IHome` class, and the home implementation extends the `IManagedAdvancedServer::ISpecializedHome` class. An IDL interface of a queryable home extends the `IManagedAdvancedClient::IQueryableIterableHome` class, and the home implementation extends the `IManagedAdvancedServer::ISpecializedQueryableIterableHome` class.

In addition, the generated business object (BO) interface is marked as queryable. For queryable homes, the EJB client programming model remains unchanged; however, a Common Object Request Broker Architecture (CORBA), Java or C++ EJB client can treat the EJB home as an `IManagedAdvancedClient::IQueryableIterableHome` object.

For more information on queryable homes, see the *Advanced Programming Guide*.

- **-polymorphichome** *beanName[:beanName]*  
Specifies CMP beans that are polymorphic, and generates a polymorphic CB home object. You can use this option only for CMP entity beans that store their persistent data in either a DB2, Oracle or Informix relational database. You must use either this option, or the **-queryable** option if the finder helper class, which is used to implement the finder methods in a CMP entity bean, uses the CB Query Service.  
**Note:** If you do not specify either the **-queryable**, or **-polymorphichome**, the tool generates a regular CB home.
- **-usewstringindo**  
Maps the container-managed fields of an entity bean to the *wstring* IDL type (rather than the *string* type) on the data object. It is preferable to map to the *string* IDL type if the data source contains single-byte character data; it is preferable to map to the *wstring* IDL type if the data source contains double-byte or Unicode character data.
- **-workloadmanaged**  
Marks a CMP entity bean or a stateless session bean for use in a workload-managed container, or to generate for a BMP entity bean or a stateful session bean a home interface that is workload-managed.
- **-family** *familyName*  
Specifies the application family name to be generated. By default, this name is set to the name of the EJB JAR file, with the word Family appended.

- **-clientdep** *client-dependent*JARFile beanName [:beanName]  
Specifies the name of the deployed JAR file that the EJB client depends on at run time. It is used to compile the client JAR file. This is the file that uses the enterprise bean being deployed.

You must specify the full path name of the file. To create multiple client JAR files, specify this option for each JAR file that you want to create.

- **-serverdep** *server-dependent*JARFile beanName [:beanName]  
Specifies the name of the deployed JAR that the EJB server (CB) depends on at run time. This is the file that runs the deployed enterprise bean.

You must specify the full path name of the file. To create multiple server JAR files, specify this option for each JAR file that you want to create. You can also use this option to identify existing JAR files that contain classes that are required by the enterprise bean being deployed. If you do, the EJB server's CLASSPATH environment variable is automatically updated to include this specified JAR file.

**Note:** Only the options **-finderhelper**, **-family**, **-clientdep**, and **-serverdep** can be specified more than once with a different value. For example:

`-family fmy1 bean1:bean2 -family fmy2 bean3:bean4` is valid.

`-family fmy1 bean1:bean2 -family fmy1 bean3:bean4` will result in an error.

- **-sentinel** [*JavaPrimitiveObjectType*=]*sentinelValue* beanName[+CMFieldName][:beanName[+CMFieldName]]  
Specifies a sentinel value for either a CM field, a Java primitive object type of an enterprise bean, or a Java primitive object type of all deployed enterprise beans.

**Note:** There must not be a space either before or after the = (equal to) sign.

Examples:

```
-sentinel java.lang.Integer=-55555
```

specifies the value -55555 for the Java primitive type java.lang.Integer

```
-sentinel java.lang.Integer=-55555
com.ibm.ejb.cb.samples.hello.tier2.Hello:
com.ibm.ejb.cb.samples.travel.tier2.hotel.Hotel
```

specifies the value -55555 for the Java primitive object type java.lang.Integer for both the beans com.ibm.ejb.cb.samples.hello.tier2.Hello and com.ibm.ejb.cb.samples.travel.tier2.hotel.Hotel

```
-sentinel -55555
com.ibm.ejb.cb.samples.travel.tier2.hotel.Hotel+hotelNumber:
com.ibm.ejb.cb.samples.travel.tier2.hotel.Hotel+price
```

specifies the value -55555 for the two fields named hotelNumber and price, of the enterprise bean com.ibm.ejb.cb.samples.travel.tier2.hotel.Hotel.

- **-strbehavior strip | corba** beanName[+CMFieldName][:beanName[+CMFieldName]]

Specifies the string behavior for a CM field, all strings in an enterprise bean, or all strings in all deployed enterprise beans. Use the **strip** option to remove trailing spaces in the string. Use **corbato** to indicate that the string is a CORBA string.

Examples:

```
-strbehavior CORBA
```

Specifies that all strings in all deployed enterprise beans in the JAR file are to have the CORBA string behavior.

```
-strbehavior corba
 com.ibm.ejb.cb.samples.hello.tier2.Hello:
 com.ibm.ejb.cb.samples.travel.tier2.hotel.Hotel
```

Specifies that the string behavior of all strings in the two enterprise beans `com.ibm.ejb.cb.samples.hello.tier2.Hello`, and `com.ibm.ejb.cb.samples.travel.tier2.hotel.Hotel` is to be CORBA

```
-strbehavior strip
 com.ibm.ejb.cb.samples.travel.tier2.hotel.Hotel+hotelNumber:
 com.ibm.ejb.cb.samples.travel.tier2.hotel.Hotel+price
```

Strips trailing spaces from the two fields `hotelNumber` and `price` (which are of string type), of the enterprise bean `com.ibm.ejb.cb.samples.travel.tier2.hotel.Hotel`

**Note:** The **-sentinel** and **-strbehavior** options can be used to change the settings at the JAR level, the bean level, or the field level. You can only set bean- and field-level settings when you use Object Builder to deploy the beans (using the Container-Managed Bean Settings page, and the Container-Managed Bean Field-Level Settings page respectively).

#### RELATED CONCEPTS

The EJB JAR file (*Writing Enterprise Beans in WebSphere*)

An introduction to enterprise beans (*Writing Enterprise Beans in WebSphere*)

#### RELATED TASKS

“Importing enterprise beans into Object Builder” on page 392

“Deploying enterprise beans using the EJB Deployment Tool” on page 411

Developing and deploying enterprise beans with EJB server (CB) tools (*Writing Enterprise Beans in WebSphere*)

Creating finder logic in the EJB server (CB) (*Writing Enterprise Beans in WebSphere*)

---

## Importing edited source files from the command line

If you make changes to method implementations in the generated source code, you need to import the changes back into Object Builder, or the changes will be overwritten the next time you generate code.

When you import edited code, only changes to method implementations are applied. The import process recognizes method implementations by the comment block that delimits them:

### User-defined method

```
// <GeneratedMethodBody>
// <Body origin="user" xmi.uuid="...">
// Insert method modifications here
...
// End method modifications here
// </Body>
// </GeneratedMethodBody>
```

### Framework method

```
// <GeneratedMethodBody>
// <Body origin="ob" xmi.uuid="...">
// Insert method modifications here
...
// End method modifications here
// </Body>
// </GeneratedMethodBody>
```

The comment block is inserted by the code generation process. Any changes you make outside of these generated comment blocks are ignored.

You can import changes in batch mode, with the “importimpl” on page 683 command:

```
importimpl -p project_directory -f platform\file1
platform\file2...platform\filen
```

where *project\_directory* is the name of the project directory that contains your model, and the file names that follow are the business object implementation files that contain your changes, including their path relative to the current project’s \Working directory, or in other words, including the name of the platform subdirectory they are in. For example:

```
importimpl -pF:\MyProject -fNT\ClaimBO.cpp NT\AgentBO.java
```



You can also import code changes from within Object Builder, from the pop-up menu of the User-Defined Business Objects folder.

#### RELATED TASKS

“Using Object Builder from the command line” on page 659

“Importing edited source files” on page 385

“Editing a business object implementation” on page 792

“Generating code” on page 551

“Implementing methods” on page 752

#### RELATED REFERENCES

“importimpl”

## importimpl

The `importimpl` command can be used from the command-line to apply changes in edited source files to the original Object Builder project. You can only import changes to method bodies, which are bracketed in the source files with the following comment lines:

```
// Insert method modifications here
...
// End method modifications here
```

Any other changes in the source files will be ignored, and you must apply them separately, by editing the project in Object Builder.

The syntax of the `importimpl` command is:

```
importimpl -p project_directory -f platform\file1
platform\file2...platform\filen
```

`importimpl` has the following options:

- **-p** *project\_directory*  
A target directory (the project directory you are importing into). Should be the same project the source files were generated from. Required.
- **-f** *platform\files*  
The source files that you have changed, including, for each one, the name of the platform subdirectory it is in. Should be business object implementation files, with modifications only to method bodies, as marked by the comment syntax shown above.

#### RELATED TASKS

“Using Object Builder from the command line” on page 659

“Importing edited source files” on page 385

“Editing a business object implementation” on page 792

“Generating code” on page 551

“Implementing methods” on page 752

---

## Generating code from the command line



You can generate code from Object Builder from the command line, using the utility “obgen” on page 685.

The utility has the following command-line syntax:

```
obgen -p project_directory
 -d destination_directory
 -a [All|BO|DO|Make|SM]
 -t [NT|AIX|Solaris|390]
 -o [IVB_UNOPTIMIZE | IVB_OPTIMIZE | IVB_TRACE | IVB_TRACE_DEBUG
]
 -changed
 -linked
```

You can use obgen to generate code for specific objects (-a[BO|DO]), code for all objects (-aAll), makefiles (-aMake), or System Management DDL files (-aSM).

You can generate to any of the platforms supported by Object Builder (-t). The code will be generated into the appropriate *platform* subdirectory of the destination directory you specify (by default, code goes to the current project’s Working\*platform*\config directory).

Note:   If you find the process of generating code using obgen slowing down, shut down the run time components. That is, use the System Manager (SM) user interface to stop any CB application servers that are running on your system, and then stop the CBConnector service (bgmain process). This improves the performance of obgen by making more memory available.

If you are generating makefiles or SM DDL files, you can have them target one of four default configurations, with equivalent output directories, with the -o option. The code will be built into the appropriate *config* subdirectory of the destination\*platform* directory (by default, code will be built into the current project’s Working\*platform*\PRODUCTION directory). Makefiles will be configured to build targets with the appropriate options (unoptimized, optimized, trace-enabled, or trace- and debug-enabled).

You can select to generate every file for the objects you select (the default), or only changed files (-changed).

You can generate for just the current project (the default), or for all dependent projects as well (-linked).

#### RELATED CONCEPTS

“Projects and models” on page 17

#### RELATED TASKS

“Using Object Builder from the command line” on page 659

“Generating code” on page 551

“Generating a makefile” on page 556

“Generating the DDL files” on page 593

“Setting Object Builder preferences” on page 27

## obgen

You can generate code from Object Builder from the command line, using the obgen.bat utility.

The utility has the following command-line syntax:

```
obgen -p <project_directory>
 -d <destination_directory>
 -a [A11|B0|C0|L0|D0|Make|SM]
 -t [NT|AIX|Solaris|390|HPUX]
 -o [IVB_UNOPTIMIZE | IVB_OPTIMIZE | IVB_TRACE | IVB_TRACE_DEBUG
]
 -changed
 -linked
```

obgen has the following options:

- **-p <project\_directory>**  
The project directory you want to generate code for. Required.  
For example: -pE:\myproject
- **-d <destination\_directory>**  
The directory you want to generate code into. By default, code is generated into the listed project's \Working\platform directory (where platform is your target development platform, as set in Object Builder's **Platform** menu).
- **-a <object\_type>**  
The type of objects you want to generate. The object type must be one of the following:

- **All**  
Generate all objects in the project (equivalent to selecting **Generate > All** from the pop-up menus of the User-Defined Business Objects folder, the Build Configuration folder, and the Application Configuration folder).
- **BO**  
Generate all business objects (equivalent to selecting **Generate > Selected** from the pop-up menu of each business object interface and business object implementation).
- **CO**  
Generate all compositions (equivalent to selecting **Generate > Selected** from the pop-up menu of each composite business object interface, and composite business object implementation).
- **LO**  
Generate all local-only objects (equivalent to selecting **Generate > Selected** from the pop-up menus of each local-only object).
- **DO**  
Generate all data objects (equivalent to selecting **Generate > All** from the pop-up menu of the User-Defined Data Objects folder).
- **Make**  
Generate all makefiles (equivalent to selecting **Generate Makefiles** from the pop-up menu of the Build Configuration folder).
- **SM**  
Generate all SM DDL (equivalent to selecting **Generate > All** from the pop-up menu of the Application Configuration folder).
- **-t <platform\_name>**  
Specify platforms to generate code for. By default, code is generated for the current platform only. Options are **NT**, **AIX**, **390**, **Solaris** and **HPUX**. Also sets the platform subdirectory code will be generated into (for example, Working\NT\).
- **-o <configuration\_type>**  
Specify the output directory and configuration type the generated makefiles will target. Also sets the directory for generated SM DDL files. You can set the following:
  - **IVB\_UNOPTIMIZE**  
Makefiles will build into the Working\*platform*\NOOPT directory. Makefiles will be generated to build unoptimized targets (equivalent to setting the default configuration **Unoptimized**, in the environment preferences for Object Builder).
  - **IVB\_OPTIMIZE**  
Makefiles will build into the Working\*platform*\PRODUCTION directory. Makefiles will be generated to build optimized targets (equivalent to setting the default configuration **Production**, in the environment preferences for Object Builder). This is the default.

– **IVB\_TRACE**

Makefiles will build into the Working\*platform*\TRACE directory. Makefiles will be generated to build targets enabled for trace (equivalent to setting the default configuration **Trace**, in the environment preferences for Object Builder).

– **IVB\_TRACE\_DEBUG**

Makefiles will build into the Working\*platform*\TRACE\_DEBUG directory. Makefiles will be generated to build targets enabled for trace and debug (equivalent to setting the default configuration **Trace and debug**, in the environment preferences for Object Builder).

• **-changed**

Generate only code for files that have changed. By default, all files are regenerated.

• **-linked**






Generate code for the current project only (not for projects listed as dependencies). Generate makefiles that refer to the \Working directories of other projects for any dependencies on other projects. This is equivalent to setting the **Team environment** option in the Object Builder Preferences notebook (on the Tasks and Objects page of the notebook).

If you do **not** specify the **-linked** option, then code is generated for objects in the specified project, and for projects listed as dependencies, and for their dependencies, and so on. This is **not** equivalent to the **Stand-alone environment** option in Object Builder: when you generate code from within Object Builder, for a standalone environment, only objects in the current project, and their direct dependencies, are included.

For example, the command:

```
obgen -pe:\myproject -aBO
```

generates the code for all business objects in the project e:\myproject , code for business objects in projects it depends on, and code for projects they depend on, until all dependencies are fulfilled. The generated code for all projects is placed in the e:\myproject\Working\*platform* directory (where *platform* is your target development platform, as set in Object Builder's **Platform** menu or with the **-t** option).

Note:      If you find the process of compiling code slowing down when you use obgen, shut down the run time components. That is, use the System Manager (SM) user interface to stop any CB application servers that are running on your system, and then stop the CBConnector service (bgmain process). This improves the performance of obgen by making more memory available.

**RELATED CONCEPTS**

“Projects and models” on page 17

## RELATED TASKS

“Generating code from the command line” on page 684

“Generating code” on page 551

“Generating a makefile” on page 556

“Generating the DDL files” on page 593

“Setting Object Builder preferences” on page 27

## make options



When you build your DLLs from the command line (by making `all.mak`), you can set options that determine how and where your DLLs and JAR files are built. When a *make* option corresponds to a build option that you can set within Object Builder, it is described in “Build options” on page 568.

While you can configure a build without using these options (for example, by specifying a set of compile and link options in the Properties wizard that enable a DLL for debugging), the *make* options are applicable cross-platform, and will set the appropriate compile and link options for whichever platform you are targeting.

There are two categories of options available for *make*:

Configuration options	Target options
<ul style="list-style-type: none"><li>• <code>IVB_BUILD_VERBOSE=1</code> (page 689)</li><li>• <code>IVB_DYNAMIC_LINK=1</code> (page 690)</li><li>• <code>IVB_OPTIMIZE=1</code> (page 690)</li><li>• <code>IVB_TRACE=1</code> (page 691)</li><li>• <code>IVB_TRACE_DEBUG=1</code> (page 691)</li><li>• <code>IVB_UNOPTIMIZE=1</code> (page 692)</li><li>• <code>IVB_COMBINE_SOURCE=1</code> (page 692)</li></ul>	<ul style="list-style-type: none"><li>• <code>activex</code> (page 692)</li><li>• <code>all</code></li><li>• <code>cleandll</code> (page 693)</li><li>• <code>cpp</code> (page 693)</li><li>• <code>java</code> (page 693)</li><li>• <code>jcb</code> (page 693)</li><li>• <code>quickest</code> (page 693)</li></ul>



When used on the command-line, some of the configuration options set the output directory. For example, the command:

```
nmake -f all.mak IVB_TRACE=1
```

builds the resulting DLLs in the directory `Working\platform\TRACE`. Options specified on the command line override the default configuration selected within Object Builder. They do not override any DLL-specific options set within Object Builder.

**Note:** If your compile command fails due to an incorrect DB2 user ID and password error, run the following command before you run the make (AIX) or nmake (NT) command:

```
export IVB_DB2AUTH="USER test USING password"
set IVB_DB2AUTH=USER test USING password
```

  If you run into this problem on the Solaris or HPUX, make sure that you are authenticated to DB2 before you run the make command.

DLL-specific options do not affect the output directory. For example, if you set `IVB_UNOPTIMIZE=1` for ClaimS.DLL (within its DLL configuration wizard), and then build from the command line with `IVB_OPTIMIZE=1`, then the specific option for ClaimS.DLL overrides the global option in terms of what is built, but accepts the global target in terms of where it is built: all DLLs are built into Working\*platform*\PRODUCTION, including the unoptimized ClaimS.DLL

If you specify conflicting configuration options on the same command line, only the highest priority configuration option takes effect. The order in which they are listed does not matter. The order of priority is:

1. `IVB_UNOPTIMIZE=1` (equivalent to `IVB_OPTIMIZE=0`)
2. `IVB_OPTIMIZE=1` (equivalent to `IVB_UNOPTIMIZE=0`)
3. `IVB_TRACE=1`
4. `IVB_TRACE_DEBUG=1`

For example:

```
nmake -f all.mak IVB_TRACE=1 IVB_UNOPTIMIZE=1 IVB_OPTIMIZE=1
```



results in all DLLs being built in Working\*platform*\NOOPT, and all DLLs being built unoptimized (except where overridden in a particular DLL's configuration). The `IVB_TRACE=1` and `IVB_OPTIMIZE=1` options are ignored.

The predefined make options provided by Object Builder function as follows:

#### **IVB\_BUILD\_VERBOSE=1**

The DLLs are compiled and linked with the maximum amount of feedback generated. Options set:

CPP Compile:

-  /Wall
-  -qinfo=all

- `390 -V -v`
- `SOLARIS -verbose=%all`
- `HP-UX -v`

Link:

- `WIN /VERBOSE`
- `AIX none` (listings exist in .map files)
- `390 none` (listings exist in .llst.files)
- `SOLARIS none` (listings exist in .map files)
- `HP-UX -Wl, -v`

javac:

- all platforms: `-verbose`

#### `WIN IVB_DYNAMIC_LINK=1`

The DLLs are linked dynamically with the VisualAge C++ run-time DLLs. Dynamic linking reduces the size of your DLLs, but makes them dependent on the presence of the VisualAge C++ DLLs. This is the default behavior. After linking, `dllrname` is called against the DLLs to rename them to equivalent versions shipped with CB (the server renames the DLLs, changing the prefix from CPP to SOM). As a result, the DLLs that are built are dynamically linked against our renamed version of these DLLs. This ensures that the DLLs are always there even though the Toolkit (and the compiler) are not required to exist on the server.

You must then package the run-time DLLs with your application:

CPP Compile:

- `/Gd+`






Only Windows needs the ability to switch between dynamic and static linking because the C/C++ run time is part of the compiler. On the other platforms, the run times are part of the operating system.

#### `IVB_OPTIMIZE=1`

You can also use `IVB_UNOPTIMIZE=0` instead. If specified on the command-line, the DLLs are built in the `Working\platform\PRODUCTION` directory. On by default. The DLLs are compiled with optimization, including inlining of code.



CPP Compile:

-  /O+
-  -O -Q
-  -O
-  -O
-  -O






javac:

- all platforms: -O

### IVB\_TRACE=1

If specified on the command-line, the DLLs are built in the Working\*platform*\TRACE directory. Defines the CBS\_TRACE\_DEBUG preprocessor macro, which then includes code that allows the DLL to send trace data to the Object Level Trace tool. Option set:

CPP Compile:




-  /DCBS\_TRACE\_DEBUG
-  -DCBS\_TRACE\_DEBUG
-  -DCBS\_TRACE\_DEBUG
-  -DCBS\_TRACE\_DEBUG
-  -DCBS\_TRACE\_DEBUG



The resulting executables and libraries are linked to the Object Level Trace library.

### IVB\_TRACE\_DEBUG=1

If specified on the command-line, the DLLs are built in the Working\*platform*\TRACE\_DEBUG directory. Options set:

CPP Compile:

-  /O- /Ti+ /Tm+ /DCBS\_TRACE\_DEBUG
-  -g -DCBS\_TRACE\_DEBUG
-  -g -DCBS\_TRACE\_DEBUG



-  -g -xs
-  -g]

The resulting executables and libraries are linked to the Object Level Trace library.

### **IVB\_UNOPTIMIZE=1**

You can also use `IVB_OPTIMIZE=0` instead. If specified on the command line, the DLLs are built in the `Working\platform\NOOPT` directory. The DLLs are compiled without optimization. Option set:

CPP Compile:

-  /0-
-  -qnoptimize

### **activex**

Builds ActiveX interfaces. Only available on Windows NT.

### **IVB\_COMBINE\_SOURCE=1**

This is the single file build speed-up process. If you specify `IVB_COMBINE_SOURCE=1` on the command line, the DLLs are built so that the header files are not processed multiple times. This reduces the time required for the build. This is equivalent to using the **Minimize C++ compiler invocations** option on the Makefile Generation page, when you specify your build preferences (**File > Preferences**, Tasks and Objects, Makefile Generation).

**Note:**When you use this single file build speed-up process you will see a warning message due to automatic changing of the `_import` and `_export` status symbols. This warning can safely be ignored. It is part of an internal mechanism to ensure that code in separate compilation units, if any, (that is, in separate DLL files) does not fail to link. This is particularly true if, for example, one file contains the definition of a function, and another one contains its actual implementation.

### **all**

Builds as follows:

NT	Unix	OS/390
<ul style="list-style-type: none"> <li>• IDL</li> <li>• C++</li> <li>• Java</li> <li>• JCB</li> <li>• ActiveX (if enabled in Preferences)</li> <li>• QuickTest (if enabled in Preferences)</li> </ul>	<ul style="list-style-type: none"> <li>• IDL</li> <li>• C++</li> <li>• Java</li> <li>• JCB</li> <li>• QuickTest (if enabled in Preferences)</li> </ul>	<ul style="list-style-type: none"> <li>• IDL</li> <li>• C++</li> <li>• Java</li> </ul>

### **cleandll**

Rebuilds DLLs (linking), without recompiling. This option is available on the command line, and can be used when you have made changes in the Build Configuration folder that require rebuilding your DLLs, but have not made any changes to your components that require regenerating or recompiling code. For example, if you move a managed object configuration from one DLL to another in Object Builder, you can rebuild from the command line with the `cleandll` option, to build the updated DLLs.

### **cpp**

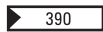
Builds IDL and C++.

### **java**

Builds IDL and Java.

### **jcb**

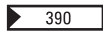
Builds Java client bindings.



Not available on OS/390.

### **quicktest**

Builds QuickTest targets.



Not available on OS/390.

### **RELATED TASKS**

“Configuring builds” on page 549

“Defining a client DLL” on page 552

“Defining a server DLL” on page 554

“Generating a makefile” on page 556

“Building the DLLs” on page 558

“Rebuilding DLLs” on page 565

“Setting Object Builder preferences” on page 27

## RELATED REFERENCES

“Default Configuration” on page 566

“Build options” on page 568

---

## obcheck



The obcheck command is a command-line model consistency checker tool that verifies models that you have built with the Object Builder. It helps you identify problems, such as dangling relationships, in your model before you compile or run your application. Running the obcheck command is the same as selecting **Check Model** from the **File** menu in Object builder.

The command format is:

```
obcheck modelName option1+ option2- option3+ ...
```

Each option is followed by a “+” or a “-” to turn the option on or off, respectively. For example, the following will check model integrity (-i+), but not platform constraints (-p-):

```
obcheck myModel -i+ -p-
```

You may use the following options with obcheck:

Option	Description	Default value
-3	CB OS/390-specific checks.	-3-
-a	Check changed framework and accessor methods.	-a+
-c	Check data object and container consistency.	-c+
-d	Locate files not in DLLs.	-d+
-e	Check for null sentinel mappings.	-e-
-f	Fix model integrity problems.	-f-
-i	Check model integrity.	-i-
-l	Display only local model messages.	-l-
-m	Check for empty method bodies.	-m+
-n	Locate object naming errors.	-n+
-o	Warn about dangerous method overrides.	-o+
-p	Check platform constraints.	-p+
-s	Check key attribute string behavior.	-s+
-t	Check for use of “invalidType”.	-t+
-w	CB workstation-specific checks.	-w+
-y	Display warning messages.	-y+

Option	Description	Default value
-z	Display informational messages.	-z+

**RELATED CONCEPTS**

“Troubleshooting” on page 905

**RELATED TASKS**

“Checking a model for consistency” on page 31

Verify models with model consistency checker (*Problem Determination Guide*)

**RELATED REFERENCES**

Consistency checker errors (*Problem Determination Guide*)



---

## Chapter 15. Object tasks

---

### Working with components

The following tasks deal with creating relationships between components, inheritance between components, importing and exporting information, and working with the various component objects:

- “Working with attributes”
- “Working with methods” on page 750
- “Working with constructs” on page 769
- “Working with business objects” on page 774
- “Working with data objects” on page 795
- “Working with keys” on page 825
- “Working with copy helpers” on page 829
- “Working with DB persistent objects” on page 832
- “Working with DB schema groups” on page 840
- “Working with DB schemas” on page 843
- “Working with PA persistent objects” on page 860
- “Working with PA schemas” on page 862
- “Working with managed objects” on page 869
- “Working with specialized homes” on page 875
- “Working with container instances” on page 883
- “Working with compositions” on page 884
- “Working with composite business objects” on page 891
- “Working with composite keys” on page 900
- “Working with the SQL View Editor” on page 850

#### RELATED CONCEPTS

Component (*Programming Guide*)

#### RELATED TASKS

“Developing in Object Builder” on page 19

---

### Working with attributes

Component attributes are defined in the business object interface. You can also define attributes for specific component objects (business object implementations, data object interfaces, data object implementations).

The get and set methods for component attributes are defined in the business object implementation and data object implementation. The mapping between attributes and data stores is accomplished using special framework methods in the data object and persistent objects.

The following tasks deal with attributes:

- “Adding an attribute” on page 699
- “Editing an attribute” on page 700
- “Mapping data object attributes to persistent object attributes” on page 730
- “Deleting an attribute” on page 701

#### **RELATED CONCEPTS**

“Attributes”

#### **RELATED TASKS**

“Working with components” on page 697

“Working with methods” on page 750

## **Attributes**

Public attributes of a component are defined in the business object interface or in a local-only object. You can define protected or private attributes in the business object implementation. When you change an attribute in a business object interface, the change is applied automatically to the business object implementation, and is applied to the key, copy helper, and managed object the next time you edit them (open and finish their properties wizard). When you change an attribute in a data object interface, the change is applied automatically to the data object implementation.

You can also define implementation-only attributes in the Business Object Implementation wizard or Data Object Implementation wizard. These attributes are not available in the IDL and are not exposed in the managed object for the component.

Attributes are defined in component objects as follows:

- Behavior:
  - Business object interface: IDL attributes
  - Business object implementation: get and set methods, in C++ or Java
  - Key: get and set methods, in C++ and Java
  - Copy helper: get and set methods, in C++ and Java
- Data:
  - Data object interface: IDL attributes
  - Data object implementation: get and set methods, in C++



- Persistent object: get and set methods, in C++
- Schema: table columns in a database, or methods of a procedural adaptor bean.

#### RELATED CONCEPTS

Component

“Get and set methods” on page 755

Attribute declarations (*Programming Guide*)

#### RELATED TASKS

“Working with attributes” on page 697

## Adding an attribute

You can explicitly define attributes in the business object interface, or in data object interfaces that are not associated with a business object. You can also add implementation-only attributes to either a business object implementation or a data object implementation. Implementation-only attributes are not exposed in the component’s managed object.

When you add objects from an interface, any elements necessary to support the interface’s attributes are added automatically. When you add a business object implementation and a data object interface in a single step, you do not need to define the data object attributes separately: you can select the data object attributes from a list of the existing business object attributes.

To define new attributes in an existing component, add them in the business object interface, edit the key and copy helper if you want the attribute to be used in those objects, and then edit the business object implementation to make it part of the data object. The changes are applied automatically to the implementations.

To add an attribute to an existing interface, follow these steps:

1. From the interface’s pop-up menu, click **Properties**.
2. In the interface’s wizard, click the page title and turn to the Attributes page.
3. From the pop-up menu of the Attributes folder on that page, click **Add**.
4. Define the attribute.
5. Click **Refresh**. The attribute is added to the Attributes folder.
6. Click **Finish**.

To make the new attribute part of an associated data object, follow these steps:

1. From the business object implementation’s pop-up menu, click **Properties**.

2. In the implementation's wizard, click the page title and turn to the Data Object Interface page.
3. Move the attribute from the **Business Object Attributes** list to the **State Data** list.
4. Click **Finish**. The attribute is added to the data object, including its data object implementation.

If appropriate, you can also edit the key and copy helper associated with the interface, and add the new attribute to them.

#### **RELATED CONCEPTS**

"Attributes" on page 698

#### **RELATED TASKS**

"Working with attributes" on page 697

"Editing an attribute"

"Editing a business object interface" on page 791

"Editing a data object interface" on page 817

#### **RELATED REFERENCES**

"Internationalization of data" on page 132

"Naming objects" on page 128

## **Editing an attribute**

When you edit attributes in an existing component, you must edit them in the business object interface. The change is automatically applied to the other objects in the component.

You can also edit an attribute in a data object interface, if it is not yet connected to a business object implementation.

To edit an attribute, follow these steps:

1. From the interface's pop-up menu, click **Properties**.
2. In the interface's wizard, click the page title and turn to the Attributes page.
3. Select an existing attribute under the Attributes folder.
4. Edit the properties of the attribute.
5. Click **Refresh**. The changes are applied.
6. Click **Finish**.

The change is automatically applied to the equivalent attribute in any related key, copy helper, implementation, or data objects.

#### RELATED CONCEPTS

“Attributes” on page 698

#### RELATED TASKS

“Working with attributes” on page 697

“Deleting an attribute”

“Editing a business object interface” on page 791

“Editing a data object interface” on page 817

#### RELATED REFERENCES

“Internationalization of data” on page 132

“Naming objects” on page 128

## Deleting an attribute

When you delete attributes from a component, you must delete them in the business object interface. References to the attribute in the rest of the component, and in other components will be automatically removed.

You can also delete an attribute from a data object interface, if it is not yet connected to a business object implementation.

To delete an attribute, follow these steps:

1. From the interface’s pop-up menu, click **Properties**.
2. In the interface’s wizard, click the page title and turn to the Attributes page.
3. Select an existing attribute under the Attributes folder.
4. From the attribute’s pop-up menu, click **Delete**. The attribute is removed.
5. Click **Finish**.

#### RELATED CONCEPTS

“Attributes” on page 698

#### RELATED TASKS

“Working with attributes” on page 697

“Editing an attribute” on page 700

“Editing a business object interface” on page 791

“Editing a data object interface” on page 817

## Setting sentinel values for null field values



CORBA has no concept of an attribute having a null value. If you want to maintain a null value read from a persistent object into a data object, and then write the null value back to the database, you must set up a sentinel value,

which Object Builder will substitute for a null value into the corresponding data object attribute. It will also substitute a null value for the sentinel when writing the data back to the database.

To set sentinel value for a data object attribute:

1. Select **Properties** from the Data Object Implementation's pop-up menu.
2. In the Properties wizard, select the Attribute Mapping page.
3. Select a data object attribute from the tree.
4. In the **Sentinel Value** field, input the value you want Object Builder to recognize as indicating null for this attribute. The value you input must be of the same type as the attribute. Enclose sentinel values for string attributes in quote marks (for example, "this is a null string value").

Be sure to choose a sentinel value that will not be a valid non-null value for the attribute.

The **Sentinel Value** field may be unavailable for the following reasons:

- The data object attribute has not been mapped to a persistent object attribute.
- The mapping is not primitive (that is, it is a key home or explode mapping); the attribute is a structure.
- The attribute has been marked as Not Null (for example, it is part of the key).

#### **Sentinel values for double-byte character strings**

Sentinel values for wstring attributes must be in the following form:

```
L"nullString"
```

If the wstring sentinel contains DBCS characters, you must set the /Sn+ C++ compile option on the server DLL.

1. In the Build Configuration folder, select the server DLL.
2. From the server DLL's pop-up menu, select **Properties**.
3. On the Name and Options page of the Properties wizard, type the following option in the **CPP Compile Options** field:

```
/Sn+
```

4. Click **Finish**.

#### **RELATED CONCEPTS**

"Null value tolerance with sentinel values" on page 155

#### **RELATED TASKS**

"Checking for null foreign key values" on page 298

"Editing a data object implementation" on page 819

"Mapping a data object to a DB persistent object" on page 703

---

## Mapping a data object to a persistent object

The following tasks deal with mapping a data object to a persistent object:

- “Mapping a data object to a DB persistent object”
- “Mapping a data object to a PA persistent object” on page 708
- “Mapping a data object to the parent’s persistent object” on page 711
- “Mapping a data object to the child’s persistent object” on page 712
- “Mapping data object attributes to persistent object attributes” on page 730
- “Customizing referential integrity” on page 714

### Note the following points:

- In the meet-in-the-middle case, when you associate an existing persistent object with a data object, Object Builder does the default mapping between data object attributes and persistent object attributes if the following properties hold true:
  - The attributes are of the same name
  - The attributes are of the same type
- In the top-down case, when you create a persistent object from a data object, Object Builder always does the default mapping for you, assigning to the attributes of the data object, persistent object attributes of similar name and type.

**Restriction:** When you map a data object to multiple persistent objects, you must map all the primary key attributes of the data object to the corresponding key attributes of each of the different persistent objects.

### RELATED CONCEPTS

“Attributes” on page 698

Data object (*Programming Guide*)

Persistent object (*Programming Guide*)

### RELATED TASKS

“Working with attributes” on page 697

“Working with data objects” on page 795

“Working with DB persistent objects” on page 832

## Mapping a data object to a DB persistent object

There are certain stages in development in which you can map a data object to a persistent object:

- When you associate a persistent object in the model with the data object implementation that you are creating (meet-in-the-middle)

### Note the following points:

- The persistent object has to use the same type of persistence as the data object implementation.
- You can customize the mapping of both attributes and special framework methods of the data object to relevant attributes and methods of the persistent object. Object Builder does not do the default mapping.
- When you create a persistent object and schema from a data object implementation (top-down)
 

**Note:** In this case you can customize only the mapping of the attributes of the two objects. The default mapping of attributes is done for you.
- When you create a data object from a persistent object. (bottom-up)
 

**Note:** As in the meet-in-the-middle case, you can customize the mapping of both attributes and special framework methods of the data object to relevant attributes and methods of the persistent object. In the bottom-up case, however, Object Builder does the default mappings for you.

**Restrictions:**

- Multiple data object attributes cannot be mapped to the same persistent object attribute.
- When you map a data object to multiple persistent objects, you must map all the primary key attributes of the data object to the corresponding key attributes of each of the different persistent objects.

**Meet-in-the-middle**

These are the preliminary steps you must follow before you can map a data object to a DB persistent object, when you associate a persistent object in the model to the data object implementation being created.

1. Create a schema by importing an SQL file.
2. Add a persistent object to the schema.
3. Add a data object implementation. The environment for the implementation must be **BOIM with any key**, and the implementation must not be transient (select any option except the **Transient** option from the Type of Persistence section).

**Note the following points:**

- To map a data object to a persistent object, there must be an association between the two objects, which you specify on the Associated Persistent Objects page of the Data Object Implementation wizard.
- As soon as you associate a persistent object with the data object, the Attributes Mapping page, the Methods Mapping page, and the Discriminators page are dynamically added to the wizard.

To define the mapping between the attributes of the data object and the persistent object, follow these steps:

1. If you are in the process of defining the data object implementation, proceed with step 2. If you have already defined the data object

implementation (you want to map the persistent object to a data object that exists in the model, and for which an implementation has been created), from the data object implementation's pop-up menu, select **Select Persistent Object**. The Data Object Implementation wizard opens to the Associated Persistent Objects page. Continue with step 3.

2. Go to the Associated Persistent Objects page.
3. Select either the persistent object, which you added to the schema that you just imported, or one which is associated with any other schema that you previously created.
4. Go to the Attributes Mapping page. Here, you can map the data object interface attributes to the attributes of the persistent object.

**Note:** Object Builder does the default mapping between business object attributes and data object attributes if the following properties hold true:

- The attributes are of the same name
- The attributes are of the same type

You can map a data object attribute to a persistent object attribute in one of the following ways:

- Using the primitive pattern
- Using the exploded mapping pattern (for structures, which are complex attributes)
- Using a foreign key
- Using a mapping helper

To define the mapping between the methods of the data object and the persistent object, follow these steps:

1. If you are in the process of defining the data object implementation, proceed with step 2. If you have already defined the data object implementation, from the data object implementation's pop-up menu, select **Select Persistent Object**. The Data Object Implementation wizard opens to the Associated Persistent Objects page. Continue with step 3.
2. Go to the Associated Persistent Objects page.
3. Select either the persistent object that you added to the schema that you just imported, or one that is associated with any other schema, which you previously created.
4. Go to the Methods Mapping page. When you define the mapping between methods, you actually "Customizing referential integrity" on page 714: you define the processing order of the persistent object methods that you associate with the data object's special framework methods `insert()`, `update()`, `retrieve()`, `del()` and `setConnection()`. These persistent object methods act directly on data in the persistent store (tables in the database).

**Note:** Object Builder does the default mapping between business object methods and data object methods if the following properties hold true:

- The method names are the same
- The method return types are the same
- The methods have the same number of parameters, and they are of the same type. (The names of the parameters are irrelevant.)

Of course, you can modify these mappings, if you want to.

Object Builder always provides the default mapping for the insert(), update(), del(), retrieve(), and setConnection() methods.

## 5. Click **Finish**.

**Attention:** If a persistent object is not created from this implementation but was created from another implementation and is used with this data object (you selected it on the Associated Persistent Objects page), you have to define the mapping between the data object methods and the persistent object methods (on the Methods Mapping page) for the special framework methods in the Methods pane to have implementations. The code for these methods gets modified according to the changes you make on the Methods Mapping page. Similarly, any changes you make to the mapping of attributes on the Attributes Mapping page, get recorded in the code for the attributes' get and set methods.

### Top-down

These are the preliminary steps you must follow before you can map a data object to a DB persistent object, when you are defining the persistent object and the schema from the implementation.

1. Add a data object implementation to a data object interface.
2. Indicate that the environment for the implementation is **BOIM with any key**.

To define the mapping between the attributes of the data object and the persistent object, follow these steps:

1. Add the persistent object and schema from the implementation: from the pop-up menu of the data object implementation, select the **Add Persistent Object and Schema** option.
2. Turn to the Attributes Mapping page of the Add Persistent Object and Schema wizard.
3. Change the mapping of the attributes, if you want to. Object Builder provides a default mapping between the attributes of the data object and those of the persistent object.

**Note:** If you have an attribute in the key that is either an unbounded string (a string whose size is not specified), or a bounded string with length in excess of 4000 characters, and you are creating a persistent object and schema for the data object, Object Builder does not provide a mapping



helper for the mapping of that attribute as the string does not contain the proper size information. You will have to provide one. Follow these steps:

- a. From the Attributes folder, select the persistent object that is mapped to this data object attribute.
- b. Type a length for this SQL type in the **Length** field of the schema column.

4. Click **Finish**.

**Note:** While you are defining the mapping of attributes using the Attributes Mapping page, you can also change the defaults that Object Builder sets for both the persistent object (names and types of persistent object attributes) and the schema (column names and SQL types for the columns).

### Bottom-up

Before you can map a data object to a persistent object in the bottom-up case, you must follow these steps:

1. Create a schema by importing an SQL file.
2. Add a persistent object to the schema.

To define the mapping between the attributes of the data object and the persistent object, follow these steps:

1. From the pop-up menu of the persistent object in the DBA-Defined Schemas folder, select **Add Data Object**. The Add Data Object wizard opens to the Names page, where you can specify the names and the file names for the data object interface and its implementation which you are adding.
2. Click **Next**. The Methods page opens, and you can define methods specific to the data object.
3. Click **Finish**.

The data object file, interface, and implementation are created in the User-Defined Data Objects folder, and are associated with the persistent object. At this point the default mapping exists between the data object and the persistent object. You can customize the mapping. Follow these steps:

1. From the data object implementation's pop-up menu, select **Properties**. The Data Object Implementation wizard opens.

Go to the Attributes Mapping page. Here, you can change the mapping between the attributes of the data object interface and those of the persistent object. You can use one of the three mapping patterns, and for each pattern, decide whether to provide a mapping helper.

**Note:** The *wchar* and *wstring* IDL data types of the data object are mapped by Object Builder to the LONG VARGRAPHIC column type in the persistent object, by default. When you are creating an application that involves wide character set data, and is required to store persistent object data in a column of data type

CHARACTER, in a DB2 table, you must write your own mapping helper. For an example, see the reference section “Mapping DBCS data types” on page 149.

1. Click **Next**. The Methods Mapping page opens, and you can override the default mappings of the special framework methods to the methods of the persistent object.
2. Click **Finish**.

**Note:** At this point the data object is stand-alone (it is not associated with a business object). To render the data object functional, you can associate it with an existing business object: first delete the business object’s associated data object interface (if any), and then, from its pop-up menu, select **Select Data Object Interface**, and specify the one that was created from the persistent object.

#### **RELATED CONCEPTS**

Data object (*Programming Guide*)

Persistent object (*Programming Guide*)

“Special framework methods” on page 758

#### **RELATED TASKS**

“Working with data objects” on page 795

“Working with DB persistent objects” on page 832

“Mapping data object attributes to persistent object attributes” on page 730

“Adding a data object implementation” on page 807

“Creating a DB schema by importing an SQL file” on page 844

“Defining relationships” on page 278

“Customizing referential integrity” on page 714

“Working with methods” on page 750

“Creating a child component” on page 306

“Editing a DB persistent object” on page 838

“Editing a DB schema” on page 855

#### **RELATED REFERENCES**

“DB2 data type mappings” on page 142

“Oracle data type mappings” on page 146

## **Mapping a data object to a PA persistent object**

Mapping a data object to a persistent object consists of mapping of attributes and methods from one object to the other. Mapping of attributes and methods is required to define the bonding between the objects. A data object attribute can be mapped to one or more persistent object attributes, and each special framework method of the data object can be mapped to one or more persistent object methods.

### **Restrictions:**

- When you use procedural adaptors, you are restricted to either the meet-in-the-middle scenario, or the bottom-up scenario since you cannot create a PA persistent object from a data object.
- When you map a data object to multiple persistent objects, you must map all the primary key attributes of the data object to the corresponding key attributes of each of the different persistent objects.

### Meet-in-the-middle

These are the preliminary steps you must follow before you can map a data object to a PA persistent object:

1. Create a PA schema and its associated PA persistent object by importing a PA bean.
2. Add a customized PA persistent object to the PA schema if you do not want to use the one Object Builder provides.
3. Add a data object implementation. (The environment for the implementation should be **Procedural Adaptors**.)

**Note:** To map a data object to a persistent object, there must be an association between the two objects.

4. Associate a PA persistent object with the data object using the Associated Persistent Objects page of the Data Object Implementation wizard (You can open this wizard by selecting **Properties** from the desired Data Object Implementation's pop-up menu).

As soon as you associate a persistent object with the data object, the Attributes Mapping page and the Methods Mapping page are dynamically added to the wizard.

**Note:** Object Builder does the default mapping between business object attributes and data object attributes if the following properties hold true:

- The attributes are of the same name
- The attributes are of the same type

To define the mapping between the attributes of the data object and the persistent object, follow these steps:

1. If you are in the process of defining the data object implementation, go to the Associated Persistent Objects page. If you have already defined the data object implementation and have associated a PA persistent object with it, then select **Select Persistent Object** from the data object implementation's pop-up menu. The Data Object Implementation wizard opens to the Associated Persistent Objects page.
2. Select either the persistent object that you added to the PA schema, or one that is associated with any other PA schema, which you previously created.
3. Turn to the Attributes Mapping page. Here, you can map the data object interface attributes to the attributes of the persistent object.

You can map a data object attribute to a persistent object attribute in one of the following ways:

- Using the primitive pattern
- Using the exploded mapping pattern (for some CORBA types that are not base types: for example, structures)
- Using a foreign key
- Using a mapping helper

To define the mapping between the methods of the data object and the persistent object, follow these steps:

1. If you are in the process of defining the data object implementation, go to the Associated Persistent Objects page. If you have already defined the data object implementation, and have associated a PA persistent object with it, then select **Select Persistent Object** from the data object implementation's pop-up menu. The Data Object Implementation wizard opens to the Associated Persistent Objects page.
2. Select either the persistent object that you added to the PA schema, which you just imported, or one that is associated with any other PA schema, which you previously created.
3. Turn to the Methods Mapping page. When you define the mapping between methods, you define the processing order of the persistent object methods that you associate with the data object's special framework methods `insert()`, `update()`, `retrieve()`, `del()`, `setConnection(long)` and `setWorkspaceID(long)`. These persistent object methods act directly on elements of transaction logic in the legacy business applications.  
**Note:** Object Builder does the default mapping between business object methods and data object methods if the following properties hold true:
  - The method names are the same
  - The method return types are the same
  - The methods have the same number of parameters, and they are of the same type. (The names of the parameters are irrelevant.)

Of course, you can modify these mappings, if you want to.

Object Builder always provides the default mapping for the `insert()`, `update()`, `del()`, `retrieve()`, `setConnection(long)` and `setWorkspaceID(long)` methods.

4. The methods that you defined for the data object appear in the User-Defined Methods folder. You can map each of them to a push-down method of the PA persistent object.

#### **RELATED CONCEPTS**

Data object (*Programming Guide*)

Persistent object (*Programming Guide*)

“Special framework methods” on page 758

“User-defined methods” on page 751

“Push-down methods” on page 759

#### **RELATED TASKS**

“Working with data objects” on page 795

“Working with PA persistent objects” on page 860

“Working with PA schemas” on page 862

“Adding a data object implementation” on page 807

“Mapping data object attributes to persistent object attributes” on page 730

“Defining relationships” on page 278

“Working with methods” on page 750

“Using push-down methods with PA persistent objects” on page 762

#### **RELATED REFERENCES**

“DB2 data type mappings” on page 142

“Oracle data type mappings” on page 146

## **Mapping a data object to the parent’s persistent object**

In top-down development, if the data object implementation that you are defining inherits from another (you select a parent implementation for the current one on the Implementation Inheritance page of the Data Object Implementation wizard), you can use one of two patterns (the key duplication pattern, or the attributes duplication pattern), to map the data object to a persistent object.

To map attributes of the data object to attributes of persistent objects that were created from the parent implementation, you use the flattening pattern. Follow these steps:

1. Turn to the Attributes Mapping page.
2. Select an attribute from the Attributes folder.
3. From the pop-up menu of the attribute, select the pattern for the mapping. You can select one of either the **Primitive**, **Key Home**, or **Explode** patterns.
4. Click the list button of the **Persistent Object Attribute** field and select an attribute that belongs to a persistent object that was created for the parent data object implementation.
5. Click **Finish**.

#### **Note the following points:**

- From the Attributes folder, you can select attributes that are specific to the data object implementation (those you define on the Attributes page of this wizard), as well as those defined for the business object and specified as

data object attributes (state data) on the Data Object Interface page of the Business Object Implementation wizard.

- The **Persistent Object Attribute** field lists not only the attributes of the parent implementation's persistent object, but also those of persistent objects belonging to the current implementation.
- You cannot select a parent persistent object for the current implementation on the Associated Persistent Objects page. That page is used only for associating with the implementation persistent objects that are at the same level of hierarchy as those that would be created directly from this implementation.

#### **RELATED CONCEPTS**

"Chapter 6. Inheritance" on page 299

"Choosing an inheritance pattern for persistence" on page 304

"Inheritance with a single datastore" on page 341

"Complex attributes and mapping patterns" on page 745

#### **RELATED TASKS**

"Creating a child component" on page 306

"Mapping a data object to a DB persistent object" on page 703

"Mapping a data object to a PA persistent object" on page 708

"Mapping a data object to the child's persistent object"

"Mapping data object attributes to persistent object attributes" on page 730

"Tutorial: Inheritance with a single datastore" on page 344

## **Mapping a data object to the child's persistent object**

In top-down development, if the data object implementation that you are defining inherits from another (you select a parent implementation for the current one on the Implementation Inheritance page of the Data Object Implementation wizard), you can use one of two types of patterns (flattening type or partitioning type), to map the data object to a persistent object.

The flattening pattern type is inheritance with a single table (Inheritance with a single datastore). There are two partitioning types. Which one you use depends on the type of inheritance between the current data object implementation and its parent:

- "Inheritance with key duplication" on page 322
- "Inheritance and overriding in helper objects" on page 300

### **Inheritance with key duplication**

When you use this pattern of mapping, you map attributes of the parent implementation, and all attributes of the current implementation to attributes of the persistent object that you are creating. Follow these steps:

1. From the pop-up menu of the data object implementation, select **Add Persistent Object and Schema**.

2. The Add Persistent Object and Schema opens to the Names page. Type the identification for the schema and the persistent object that you are defining.
3. Click **Next**. The Attributes Mapping page opens. Click the **Key Duplication**(vertical partitioning) button. Object Builder maps only those inherited attributes of the parent implementation which are key attributes of its business object, as well as all attributes of the current implementation, to attributes of the new persistent object. That is, for each of the key attributes of the parent implementation, and all attributes of the current implementation, it creates corresponding attributes in the persistent object, and does the mapping.
4. Click **Next**. The Columns and Attributes page opens. You can view the definition of the persistent object attributes and the corresponding schema columns that are created.
5. Click **Next**, and add any comments you want to, on the Comments page. You can type comments specific to the persistent object, the schema, and each of the schema columns.
6. Click **Finish**.

Unlike using inheritance with attribute duplication, with this pattern, the create, retrieve, update, and delete methods will be automatically mapped to both the parent data object implementation's persistent object and the child data object implementation's persistent object.

### **Inheritance with overriding persistence**

When you use this pattern of mapping, you map all attributes of the parent implementation, both key attributes and non-key attributes, and all attributes of the current implementation to attributes of the persistent object that you are creating.

Follow the same steps as for **Inheritance with key duplication**, but in step 3, select the **Attributes Duplication**(the horizontal partitioning pattern) button instead of the **Key Duplication** (the vertical partitioning pattern) button.

Unlike using inheritance with key duplication, with this pattern, the create, retrieve, update, and delete methods will be automatically mapped only to the child data object implementation's own persistent object.

Object Builder maps all inherited attributes of parent implementation - those which are key attributes of its business object as well as the non-key attributes, and all attributes of the current implementation, to attributes of the new persistent object. That is, for every one of the attributes that are inherited from the parent implementation, and all attributes of the current implementation, it creates corresponding attributes in the persistent object, and does the mapping.

**Note:** You can map a data object to persistent objects using these same patterns even in the meet-in-the-middle case (on the Attributes Mapping page of the Data Object Implementation wizard), when you associate a data object implementation with one or more persistent objects with matching persistence type that exist in the model. However, you will have to do the entire mapping on your own. For a sample mapping helper, see the section “Mapping DBCS data types” on page 149.

#### **RELATED CONCEPTS**

Data object (*Programming Guide*)

Persistent object (*Programming Guide*)

Schema (*Programming Guide*)

“Chapter 6. Inheritance” on page 299

“Choosing an inheritance pattern for persistence” on page 304

“Inheritance with key duplication” on page 322

“Inheritance with attributes duplication” on page 307

“Complex attributes and mapping patterns” on page 745

#### **RELATED TASKS**

“Creating a child component” on page 306

“Adding a persistent object and schema” on page 833

“Mapping a data object to a DB persistent object” on page 703

“Mapping data object attributes to persistent object attributes” on page 730

“Defining a child with key duplication” on page 325

“Tutorial: Inheritance with key duplication” on page 327

“Defining a child with attributes duplication” on page 309

“Tutorial: Inheritance with attributes duplication” on page 310

## **Customizing referential integrity**

When every value of each foreign key of a database is valid, the database is in a state of referential integrity. A foreign key is a subset of columns in a table whose values match at least one primary key, or unique key value of a row of the parent table. For a database to be in a state of referential integrity, a referential constraint must be met. This referential constraint is that the values of the foreign key are valid only if one of the following statements is true:

- The values of the foreign key appear as values of a parent key (the key of the parent table).
- Some component of the foreign key is null.

Referential constraints are optional and can be defined in CREATE TABLE and ALTER TABLE statements. Referential constraints are enforced by the database manager during the execution of INSERT, UPDATE, DELETE, ALTER TABLE ADD *constraint*, and SET CONSTRAINTS statements. Corresponding to these



statements, the data object has the set of special framework methods: `insert()`, `update()`, `retrieve()`, `del()`, and `setConnection()` that perform, respectively, the same tasks as the SQL statements:

- `insert()` is called when a table is created or altered.
- `update()` puts data back into the database when a table is altered.
- `retrieve()` gets data from the database.
- `del()` deletes a row in the table.
- `setConnection()` defines the database that is affected by the SQL statements in the `insert()`, `update()`, `retrieve()`, and `del()` methods. This method is implemented only if the data object implementation uses Embedded SQL.

You can customize referential integrity by specifying the processing order of methods so that they conform to constraints applied by the database.

**Note:** You can access the Methods Mapping page, where you can specify the processing order using the “Method Reordering” on page 769 controls only if there is a persistent object associated with the data object. The `insert()`, `update()`, `retrieve()`, and `del()` methods are not implemented for a transient data object implementation.

To specify the order of the persistent object methods, follow these steps:

1. Select the data object implementation that corresponds to the persistent object whose methods you want to arrange in a specific processing order.
2. From the data object implementation’s pop-up menu, select **Properties**. The Data Object Implementation wizard opens.
3. Click the arrow to the left of the page name, and select the Methods Mapping page from the list. The page opens.
4. The “Special Framework Methods” on page 768 folder contains the framework methods you can customize: `insert()`, `update()`, `retrieve()`, `del()`, and `setConnection()`.

**Note:** The `setConnection()` method is available only if you specified the Type of Persistence as **Embedded SQL** on the Behavior page of the Data Object Implementation wizard.

5. Select the method you want customized, display its pop-up menu and select **Add Mapping**. The **Persistent Object Method** field appears with the `del()` method selected by default. The method name has the form:  
*POInstance\_name.Method\_name*.
6. Click the list button and select the persistent object methods in the order you want them executed for the selected framework method. For each of the methods `insert()`, `update()`, `retrieve()`, `del()`, and `setConnection()`, you can select the **Always complete calling sequence (ignore warnings)** check box if you want the next method to be implemented even if a warning message is issued.

7. Click **Finish**. The ordered list of methods is saved. You can view it later by opening the same wizard. You can also view the order you specified by examining the method body in the Source pane after selecting the special framework method in the Methods pane.

**RELATED CONCEPTS**

Persistent object (*Programming Guide*)

**RELATED TASKS**

“Adding a data object implementation” on page 807

“Mapping a data object to a DB persistent object” on page 703

---

## Attribute mapping properties

The following topics describe the details of specific attribute mappings.

- “Data Object Attributes”
- “Key Attributes” on page 717
- “Mapping helper class” on page 719
- “Mapping Patterns” on page 722
- “Patterned Attribute Mapping Selection” on page 724
- “Persistent Object Attributes” on page 726
- “Schema Columns” on page 728

**RELATED CONCEPTS**

“Object Builder” on page 1

**RELATED TASKS**

“Working with attributes” on page 697

“Mapping data object attributes to persistent object attributes” on page 730

“Mapping a data object to a persistent object” on page 703

## Data Object Attributes



When you map a data object to a persistent object, you map the attributes of the data object to those of the persistent object.

You can do this on the Attributes Mapping page of either the Data Object Implementation wizard (select **Add Implementation** from the pop-up menu of the data object interface, and associate a persistent object with the implementation on the Associated Persistent Objects page), or the Add Persistent Object and Schema wizard (select **Add Persistent Object and Schema** from the pop-up menu of the data object implementation).

When you select a data object attribute in the Attributes folder on the Attributes Mapping page, the following sections are available:

- Data Object Attribute Properties
- “Mapping helper class” on page 719
- Sentinel value information

### Data Object Attribute Properties

This section is read-only and displays the properties of the data object attribute that is selected in the Attributes folder.

- **Data Object Attribute**  
This field shows the name of the data object attribute.
- **Type**  
This field shows the data type of the data object attribute.
- **Size**  
If the data type of the data object attribute is *string*, this field shows the string size. If you use the default mapping to LONG VARCHAR, it has a buffer size of 2000. You may specify another size.

### Sentinel value information

You can handle null field value input and output by specifying a sentinel value for the attribute to use in case a null is encountered.

#### RELATED CONCEPTS

Data object (*Programming Guide*)

Schema (*Programming Guide*)

Persistent object (*Programming Guide*)

“Mapping helper” on page 735

“Complex attributes and mapping patterns” on page 745

Null value tolerance with sentinel values

#### RELATED TASKS

“Mapping a data object to a persistent object” on page 703

“Setting sentinel values for null field values” on page 701

“Checking for null foreign key values” on page 298

## Key Attributes



When you map a data object to a persistent object, you map the attributes of the data object to those of the persistent object. If you want to use a key to define the mapping between the two objects, you map the attributes of the key to those of the persistent object.

You can do this on the Attributes Mapping page of either the Data Object Implementation wizard (select **Add Implementation** from the pop-up menu of the data object interface, and associate a persistent object with the

implementation on the Associated Persistent Objects page), or the Add Persistent Object and Schema wizard (select **Add Persistent Object and Schema** from the pop-up menu of the data object implementation).

When you map a data object to a persistent object, if you select **Key Home** from among the “Mapping Patterns” on page 722, and specify the key to be used to define the mapping, the key appears in the Attributes folder, beneath the data object attribute.

The Attributes Mapping page will then have the following fields:

### **Key**

This field shows the key selected for the mapping. You can click the list button, and select a different key, if you want to. The list shows the keys defined for the business object associated with this data object. The selected key and its attributes appear in the folder.

### **Home to Query**

Type the name of a home, which is to contain the object that is referenced by the data object. This is the object for which the selected key is the foreign key. The home must be the same as the one that you use for the configured managed object of the object reference that you will configure into the same application family.

### **Factory Name**

Type the name of a factory. The resulting factory key string will refine the factory key. Entry in this field is optional. However, if you provide a name, it must be the same as the factory name that you use when you configure the managed object (the name that you specify in the **Name in Factory Finding Registry** field in the “Home Options” on page 587 section on the Home page of the Configure Managed Object wizard).

### **Factory Finder Name**

This field is enabled only if you select the Key Home mapping pattern. Entry in this field is optional. You can specify the name of a factory finder that will be used instead of the default factory finder that Object Builder emits (*/host/resources/factory -finders/SERVERNAME-server-scope-widened*). You cannot specify substitution values such as *\*SERVERNAME* in this field.

### **Check for null**

Select this check box if you want to return NIL immediately if a retrieved key is null. If you leave this check box clear, and the key is null, then the `findByPrimaryKey` call may return a valid value instead of null, which can produce unexpected results.

**Note the following points:**

- The data object attributes that compose the key are shown below the key in the folder.
- From the pop-up menu of each of these key attributes, you can select either the **Primitive** or **Key Home** mapping as before, according to the conditions satisfied. You will not be able to select the **Explode** type of mapping as structures are not supported in keys.
- So, you can map each of the key attributes to a persistent object attribute, opting to use your own mapping helper, or accepting the default mapping helper that Object Builder provides in certain situations.
- The mapping from a key attribute to a persistent object attribute is one-to-one.

#### RELATED CONCEPTS

Data object (*Programming Guide*)

Schema (*Programming Guide*)

Persistent object (*Programming Guide*)

"Mapping helper" on page 735

"Chapter 6. Inheritance" on page 299

"Choosing an inheritance pattern for persistence" on page 304

"Inheritance and overriding in helper objects" on page 300

"Inheritance with key duplication" on page 322

"Complex attributes and mapping patterns" on page 745

"Foreign key patterns" on page 284

Query Service (*Advanced Programming Guide*)

"Null value tolerance with sentinel values" on page 155

#### RELATED TASKS

"Mapping a data object to a persistent object" on page 703

"Mapping attributes using a key" on page 732

"Mapping business object reference attributes" on page 744

"Creating a child component" on page 306

"Defining a child with key duplication" on page 325

"Tutorial: Inheritance with key duplication" on page 327

"Defining a 1-n relationship" on page 281

"Defining a foreign key pattern" on page 285

"Configuring a managed object" on page 588

"Checking for null foreign key values" on page 298

## Mapping helper class



When you map a data object to a persistent object, you map the attributes of the data object to those of the persistent object. You can also use an intermediate helper class to define the mapping between the attributes of the two objects.

You can do this on the Attributes Mapping page of either the Data Object Implementation wizard (select **Add Implementation** from the pop-up menu of the data object interface, and associate a persistent object with the implementation on the Associated Persistent Objects page), or the Add Persistent Object and Schema wizard (select **Add Persistent Object and Schema** from the pop-up menu of the data object implementation).

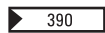
This section enables you to use an intermediate helper class to define the mapping between the attributes of the two objects. You can either provide your own mapping helper, or use the one provided by Object Builder. This section is available for any of the mapping patterns that you choose.

The default mapping helper (the class and its methods) is provided in the following cases:

- When a Stringified Object Reference (SOR) of the data object is mapped to a persistent object of type *char*. This happens if there exists an object reference between the selected object and another object.

**Warning:** When an SOR of the data object is mapped to a persistent object of type *char(n)*, DB2's embedded SQL truncates the *ByteString* that contains the handle at the first occurrence of a null, even if the field of type *char(n)* is declared as FOR BIT DATA. So, it is recommended that you map an object reference handle to either a VARCHAR(n), or a LONG VARCHAR instead of to a *char(n)* type.

- When a Stringified Object Reference (SOR) of the data object is mapped to a persistent object of type VARCHAR. This happens if there exists an object reference between the selected object and another object.
- When a data object attribute of type *string* is mapped to a persistent object attribute of type VARCHAR.
- When a data object attribute of type *wstring* is mapped to a persistent object attribute of type DB2VARGRAPHIC.



When one of the constrain platforms is **390** (you select **Platform > Constrain > 390**), *wchar* and *wstring* are not available for selection as an attribute type for your object.

- When a data object attribute of type *ByteString* is mapped to a persistent object attribute of type DB2VARCHAR.
- When a data object attribute of type *ByteString* is mapped to a persistent object attribute of type *char[]* (length greater than 0). Note the warning about mapping to the *char* type. Map the *ByteString* to a VARCHAR(n), or a LONG VARCHAR, especially for embedded SQL.
- When a data object attribute of type *string* is mapped to a persistent object attribute of type *char* (for SQL CHAR or CHARACTER columns).

There are also helpers to handle some common EJB-to-DB2 mappings (These are useful for EJB deployment scenarios.):

- string to java.sql.Date: data object attribute of type string is mapped to a persistent object attribute of type char [11].
- string to java.sql.Time: data object attribute of type string is mapped to a persistent object attribute of type char[9].
- string to java.sql.TimeStamp: data object attribute of type string is mapped to a persistent object attribute of type char[27].
- string to java.sql.BigInteger: data object attribute of type string is mapped to a persistent object attribute of type char[7].
- string to java.sql.BigDecimal: data object attribute of type string is mapped to a persistent object attribute of type char[7].

**Restriction:** Object Builder does not provide the default mapping between complex data types ( *any*, *wchar* and *wstring* and types defined as constructs, which include typedefs, structures, and unions) and DB2 database types. You must provide your own helper class for these mappings. No other kind of mapping is permitted. Structures may be mapped according to either the primitive pattern, in which you must provide a user-defined mapping helper, or the explode pattern, in which you may map the individual members of the structure to the persistent object. The members, in turn, obey the same mapping rules and offer the same mapping options based upon their datatypes as do the top-level attributes of a data object implementation.

This section has the following fields:

- **Class Name**  
Type a name for the class that is to contain the mapping methods between the data object and the persistent object, or accept the default.
- **PO to DO Method**  
Type a name for the mapping method from the persistent object to the data object, or accept the default.
- **DO to PO Method**  
Type a name for the mapping method from the data object to the persistent object, or accept the default
- **Sentinel Value**  
Type a value (the same type as the attribute) to serve as a sentinel value when the object is retrieving and writing null values to a database.  
**Note:**You can use IDL-defined constants that you created earlier as values for the initializers. However, Object Builder does not automatically include the IDL where the constant is defined: you must explicitly include it.
- **Java Mapping Helper**
- **Java Mapping Pair**

### **Attribute Reordering**

When you map an attribute of the data object to more than one persistent object attribute, even if they are of like type, it is recommended you use a

mapping helper. In this case, you have to ensure that the order in which the persistent object attributes are listed in the mapping helper method signatures is the same as the order in which they are mapped to the data object attribute. You can use the following buttons to reorder the attributes:

- **Move Up**

This button is available for selection when you have selected any of the persistent object attributes except the first one, from the Attributes folder. Each time you use this button, the selected attribute shifts one level higher in the tree.

- **Move Down**

This button is available for selection when you have selected any of the persistent object attributes except the last one, from the Attributes folder. Each time you use this button, the selected attribute shifts one level lower in the tree.

#### **RELATED CONCEPTS**

Data object (*Programming Guide*)

Schema (*Programming Guide*)

Persistent object (*Programming Guide*)

“Mapping helper” on page 735

“Complex attributes and mapping patterns” on page 745

“Null value tolerance with sentinel values” on page 155

#### **RELATED TASKS**

“Mapping a data object to a persistent object” on page 703

“Adding file adornments” on page 240

“Generating code” on page 551

“Setting sentinel values for null field values” on page 701

## **Mapping Patterns**

When you map a data object to a persistent object, you can select different ways or patterns in which the mapping is to be done. The pop-up menu of the data object attribute enables you to select one of the following mapping patterns:

- Primitive
- Key Home
- Explode

**Note:** The **Key Home** and **Explode** patterns are available only if certain conditions are satisfied.

### **Primitive**

This is the only option available, and the type of mapping provided by default by Object Builder if the data object attribute is a basic CORBA type, and the conditions for mapping as a key are not satisfied. Even if other



options are available, you can select this option if the data object attribute is not a complex type that requires a mapping helper.

**Note:** In general, all attributes are queryable. However, user-defined mappings and handle-mapped object references may result in poorer query performance because the query engine cannot push down these references to the database. Key Home mappings for object references, and Exploded mappings for structures usually perform well in queries.

### **Key Home**

When there is a reference between two objects, you can use one or more foreign keys to define the mapping between the data object and the persistent object by mapping the key attributes to the persistent object attributes. The mapping from the key to the persistent object is one-to-one.

In this situation, and even when there are nested object reference attributes (for example, if the key contains an attribute, which is itself an object reference), you can use the **Key Home** option.

**Note:**Your selection of a handle does not affect the default mapping of object references, which is the mapping using the **Key Home** pattern. It is relevant only if you override the default setting of the persistent object and schema's attribute mapping from **Key Home** to **Primitive**.

You will have to provide the name of a home, which is to contain the object that is referenced by the data object. This is the object for which the selected key is the foreign key.

**Note:**If you provide a name for the factory, it must be the same as the factory name that you use when you configure the managed object (the name that you specify in the **Name in Factory Finding Registry** field in the Home Options section in the Home page of the Configure Managed Object wizard).

**Restriction:** When you map attributes using a foreign key, and create a persistent object and schema from the data object implementation, it will not automatically create a foreign key in the schema.

**Warning:** If your component stores an object reference in a database, and you expect the database field to be queried by other objects or applications, then the object reference must be mapped from the data object to the persistent object as a key (**Key Home** option). Otherwise, queries will not be valid.

### **Explode**

This option is available only if the data object attribute is of a complex type such as a struct. Explode enables you to map each member of the structure to a relevant attribute in the persistent object.

### Note the following points:

- This version of Object Builder supports the **Explode** mapping only for complex attributes of the struct type.
- If a data object attribute is of type structure, you have the option of doing a primitive mapping with a user-defined mapping helper, or mapping each of the structure members to a persistent object attribute. (Select **Explode** from the pop-up menu of the attribute.)
- Further, if a structure member is of type interface, you can map that member using the foreign key mapping pattern.
- If there are any reference collection relationships defined for the object, Object Builder provides the mapping for the relationship attribute, with a default mapping helper.

#### RELATED CONCEPTS

Data object (*Programming Guide*)

Schema (*Programming Guide*)

Persistent object (*Programming Guide*)

“Mapping helper” on page 735

“Chapter 6. Inheritance” on page 299

“Choosing an inheritance pattern for persistence” on page 304

“Inheritance and overriding in helper objects” on page 300

“Inheritance with key duplication” on page 322

“Complex attributes and mapping patterns” on page 745

“Foreign key patterns” on page 284

Query Service (*Advanced Programming Guide*)

#### RELATED TASKS

“Mapping a data object to a persistent object” on page 703

“Creating a child component” on page 306

“Defining a child with key duplication” on page 325

“Tutorial: Inheritance with key duplication” on page 327

“Defining a 1-n relationship” on page 281

“Defining a foreign key pattern” on page 285

## Patterned Attribute Mapping Selection

This section appears only if the data object implementation from which you are creating the persistent object and schema inherits from another data object implementation.

You get this section on the Attributes Mapping page of the Add Persistent Object and Schema wizard (select **Add Persistent Object and Schema** from the pop-up menu of the data object implementation).

Use this section to have Object Builder map the attributes of the data object to the persistent object based on the type of inheritance pattern you want, using one of the following patterns:

- Key Duplication (vertical partitioning)
- Attributes Duplication (horizontal partitioning)

### **Key Duplication**

When you select this button, Object Builder maps only those attributes of the parent data object implementations that are key attributes of the business object, as well as attributes of the current implementation, to the attributes of the persistent object that is being created.

### **Attributes Duplication**

When you select this button, Object Builder maps all attributes of the parent data object implementations (this list of attributes includes key as well as non-key attributes of the business object that were designated as state data, as well as any attributes defined specifically for the parent implementation), and all attributes of the current implementation to attributes of the persistent object that is being created.

#### **RELATED CONCEPTS**

“Chapter 6. Inheritance” on page 299

“Choosing an inheritance pattern for persistence” on page 304

“Inheritance and overriding in helper objects” on page 300

“Inheritance with key duplication” on page 322

“Inheritance with attributes duplication” on page 307

“Inheritance with a single datastore” on page 341

Data object (*Programming Guide*)

Schema (*Programming Guide*)

Persistent object (*Programming Guide*)

“Mapping helper” on page 735

“Complex attributes and mapping patterns” on page 745

“Foreign key patterns” on page 284

Query Service (*Advanced Programming Guide*)

#### **RELATED TASKS**

“Creating a child component” on page 306

“Defining a child with key duplication” on page 325

“Tutorial: Inheritance with key duplication” on page 327

“Defining a child with attributes duplication” on page 309

“Tutorial: Inheritance with attributes duplication” on page 310

“Defining a child with a single datastore” on page 342

“Tutorial: Inheritance with a single datastore” on page 344

“Mapping a data object to a persistent object” on page 703

“Defining a 1-n relationship” on page 281

“Defining a foreign key pattern” on page 285

## Persistent Object Attributes

When you map a data object to a persistent object, you map the attributes of the data object to those of the persistent object.

You can do this on the Attributes Mapping page of either the Data Object Implementation wizard (select **Add Implementation** from the pop-up menu of the data object interface, and associate a persistent object with the implementation on the Associated Persistent Objects page), or the Add Persistent Object and Schema wizard (select **Add Persistent Object and Schema** from the pop-up menu of the data object implementation).

In the Data Object Implementation wizard, only the **Persistent Object Name** field is seen when you select a persistent object attribute in the Attributes folder. This field lists only those persistent objects that are associated with the implementation.

If you are in the Add Persistent Object and Schema wizard, and select a persistent object attribute in the Attributes folder, you see the following sections:

- Mapping Information
- Persistent Object Attributes
- “Schema Columns” on page 728

### Mapping Information

This field shows the full name of the persistent object attribute that is being created and mapped to the data object attribute.

### Persistent Object Attributes

This section shows details about the persistent object being created. This section has the following controls:

- PO Attribute
- Type
- Size
- PO Key

### PO Attribute

This column shows the attribute name for the persistent object, corresponding to the selected data object attribute. You can change the persistent object attribute’s name.

### Type

This field shows the data type of the persistent object attribute. If you change

the SQL type corresponding to the column in the Schema section, this field changes to show the corresponding implementation language data type.

**Note:** ...DB2VARCHAR is the special persistent object attribute type, which is a structure, and corresponds to the SQL types VARCHAR, VARGRAPHIC, LONG VARCHAR and LONG VARGRAPHIC. Details of the structure corresponding to each attribute can be viewed in the .hpp file generated from the persistent object created. This information will be useful if you are mapping a persistent object to a data object and are “Mapping attributes using a mapping helper” on page 738 that you have defined yourself.

### Size

This field shows the size of the buffer containing the string for persistent object attributes of the *string* type (*char[]*), or the size of the buffer within the structure for persistent object attributes of the ..DB2VARCHAR type. You can either accept the default or type a new value.

### Note the following points:

- The SQL types that correspond to the persistent object attribute type *char[]* are CHARACTER, GRAPHIC, DATE, TIME, and TIMESTAMP. The SQL types that correspond to the persistent object attribute type ...DB2VARCHAR are VARCHAR, LONG VARCHAR, VARGRAPHIC, and LONG VARGRAPHIC.
- The default size for GRAPHIC is 2 though the default length for this type is 1. This is because the persistent object attribute types are C++ types, and C++ automatically adds the null character (\0) at the end of strings.

### PO Key

This column contains a ✓ if there exists a key object that has the corresponding attribute as a key attribute; otherwise, it is blank. You can select the check box or clear it. However, if the corresponding schema column is a DB key, this attribute must be a PO key: you cannot clear the check box.

#### RELATED CONCEPTS

Data object (*Programming Guide*)

Schema (*Programming Guide*)

Persistent object (*Programming Guide*)

“Mapping helper” on page 735

“Complex attributes and mapping patterns” on page 745

#### RELATED TASKS

“Mapping a data object to a persistent object” on page 703

## Schema Columns

You get this section on the Attributes Mapping page of the Add Persistent Object and Schema wizard (select **Add Persistent Object and Schema** from the pop-up menu of the data object implementation).

This section shows details of the schema being created. It has the following controls:

- Column Name
- SQL Type
- ForBitData
- Length
- Scale
- DB Key
- Not Null

### Column Name

This field shows the column being created in the schema corresponding to the selected persistent object attribute. The name may or may not be in quotation marks. The default top-down behavior is to put it in quotation marks, but you can change this default on the Tasks and Objects page in the Preferences guide. You can change the name of the column, including adding or removing quotation marks from around it.

### SQL Type

This field shows the SQL data type of the column. You can click the list button and select a different type from the list. The attribute type of the persistent object changes accordingly.

### ForBitData

Click the field and select the check box if the schema column is to be used to store an object reference. The possible SQL types for columns that store Stringified Object References (SORs) are VARCHAR, CHARACTER or LONG VARCHAR. If the SQL type is not one of these types, this field is not editable. If a data object attribute is an object reference, and a schema and a persistent object are being added to its implementation, its **ForBitData** check box is selected by default.

### Note the following points:

- If this check box is selected, the .sql file generated from the schema will have the FOR BIT DATA information appended to the column type. For example, the line in the generated code corresponding to the column of type VARCHAR, which is used to store an SOR will be VARCHAR FOR BIT DATA.

- If this check box is not selected for a column that is to store object references, the string that is used to store the object reference will be treated as character data (SORs are considered binary data), and will undergo character conversions when stored in the database.
- The default SQL type that is used to store SORs is VARCHAR. If you choose to use CHARACTER or LONG VARCHAR instead, you must provide a mapping helper.
- If you change the default column type from VARCHAR to a non-string type (for example, INTEGER), the **ForBitData** check box is automatically cleared. If you then change it back to a string type that stores an SOR, the check box is not automatically selected: you must click in the **ForBitData** field and select the check box.

### Length

This column shows the length of the SQL type. For the DECIMAL type, this length indicates the precision, which is the total number of digits in the decimal. If you had defined the attribute type as *string* and you did not specify its length (when you defined the business object), it gets the default length of 32 characters.

### Scale

If the attribute is of the SQL type GRAPHIC or VARGRAPHIC, you can select one of the scales: K (Kilo= $10^3$ ), M (Mega= $10^6$ ) and G (Giga= $10^9$ ). For example, if the SQL type BLOB is of length 63 and scale K, its actual size is  $63 \times 10^3$ , represented by 63 K. If the SQL type is DECIMAL, you can select a number from 1 to 31 for the scale, indicating the number of digits after the decimal point.

### DB Key

This check box contains a  if there exists a key object that has the corresponding attribute as a key attribute; otherwise, it is blank. You can select the check box or clear it. Any column that is a database key must not be null and must be selected as a persistent object key as well. Object Builder, by default, selects the corresponding **Not Null** and **PO Key** check boxes if you select a column as the database key.

### Not Null

This check box contains a  if there exists a key object that has the corresponding attribute as a key attribute; otherwise, it is blank. You can select the check box if it is not selected, but you can clear it only if the attribute is neither a PO key nor a DB key.

### RELATED CONCEPTS

Data object (*Programming Guide*)

Schema (*Programming Guide*)

Persistent object (*Programming Guide*)

“Mapping helper” on page 735

“Complex attributes and mapping patterns” on page 745

#### RELATED TASKS

“Mapping a data object to a persistent object” on page 703

---

## Mapping data object attributes to persistent object attributes

You can map attributes of the data object to those of the persistent object using any one of the following methods:

- “Mapping attributes using the Primitive pattern” on page 731
- “Mapping attributes using a key” on page 732
- “Mapping attributes using a mapping helper” on page 738

The following tasks deal with mapping of complex attributes of the data object to persistent object attributes:

- “Mapping complex attributes using the Primitive pattern” on page 746
- “Mapping complex attributes using the Explode pattern” on page 748

The following task deals with mapping of business object reference attributes of the data object to persistent object attributes:

- “Mapping business object reference attributes” on page 744

#### Note the following points:

- In the meet-in-the-middle case, when you associate an existing persistent object with a data object, Object Builder does the default mapping between data object attributes and persistent object attributes if the following properties hold true:
  - The attributes are of the same name
  - The attributes are of the same type
- In the top-down case, when you create a persistent object from a data object, Object Builder always does the default mapping for you, assigning to attributes of the data object, persistent object attributes of similar name and type.

**Restriction:** When you map a data object to multiple persistent objects, you must map all the primary key attributes of the data object to the corresponding key attributes of each of the different persistent objects.

#### RELATED CONCEPTS

“Attributes” on page 698

Data object (*Programming Guide*)

Persistent object (*Programming Guide*)



#### RELATED TASKS

“Working with attributes” on page 697

“Working with data objects” on page 795

“Working with DB persistent objects” on page 832

## Mapping attributes using the Primitive pattern

When you map an attribute of the data object to an attribute of the persistent object of corresponding type, you do not have to use a mapping helper to provide the conversion, or use a key as an intermediate object to perform the mapping.

**Note:** In general, user-defined mappings are not queryable, but object references are always queryable. **Key Home** mappings for object references, and **Explode** mappings for structures are not primitive mappings, but can participate in object queries. So can handle mappings for object references. But, with handle-mapped object references, the query may not perform as well because the query engine cannot push down these references to the database.

To define a mapping using the default mapping pattern, follow these steps:

1. In the Tasks and Objects pane, select the data object implementation of the object whose attributes you want to map to the attributes of a persistent object.

**Note:** The data object implementation can be selected from either the User-Defined Business Objects folder or the User-Defined Data Objects folder.

2. From the implementation’s pop-up menu, select **Properties**. The Data Object Implementation wizard opens to the Name and Platform page. Click **Next**, or click the title of page, and select Attributes Mapping page from the list. The page opens.

**Note:** You can also define the mapping when you are defining the data object implementation, if you have selected an associated persistent object to be used with the data object on the Associated Persistent Objects page of this wizard, or when you add a persistent object and schema from a data object implementation.

3. From the Attributes folder, select the data object attribute that you want to map.
4. From the pop-up menu of the attribute, select **Primitive**.
5. Click the list button, and select the corresponding PO attribute from the **Persistent Object Attribute** list.

**Note:** When you click **Finish**, if there are any mappings whose types are not suitable for the default mapping pattern, you will be notified.

#### RELATED CONCEPTS

Data object (*Programming Guide*)  
Persistent object (*Programming Guide*)  
“Mapping helper” on page 735  
Query Service (*Advanced Programming Guide*)

#### RELATED TASKS

“Mapping a data object to a DB persistent object” on page 703  
“Adding a data object implementation” on page 807  
“Adding a persistent object and schema” on page 833

#### RELATED REFERENCES

“DB2 data type mappings” on page 142  
“Oracle data type mappings” on page 146

## Mapping attributes using a key

When there is a one-to-one relationship (a reference) between two objects, you can use one or more foreign keys to define the mapping between the data object and the persistent object by mapping the key attributes to the persistent object attributes. You can use this method to also map nested object reference attributes (for example, if the key contains an attribute, which is itself an object reference).

### Note the following points:

- In general, user-defined mappings are not queryable, but object references are always queryable. Key Home mappings for object references, and Exploded mappings for structures are not primitive mappings, but can participate in object queries. So Object Builder can handle mappings for object references. But, with handle-mapped object references, the query may not perform as well because the query engine cannot push down these references to the database.
- If the persistent object is created from the implementation itself, Object Builder uses the **Key Home** pattern as the default attribute mapping pattern, but you can change this by deleting the **Key Home** mapping, and creating a **Primitive** mapping instead.
- You can map an attribute using both the key mapping, and the primitive mapping: multiple mapping is supported.

To define the mapping, follow these steps:

1. In the Tasks and Objects pane, select the data object implementation of the object which has a reference to another object (that is, at least one of the attributes of the data object must have as its type an interface of another object, and the other object must have at least one key defined).

**Note:** You can select the data object implementation from either the User-Defined Business Objects folder or the User-Defined Data Objects folder.

- From the implementation's pop-up menu, select **Properties**. The Data Object Implementation wizard opens to the Name and Platform page. Click **Next**, or click the title of the page, and select Attributes Mapping page from the list. The page opens.

**Note:** You can also define the mapping when you are defining the data object implementation, if you have selected an associated persistent object to be used with the data object on the Associated Persistent Objects page, or when you add a persistent object and schema for the implementation, using the Add Persistent Object and Schema wizard.

- From the **Attributes** folder, select the data object attribute that you want to map. From its pop-up menu, you have the following choices: **Primitive** (the default mapping), **Key Home**, or **Explode** (if the data object attribute is a structure). The choices are not complementary: the mapping can be done using more than one pattern.
- Select the **Key Home** option. The first key defined for the referenced object is taken as the default, and appears (with its attributes) in the folder, beneath the data object attribute.

**Restriction:** When you create a DB persistent object and DB schema from this data object implementation, it will not automatically create a foreign key in the DB schema. That is, the FOREIGN KEY constraint will not be created in the table's .sql description file.

- You can change the key you want to use to define the mapping: Select the key in the Attributes folder. Click the list button of the **Key** field and select a key from the list of keys defined for the object.
- Type a name for the home, which is to contain the object that is referenced by the data object in the **Home to Query** field. The name that you specify must be the same as that which you specify as the name of the home on the Home page of the Configure Managed Object wizard when you later configure the managed object.
- Optionally, type the name of a factory in the **Factory Name** field. The name that you specify must be the same as that which you specify as the name in factory-finding registry on the Home page of the Configure Managed Object wizard when you later configure the managed object.
- Select a key attribute. (From its pop-up menu, only the **Primitive** mapping pattern is available. But, if the key attribute is an object reference, then it can be mapped using the **Key Home** pattern.) The **Mapping Helper Class** section appears. For each key attribute, you can optionally specify a mapping helper class and its methods to define the mapping to a persistent object attribute.

**Note the following points:**

- You must map all the key attributes of the data object. You can map each key attribute to only one attribute per persistent object.
- Key attributes can be mapped using either the **Primitive** pattern, or the **Key Home** pattern. The **Explode** pattern is not supported.

If you want to provide a mapping helper, follow these steps:

- Type the name of the mapping helper class in the **Class Name** field.
- Type the name of the method that does the mapping from the key attribute to the persistent object attribute in the **Key to PO Method** field.
- Type the name of the method that does the mapping from the persistent object attribute to the key attribute in the **PO to Key Method** field.
- From the key attribute's pop-up menu, select **Add Mapping**. The first of the defined persistent object attributes is mapped to the selected key attribute. You can change the persistent object attribute you want to use for the mapping: Click the list button of the **Persistent Object Attribute** field and select an attribute from the list of attributes defined for the persistent object. The **Type** field shows the type of the selected attribute. For *char* and *string* types, the **Size** field shows the size of the type.

**Note:** The mapping from a key attribute to a persistent object attribute is one-to-one. If the persistent object is associated with a schema, and the schema has at least one foreign key, the default one-to-one mapping between the key attributes and the persistent object attributes is provided by Object Builder.

#### RELATED CONCEPTS

Data object (*Programming Guide*)  
 Persistent object (*Programming Guide*)  
 Object relationships (*Programming Guide*)  
 "Mapping helper" on page 735  
 Query Service (*Advanced Programming Guide*)  
 "Foreign key patterns" on page 284  
 "Home" on page 581

#### RELATED TASKS

"Mapping data object attributes to persistent object attributes" on page 730  
 "Mapping business object reference attributes" on page 744  
 "Adding a data object implementation" on page 807  
 "Adding a persistent object and schema" on page 833  
 "Mapping a data object to a DB persistent object" on page 703  
 "Mapping attributes using a mapping helper" on page 738

“Working with specialized homes” on page 875

“Defining a foreign key pattern” on page 285

## Mapping helper

► CHANGED

A mapping helper is a class that contains mapping methods that are responsible for type conversion between attributes of a data object implementation and a persistent object. Every mapping helper class contains at least two static methods that always return void. These methods must be declared as public members of the class.

Type conversion is required for greater flexibility. For example, an attribute of type *string* may be required to map to an attribute of type VARCHAR, so that the length of the string is not a fixed, predetermined quantity. In this way, the string has the ability to take on different values, depending on the run-time allotment of the string's contents.

Object Builder provides the default mapping helper file (DB2MappingHelper.hpp, which contains the mapping helper class and its methods) in the following cases:

- When an object reference is stringified and mapped to a persistent object attribute of type *char*. This happens if there exists an object reference between the selected object and another object.  
**Warning:** When an object reference is stringified and mapped to a persistent object attribute of type *char[]*, DB2's embedded SQL truncates the *ByteString* that contains the handle at the first occurrence of a null, even if the field of type *char[n]* is declared as FOR BIT DATA. So, it is recommended that you map an object reference handle to either a VARCHAR[n], or a LONG VARCHAR instead of to a *char[n]* type.
- When an object reference is stringified and mapped to a persistent object attribute of type VARCHAR. This happens if there exists an object reference between the selected object and another one.
- When a data object attribute of type *string* is mapped to an ESQL persistent object attribute for a VARCHAR column. (A data object attribute of type *string* is normally mapped to a persistent object attribute of C++ string type. For example, a string of length 20 is mapped to *char[21]*.)
- When a data object attribute of type *wstring* is mapped to an ESQL persistent object attribute for a VARGRAPHIC column.

► 390

When one of the constraint platforms is 390 (you select **Platform** > **Constrain** > **390**), *wchar* and *wstring* are not available for selection as an attribute type for your object.

- When a data object attribute of type *ByteString* is mapped to a persistent object attribute of type DB2VARCHAR.

- When a data object attribute of type *ByteString* is mapped to a persistent object attribute of type *char[]* (length greater than 0). Note the warning about mapping to the *char* type. Map the *ByteString* to a VARCHAR(n), or a LONG VARCHAR, especially for embedded SQL.
- When a data object attribute of type *string* is mapped to a persistent object attribute of type *char* (for SQL CHAR or CHARACTER columns).

There are also helpers to handle some common EJB-to-DB2 mappings (These are useful for EJB deployment scenarios.):

- string to java.sql.Date: data object attribute of type string is mapped to a persistent object attribute of type char [11].
- string to java.sql.Time: data object attribute of type string is mapped to a persistent object attribute of type char[9].
- string to java.sql.TimeStamp: data object attribute of type string is mapped to a persistent object attribute of type char[27].
- string to java.sql.BigInteger: data object attribute of type string is mapped to a persistent object attribute of type char[7].
- string to java.sql.BigDecimal: data object attribute of type string is mapped to a persistent object attribute of type char[7].

**Note:** Whenever Object Builder provides the mapping helper, it is recommended that you use it rather than provide your own.

**Restriction:** Object Builder does not provide the default mapping between complex data types (including *any*, *Object*, and types defined as constructs, such as typedefs, structures, and unions) and DB2 database types. You must provide your own helper class for these mappings.

If the type of an attribute in the data object implementation is a *typedef*, Object Builder provides the same default mapping as the type that the typedef is aliased to. For example, for a typedef called *MyFloat* of the CORBA *float* type, the default mapping of the attribute to a new persistent object will be *double*.

When you are creating an application that involves wide character set data (for example *wchar* or *wstring*), and is required to store persistent object data in a column of data type CHARACTER, in a DB2 table, you must write your own mapping helper. For an example, see the reference section “Mapping DBCS data types” on page 149.

**Note:** By default, Object Builder maps the *wchar* and *wstring* IDL data types of the data object to the LONG VARGRAPHIC column type in the persistent object.

The mapping helper information can be viewed in the section “Mapping helper class” on page 719 on the Attributes Mapping page of the Data Object

Implementation wizard. The .cpp file generated from the data object implementation contains the mapping helper (DB2MappingHelper.hpp) in its include section.

If you want to provide your own mapping helper, you must create (outside Object Builder) a .hpp file, which contains the mapping helper class. When you define the mapping helper, follow these rules:

- Define both the mapping methods: from the persistent object to the data object, and from the data object to the persistent object, in the mapping helper class.
- Declare both mapping methods as public members of the class.
- Define both methods as inline methods to avoid linker errors.
- Define both methods as static methods.
- Define the return type of both methods as void.
- Pass the input arguments for both methods by const reference. Pass the input arguments as prescribed by the CORBA C++ type bindings for “in” arguments, and the output arguments as prescribed by the CORBA C++ type bindings for “inout” arguments. Be sure to obey the CORBA memory management rules within the implementation of your helper.
- For the persistent object to data object mapping method, use the following signature:

```
inline static void PO_to_DO_mapping_method_name(att1, att2, ...attn,
attribute_of_the_data_object)
```

where att1, att2,... attn are the persistent object attributes that are mapped to the data object attribute, and require the mapping helper.

- For the data object to persistent object mapping method, use the following signature:

```
inline static void DO_to_PO_
mapping_method_name(attribute_of_the_data_object, att1, att2,..., attn)
```

where att1, att2,... attn are the persistent object attributes that are mapped to the data object attribute, and require the mapping helper.

- Ensure that the .hpp file has the same name as the mapping helper class name.

**Note the following points:**

- When you use a mapping helper when a foreign key is used for the mapping, the mapping methods must be defined from the key to the persistent object, and from the persistent object to the key.
- When you map a data object attribute to a persistent object attribute using a mapping helper that you provide, you have to specify the name of the mapping helper file (without the extension) as the class name, and the names of the methods used for the mapping.

- Query Pushdown will be defeated over user-defined mapping helpers. You can still form OOSQL queries involving business object attributes that are ultimately mapped using a user-defined helper. However, the performance of the query may suffer since some or all of it must be executed in the object space.

#### RELATED CONCEPTS

Data object (*Programming Guide*)

Persistent object (*Programming Guide*)

Data object customization for cardinality relationships (Additional customizations) (*Programming Guide*)

#### RELATED TASKS

“Mapping attributes using a mapping helper”

“Mapping a data object to a DB persistent object” on page 703

“Mapping a data object to a PA persistent object” on page 708

“Adding a data object implementation” on page 807

“Adding a persistent object and schema” on page 833

#### RELATED REFERENCES

“DB2 data type mappings” on page 142

“Oracle data type mappings” on page 146

“Mapping DBCS data types” on page 149

## Mapping attributes using a mapping helper



When you map an attribute of the data object to an attribute of the persistent object of corresponding type, you do not have to use a mapping helper to provide the conversion. When you map attributes of different types, a mapping helper is required. The mapping helper is a class that contains mapping methods. Mapping methods provide the conversion between the attribute types of the two objects. You can either use the mapping helpers provided by Object Builder, or you can define your own.

#### Note the following points:

- Even if you map a single data object attribute to multiple persistent object attributes of the same type, it is recommended that you use a mapping helper so that any method that copies the persistent object attributes to the data object attribute copies all mapped persistent object attributes; not just the last one that is mapped.
- In general, user-defined mappings are not queryable, but object references are always queryable. **Key Home** mappings for object references, and **Explode** mappings for structures are not primitive mappings, but can participate in object queries. So can handle mappings for object references.



But, with handle-mapped object references, the query may not perform as well because the query engine cannot push down these references to the database.


Object Builder provides the default mapping helper (the class and its methods) in the following cases:

- When a Stringified Object Reference (SOR) of the data object is mapped to a persistent object attribute of type VARCHAR. To use this mapping helper, you should have followed these steps:
  1. Specified the type of one of the attributes to be a reference to an object on the Attributes page of the Business Object Interface wizard. For example, if you wanted the selected object (say Claim) to reference the Policy object, you should have selected the type of one of the attributes of the Claim object to be of type Policy PolicyInterf, where PolicyInterf is the interface defined for the business object named Policy.
  2. Selected a handle for storing pointers in the section: “Handle for Storing Pointers” on page 255 on the Behavior page of the Data Object Implementation wizard.
  3. Mapped the data object attribute that is of an object reference type to the persistent object attribute of type VARCHAR on the Attributes Mapping page of the Data Object Implementation wizard. The object reference handle is mapped to a FOR BIT DATA field in DB2.
- When a Stringified Object Reference (SOR) of the data object is mapped to a persistent object of type char.

To use this mapping helper, you should have followed the same steps as in the previous case, only replacing the persistent object attribute of type VARCHAR with one of type char.

**Warning:** When an SOR of the data object is mapped to a persistent object of type char(n), DB2’s embedded SQL truncates the ByteString that contains the handle at the first occurrence of a null, even if the field of typechar(n) is declared as FOR BIT DATA. So, it is recommended that you map an object reference handle to either a VARCHAR(n), or a LONG VARCHAR instead of to a char(n) type.
- When a data object attribute of type string is mapped to a persistent object attribute of type VARCHAR (A data object attribute of type string is normally mapped to a persistent object attribute of C++ string type. For example, a string of length 20 is mapped to char[21].) To use this mapping helper, you should have followed these steps:
  1. Specified one of the attributes of the business object to be of type string on the Attributes page of the Business Object Interface wizard.
  2. Changed the SQL type of the column name that corresponds to the business object attribute of type string, to VARCHAR on the Name and Attributes page of the Add Persistent Object and Schema wizard.

3. Mapped the data object attribute that is of type string to the persistent object attribute of type VARCHAR on the Attributes Mapping page of the Data Object Implementation wizard.
- When a data object attribute of type wstring is mapped to a persistent object attribute of IDL type DB2VARGRAPHIC (persistent object SQL type VARCHAR).

 **390** When one of the constrain platforms is **390** (you select **Platform > Constrain > 390**), wchar, wstring, and long long are not available for selection as an attribute type for your object.

- When a data object attribute of type ByteString is mapped to a persistent object attribute of type DB2VARCHAR (persistent object SQL type VARCHAR).
- When a data object attribute of type ByteString is mapped to a persistent object attribute of type char[] (length greater than 0). It is recommended that you map the ByteString to a VARCHAR(n), or a LONG VARCHAR, especially for embedded SQL.

You can view the mapping helper information on the Attributes Mapping page of the Data Object Implementation wizard when you select the mapped data object attribute in the folder. The .cpp file generated from the data object implementation contains the mapping helper file (DB2MappingHelper.hpp) in its include section.

#### Restrictions:



- Object Builder does not provide a default mapping from the data object implementation to the persistent object for the CORBA any, union, sequence or array data types; or for any typedef ultimately to the aforementioned types (except for sequence IManagedClient ByteString, which Object Builder does understand). In these cases, you must provide your own mapping helper class.
- If the type of an attribute in the data object implementation is a typedef, Object Builder provides the same default mapping as the type that the typedef is aliased to. For example, for a typedef called MyFloat of the CORBA float type, the default mapping of the attribute to a new persistent object will be double.
- When you are creating an application that involves wide character set data (for example wchar or wstring), and is required to store persistent object data in a column of data type CHARACTER, in a DB2 table, you must write your own mapping helper. For an example, see the reference section “Mapping DBCS data types” on page 149.

**Note:** The wchar and wstring IDL data types of the data object are mapped by Object Builder to the LONG VARGRAPHIC column type in the persistent object, by default.

- Object Builder does not provide the mapping helper in this case too: if you have an attribute in the key that is either an unbounded string (a string

whose size is not specified), or a bounded string with length in excess of 4000 characters, and you are creating a persistent object and schema for the data object. That is because the string does not contain the proper size information. You will have to provide a mapping helper. Follow these steps:

1. From the Attributes folder, select the persistent object that is mapped to this data object attribute.
2. Type a length for this SQL type in the **Length** field of the schema column.

-   For top-down development, the IDL data type `long long` maps to the `::CORBA::LongLong` data type in C++, and the `long` data type in Java, and is not supported on the OS/390 and Solaris platforms due to C++ compiler restrictions.
- For mapping helpers that convert to or from String types, the required buffer length is not available outside the helper. Therefore, the mapping helper will be passed a parameter that is an IDL inout String, which results in a C++ signature of `char*&`. The mapping helper is responsible for allocating the buffer itself.

Note that this is a change to the interface of the 2.0 generated code. Existing code that uses `char*` will still work for DB objects. Using `String_var`, inadvisable even before, is no longer permissible.

To map attributes using a mapping helper, follow the steps laid out, noting the recommendations that precede them:

**Recommendations:**

- Follow steps 4 through 11 in the sequence laid out.
  - If you want to provide your own mapping helper, follow steps 1 through 11.
  - If you want to use the mapping helper provided by Object Builder, follow steps 4 through 7.
1. Create (outside Object Builder) a `.hpp` file, which contains the mapping helper class.

When you define the mapping helper, follow these rules:

- Ensure that the `.hpp` files have the same name as the mapping helper class name.
- Define both the mapping methods: from the persistent object to the data object, and from the data object to the persistent object, in the mapping helper class.
- Declare both mapping methods as public members of the class.
- Define both methods as inline methods to avoid linker errors.
- Define both methods as static methods.
- Define the return type of both methods as `void`.

- Pass the input arguments for both methods by const reference. Pass the input arguments as prescribed by the CORBA C++ type bindings for “in” arguments, and the output arguments as prescribed by the CORBA C++ type bindings for “inout” arguments. Be sure to obey the CORBA memory management rules within the implementation of your helper.

- For the persistent object to data object mapping method, use the following signature:

```
inline static void PO_to_DO_mapping_method_name(att1, att2,
...attn, attribute_of_the_data_object)
```

where att1, att2,... attn are the persistent object attributes that are mapped to the data object attribute, and require the mapping helper.

- For the data object to persistent object mapping method, use the following signature:

```
inline static void DO_to_PO_
mapping_method_name(attribute_of_the_data_object, att1,
att2,..., attn)
```

where att1, att2,... attn are the persistent object attributes that are mapped to the data object attribute, and require the mapping helper.

**Note:** The mapping helper file can be located in any directory that is in your include search path.

2. In the Tasks and Objects pane, select the data object implementation of the object whose attributes you want to map to the attributes of a persistent object.

**Note:** The data object implementation can be selected from either the User-Defined Business Objects folder or the User-Defined Data Objects folder.

3. From the implementation’s pop-up menu, select **Properties**. The Data Object Implementation wizard opens to the Name and Platform page. Click **Next**, or click the arrow to the left of the page name, and select Attributes Mapping page from the list. The page opens.

**Note:** You can also define the mapping when you are defining the data object implementation, if you have selected an associated persistent object to be used with the data object on the Associated Persistent Objects page.

4. From the Attributes folder, select the data object attribute that you want to map to the persistent object.
5. From the data object attribute’s pop-up menu, select **Mapping**.
6. Click the list button of the **Persistent Object Attribute** field, and select the attribute of the persistent object that you want to map to this data object attribute. The persistent object attribute is added to the tree beneath the data object attribute.

**Note:** The order in which you map the different persistent object attributes to the data object attribute must be the same as the order in which they are listed in the mapping helper method signatures.

7. Select the data object attribute from the folder.

**Note:** If the mapped attributes meet the conditions for which Object Builder provides the default mapping helper, the **Map using helper class** option is automatically selected and the names of the mapping helper class and methods are shown in their respective fields. It is recommended that you use the default mapping helper provided. If you still want to provide your own mapping helper, follow steps 9 through 11.

8. Specify the mapping pattern: select the **Map using helper class** option.
9. Type the name of the mapping helper class in the **Class Name** field.
10. Type the name of the method that does the mapping from the attribute of the data object to the attributes of the persistent object in the **DO to PO Mapping Method** field.
11. Type the name of the method that does the mapping from the attributes of the persistent object to the attributes of the data object in the **PO to DO Mapping Method** field.

**Note:** Mapping helpers are also used when you map an attribute of the data object to an attribute of the persistent object using a foreign key. If a key attribute and the persistent object attribute being mapped are of different types, the mapping helper includes the methods that map between the key and the persistent object.

#### **RELATED CONCEPTS**

Data object (*Programming Guide*)

Persistent object (*Programming Guide*)

“Mapping helper” on page 735

Query Service (*Advanced Programming Guide*)

#### **RELATED TASKS**

“Adding a data object implementation” on page 807

“Adding a persistent object and schema” on page 833

“Mapping a data object to a DB persistent object” on page 703

“Mapping a data object to a PA persistent object” on page 708

“Adding file adornments” on page 240

#### **RELATED REFERENCES**

“DB2 data type mappings” on page 142

“Oracle data type mappings” on page 146

“Mapping DBCS data types” on page 149

## Mapping business object reference attributes

When an attribute of a business object is of the interface type (that is, it uses the interface of another business object as its type), it is called a reference attribute. If the primary key of the business object itself contains a business object reference attribute, this reference attribute is a nested attribute. You can map such both nested and non-nested attributes of the data object using either of the following methods:

- Map the attribute to a foreign key
- Map the attribute to a stringified handle

### Mapping to a foreign key

In this method, you map the primary key of the object reference attribute to the foreign key (which can be either simple or compound) of the referencing object's persistent object and table (schema).

To map the business object attribute to its primary key, follow these steps:

1. Map the business object reference attribute of the data object to its primary key attributes
2. Map the primary key attributes to primitive fields in the persistent object.

Follow the steps in the task "Mapping attributes using a key" on page 732

Using this method, Object Builder is able to handle an object reference mapping to database tables with PRIMARY KEY constraints that include FOREIGN KEY constraints to other tables.

**Note:** When you map attributes using a foreign key, and create a persistent object and schema from the data object implementation, it will not automatically create a foreign key in the schema. That is, the FOREIGN KEY constraint is not created in the table's .sql description file.

### Mapping to a stringified handle

To map the business object attribute to a stringified handle, follow these steps:

1. Map the reference attribute to a stringified handle
2. Map the stringified handle to a binary field (such as a VARCHAR(2000) FOR BIT DATA in DB2) in the persistent object.

Follow the steps in the task "Mapping attributes using a mapping helper" on page 738

#### RELATED CONCEPTS

Data object (*Programming Guide*)

Persistent object (*Programming Guide*)

"Mapping helper" on page 735

Query Service (*Advanced Programming Guide*)

Object relationships (*Programming Guide*)

“Foreign key patterns” on page 284

“Home” on page 581

#### **RELATED TASKS**

“Mapping data object attributes to persistent object attributes” on page 730

“Adding a data object implementation” on page 807

“Adding a persistent object and schema” on page 833

“Mapping a data object to a DB persistent object” on page 703

“Mapping attributes using a key” on page 732

“Mapping attributes using a mapping helper” on page 738

“Working with specialized homes” on page 875

“Defining a foreign key pattern” on page 285

## **Complex attributes and mapping patterns**

An attribute that is made up of multiple entities is called a complex attribute. For example, an attribute whose data type is a structure, is a complex attribute.

**Note:** Complex attributes in Object Builder are those whose data types correspond to the CORBA constructed types, template types, or complex declarators.

#### **Restrictions:**

- This release of Object Builder supports the mapping of structs in whole or by their individual fields. Other complex types can be mapped only if you define mapping helpers for them.
- Nested structures are not supported. However, structures whose members are themselves other structures are supported.

When you make a complex attribute persistent using a relational backend data store, you can use the following mapping patterns:

- Primitive
- Explode

**Primitive:** The structure is streamed out into a single column whose format is known to the client programmer. That is, you create a single attribute in the persistent object (and a corresponding column in the associated schema) to support a complex data object attribute. You then provide a mapping helper which can both convert (or, flatten) the structure into a field in the persistent object and schema, and do the reverse: convert the field in the persistent object and schema into a structure.

**Explode:** Each of the primitive members of the attribute is mapped to a different column in the table. This is the same table to which the data object

that contains the attribute is mapped. You must select the complex attribute members as distinct items from which to create a persistent object and a schema.

The members of the complex attribute are displayed on the Attributes Mapping page of the Data Object Implementation wizard. You can associate a member of a complex attribute with one or more persistent object attributes.

#### **RELATED CONCEPTS**

Persistent object (*Programming Guide*)

Schema (*Programming Guide*)

#### **RELATED TASKS**

“Working with DB persistent objects” on page 832

“Working with DB schemas” on page 843

“Adding a data object implementation” on page 807

“Mapping a data object to a DB persistent object” on page 703

“Editing a DB schema” on page 855

“Editing a generated SQL file” on page 857

#### **RELATED REFERENCES**

“DB2 data type mappings” on page 142

“Oracle data type mappings” on page 146

Type and constant declarations (*Programming Guide*)

## **Mapping complex attributes using the Primitive pattern**

Mapping of a complex attribute using the primitive pattern is similar to mapping an ordinary attribute of the data object to one in the persistent object.

**Note:** This release of Object Builder supports the mapping of structs in whole or by their individual fields. Other complex types can be mapped only if you define mapping helpers for them.

### **Top-down**

1. From the pop-up menu of the data object implementation in either the User-Defined Business Objects or the User-Defined Data Objects folder, select **Add Persistent Object and Schema**.
2. The Add Persistent Object and Schema wizard opens to the Names page. Name the persistent object and schema you are adding.
3. Click **Next**. The Attributes Mapping page opens. By default, Object Builder maps all struct data object attributes using the **Explode** mapping pattern.



4. Delete the default mapping. From the pop-up menu of **Explode** in the Attributes folder, select **Delete**. The exploded form of the mapping is deleted.
5. The pop-up menu of the complex attribute has two options: **Primitive** and **Explode**. Select **Primitive**. The complex attribute is mapped directly to the persistent object attribute.
6. Click **Finish**. A message informs you that the mapping between the complex attribute of the data object and the persistent object requires a mapping helper.
7. Click **Yes** to add the mapping helper. (Since, for this release, only structures are supported as complex types, and the primitive mapping of a structure (*struct*) requires a mapping helper.)

When you examine the properties of the persistent object (**Properties** from the pop-up menu of the persistent object), you will see that each of the complex attributes of the data object for which you defined the Primitive mapping is mapped to just one attribute of the persistent object, and therefore to the corresponding column in the schema (the backend database table).

### Meet-in-the-middle

1. From the pop-up menu of the data object implementation in either the User-Defined Business Objects or the User-Defined Data Objects folder, select **Properties**.
2. As long as the environment for the data object implementation is **BOIM with any key**, you can associate persistent objects with the implementation. Click the arrow to the left of the page name, and select Associated Persistent Objects page from the list. The page opens.
3. Add a persistent object instance, and click **Next**.
4. The Attributes Mapping page opens. Object Builder does not provide the default mappings. For each attribute in the Attributes folder, you can provide a mapping. The simple attributes have only the **Primitive** mapping option available from their pop-up menus; the complex attributes have both the **Primitive** as well as the **Explode** mapping options.
5. For each of the complex attributes, select the **Primitive** mapping pattern. Each complex attribute is mapped directly to the persistent object attribute.
6. Click **Finish**. A message informs you that the mapping between the complex attribute of the data object and the persistent object requires a mapping helper.
7. Click **No** to have the mapping exist in its primitive form, without the mapping helper.

### Bottom-up

Complex types in database columns are not supported. So, there is no mapping of complex attributes in the bottom-up case.

#### RELATED CONCEPTS

Persistent object (*Programming Guide*)  
Schema (*Programming Guide*)

#### RELATED TASKS

“Adding a data object implementation” on page 807  
“Mapping a data object to a DB persistent object” on page 703  
“Mapping data object attributes to persistent object attributes” on page 730

#### RELATED REFERENCES

“DB2 data type mappings” on page 142  
“Oracle data type mappings” on page 146

## Mapping complex attributes using the Explode pattern

The **Explode** pattern is the default mapping pattern provided by Object Builder when you map a complex attribute of a data object to an attribute of the persistent object. The complex attribute is exploded into its primitive component data elements and mapped across a set of columns in a table.

**Note:** This release supports the **Explode** mapping pattern only for attributes that are structures (type *struct*).

To map attributes using the **Explode** pattern, follow these steps:

1. From the pop-up menu of the data object implementation in either the User-Defined Business Objects or the User-Defined Data Objects folder, select **Add Persistent Object and Schema**.
2. The Add Persistent Object and Schema wizard opens to the Names page. Name the persistent object and schema you are adding.
3. Click **Next**. The Attributes Mapping page opens. By default, Object Builder maps all complex data object attributes using the **Explode** mapping pattern. In this pattern, each member of the complex attribute is mapped to a different persistent object attribute that is associated with the same database table.

**Note:** If you are editing the properties of the data object implementation (that is, you are redefining the mapping that is to be set up between the data object and any persistent objects that are to be created later from this data object implementation), from the pop-up menu of the complex attribute, select **Explode**. Then, continue with step 4. In this case, however, those persistent objects created before you edit the data object implementation retain their original mapping pattern.

4. Accept the default mapping, and click **Finish**. You can, however, change the attributes of the persistent object to which the members of the complex attribute of the data object are mapped.

When you examine the properties of the persistent object (**Properties** from the pop-up menu of the persistent object), you will see that instead of each of the complex attributes of the data object being mapped to a persistent object attribute, only the members of the complex attributes are mapped. So, each member of the data object's complex attributes has a counterpart in the corresponding database table.

**Note:** For the **Explode** mapping, it is not necessary to provide your own implementation of the mapping helper: you can use the one provided by Object Builder.

### Meet-in-the-middle

1. From the pop-up menu of the data object implementation in either the User-Defined Business Objects or the User-Defined Data Objects folder, select **Properties**.
2. As long as the environment for the data object implementation is **BOIM with any key**, you can associate persistent objects with the implementation. Click the arrow to the left of the page name, and select the Associated Persistent Objects page from the list.
3. Add a persistent object instance, and click **Next**.
4. The Attributes Mapping page opens. Object Builder does not provide the default mappings. For each attribute in the Attributes folder, you can provide a mapping. The simple attributes have only the **Primitive** mapping option available from their pop-up menus; the complex attributes have both the **Primitive** as well as the **Explode** mapping options.
5. For each of the complex attributes, select the **Explode** mapping pattern. Each of the component data elements of the complex attribute is mapped directly to a different persistent object attribute in the associated schema (table in the backend store: the database).
6. Click **Finish**.

The resulting mapping is the same as in the top-down case.

### Bottom-up

Complex types in database columns are not supported. So, there is no mapping of complex attributes in the bottom-up case.

#### RELATED CONCEPTS

Persistent object (*Programming Guide*)

Schema (*Programming Guide*)

**RELATED TASKS**

“Adding a data object implementation” on page 807

“Mapping a data object to a DB persistent object” on page 703

**RELATED REFERENCES**

“DB2 data type mappings” on page 142

“Oracle data type mappings” on page 146

---

## Working with methods

There are several different kinds of methods in Component Broker: user-defined methods (methods you define for a component or component object), the get and set methods that are automatically generated for attributes you define, framework methods that are automatically generated to support the server programming model, relationship methods that are generated whenever you define a relationship between two objects, and special framework methods that data objects use to handle persistent data.

For most cases, you only need to provide implementations for user-defined methods. Default implementations are provided for other methods.

Methods that you define at the interface level, get and set methods for attributes that you define at the interface level, and relationship methods are automatically designated as public. You can define private or protected methods only for an object’s implementation. You can see these modifiers that indicate the scope of access of the methods in the method signatures, when the methods are listed in their respective folders in the Methods pane.

The following objects display their methods in the Methods pane when you select them in the Tasks and Objects pane:

- business object interface
- business object
- data object interface
- data object implementation
- persistent object
- key
- copy helper

The following tasks deal with methods:

- “Implementing methods” on page 752
- “Adding an initializer method” on page 753
- “Editing user-defined methods” on page 754

- “Editing get and set methods” on page 756
- “Editing framework methods” on page 757
- “Editing special framework methods” on page 758
- “Using push-down methods with DB persistent objects” on page 761
- “Using push-down methods with PA persistent objects” on page 762
- “Handling exceptions thrown by PA bean push-down methods” on page 163
- “Customizing business object OOSQL implementation methods” on page 765
- “Customizing persistent object ESQL framework methods” on page 766
- “Deleting a method” on page 767

#### **RELATED CONCEPTS**

- “User-defined methods”
- “Get and set methods” on page 755
- “Framework methods” on page 757
- “Special framework methods” on page 758
- “External files for method bodies” on page 383

#### **RELATED TASKS**

- “Working with components” on page 697
- “Working with attributes” on page 697

## **User-defined methods**

You can define methods on the following objects:

- Business object interfaces  
Methods you define here are available to other components and applications, through the managed object. The method implementation is defined in the business object implementation.
- Business object implementations  
Methods you define here are specific to the implementation, and not exposed in the component’s interface.
- Data object interfaces  
Methods you define here are available to other objects, but are not exposed in the managed object. The method implementation is defined in the data object implementation.
- Data object implementations  
Methods you define here are specific to the implementation, and not exposed in the component’s interface.
- Local-only objects

Once you define a method on an object, it appears in the Methods pane when you click on the object.

When you define a method for an interface, its definition is automatically added to any associated implementation objects.

To provide the method body for a method, click on the business object implementation or data object implementation, and then click the method in the Methods pane. You can now type the method body directly in the Source pane.

You can also provide a method body by referencing an external file in the method's Method Implementation wizard (accessed from the method's pop-up in the Methods pane). The external file can be a template, with macros that you can substitute values for.

#### **RELATED CONCEPTS**

Business object (*Programming Guide*)

"External files for method bodies" on page 383

#### **RELATED TASKS**

"Adding a business object interface" on page 777

"Implementing methods"

"Adding an initializer method" on page 753

"Editing user-defined methods" on page 754

"Deleting a method" on page 767

## **Implementing methods**

Once you have defined a method interface and properties (either in the business object interface and implementation, or data object interface and implementation), you can add actual logic to the method body.

To add implementation code for a method you have defined, follow these steps:

1. In the User-Defined Business Objects folder, find the business object implementation or data object implementation (for example, CarPolicyBO) whose method you want to implement.
2. Click on the object. The following folders appear in the Methods pane:
  - User-Defined Methods
  - User-Defined Attributes
  - User-Defined Relationships
  - Framework Methods
  - File Adornments

The User-Defined Methods folder is expanded by default, and under it appear the methods you have defined for the business object.

3. Click a method.

The signature of the method appears in the Source pane. You can add an implementation to the signature directly in the Source pane, or you can edit the properties of the method and get its implementation from elsewhere.

4. From the method's pop-up menu, click **Properties**. The Method Implementation wizard opens to the Implementation page.
5. Select whether to use the implementation defined in the Source pane, or get the implementation code for the method from an external file.  
If you select to get the information from an external file, you can specify it as a template file, in which case you can use substitution macros, as defined on the Template File Macros page of the wizard.
6. Select whether to have a single implementation for all platforms, or use a separate implementation for each platform.  
If you select to have a different implementation for each platform, then the implementation you provide in the Source pane will apply only to the current platform selected in the **Platform > View** menu. You can switch the view to provide implementations for each of the platforms you intend to deploy on.
7. Click **Finish**. The selected behavior will be used the next time code is generated for the business object implementation.

#### **RELATED CONCEPTS**

"User-defined methods" on page 751

"Multi-platform development" on page 419

"External files for method bodies" on page 383 (Template files)

#### **RELATED TASKS**

"Working with methods" on page 750

"Importing edited source files" on page 385

## **Adding an initializer method**

If there is code that you need called when a component is loaded (the equivalent of a static initializer method in Java), you can put the code in a static method that gets called by a static attribute. When the component is loaded, the attribute is initialized, and calls the initializer method.

To add an initializer method that will contain code to be executed on the loading of the component, follow these steps:

1. Open the Business Object Implementation wizard (from the implementation's pop-up menu, click **Properties**).
2. Click the page title and turn to the Methods page.
3. Add a static method that returns type `int`.

4. Click the page title and turn to the Attributes page.
5. Add a static attribute of type `int`. In the attribute's initializer field, type a call to the static method.
6. Click **Finish**. The wizard closes.
7. In the Tasks and Objects pane, make sure the business object implementation is in focus.
8. In the Methods pane, expand the User-Defined Methods folder and select the method you defined. Its skeleton implementation appears in the Source pane.
9. In the Source pane, add to the method body any code you want called during initialization of the component. As the final step, return some `int` value to the calling attribute, to complete the attribute's initialization.
10. Click **File > Save**.

#### RELATED CONCEPTS

"User-defined methods" on page 751

"Attributes" on page 698

#### RELATED TASKS

"Working with methods" on page 750

"Adding an attribute" on page 699

"Implementing methods" on page 752

## Editing user-defined methods



To edit the signature of a user-defined method, follow these steps:

1. From the pop-up menu of the business object interface or data object interface where the method is defined, click **Properties** to open the Business Object Interface wizard.
2. Click the page title and turn to the Methods page.
3. Select the method under the Methods folder.
4. Make your changes to the method.  
**Note:**If the implementation language is Java, you can indicate that methods that you define for objects are to be synchronized. This is important to prevent concurrency problems in critical sections of code.
5. Click **Finish**.

You can edit the implementation of a user-defined method directly in the Source pane. If the Source pane is in read-only mode, then the implementation is being provided from an external file, as set in the method's wizard.

To access a method's wizard, follow these steps:



1. Select the business object implementation or data object implementation that implements the method.
2. In the Methods pane, select the user-defined method.
3. From the pop-up menu of the method, click **Properties** to open the Method Implementation wizard.
4. Make your changes in the wizard and click **Finish** to apply them.

You can add adornments to methods, to add descriptive comments to them. These adornments will be output to the generated files.

#### **RELATED CONCEPTS**

“User-defined methods” on page 751

#### **RELATED TASKS**

“Implementing methods” on page 752

“Importing edited source files” on page 385

Adding method adornments

## **Get and set methods**

Object Builder adds get and set methods to objects for each public attribute you define, as follows:

- **Business object implementation:**  
Has get and set methods for each public attribute in the business object interface. Read-only attributes, such as those that have been added to a key, have no set method.
- **Key:**  
Has get and set methods for each attribute that makes up the key.
- **Copy helper:**  
Has get and set methods for each attribute that makes up the copy helper.
- **Data object implementation:**  
Has get and set methods for each attribute defined in the data object interface.
- **Persistent object:**  
Has get and set methods for each attribute in the data object that it provides persistence for.

The implementations for get and set methods are provided by Object Builder, although you can edit them if necessary.

#### **RELATED CONCEPTS**

“Attributes” on page 698

“Special framework methods” on page 758

#### RELATED TASKS

“Editing get and set methods”

## Editing get and set methods

▶ CHANGED

Get and set methods provide access to attributes defined in either a business object interface or a data object interface.

To edit the signature of a get or set method, you must edit the attribute it represents, in the business object interface or data object interface.

By default, the get and set implementations are read-only. To edit the implementation of a get or set method (not recommended), follow these steps:

1. Click on the business object implementation or data object implementation in the Tasks and Objects pane.
2. In the Methods pane, locate the get or set method under the Attributes folder.
3. From the pop-up menu of the attribute, click **Properties**. The Method Implementation wizard appears, open to the Implementation page.
4. Select the check box **Method body is the same for all platforms** if you plan to provide your own code for the method body, and you want it to be the same for all platforms.
5. Click **Use the implementation defined in the Source pane**.  
You could also click **Use an external file**, and select an external file that contained the method implementation.
6. Click **Finish**.

At any time, you can reset the implementation by opening the Method Implementation wizard and clicking **Return to Default**. If you want to return to using only the default, click **Use the implementation provided by Object Builder** to put the implementation back into read-only mode.

You can add adornments to get and set methods, to add descriptive comments to them. These adornments will be output to the generated files.

#### RELATED CONCEPTS

“Get and set methods” on page 755

“Attributes” on page 698

#### RELATED TASKS

“Working with methods” on page 750

“Importing edited source files” on page 385

“Editing an attribute” on page 700

Adding method adornments

## Framework methods

Framework methods are added to an object by Object Builder. Generally, framework methods are only called by other framework methods, or by Component Broker services.

Framework methods provide the functionality your objects need to work in a Component Broker distributed environment.

There are also special framework methods, which are a particular kind of framework method that let a component access its persistent data.

### RELATED CONCEPTS

“Special framework methods” on page 758

### RELATED TASKS

“Editing framework methods”

## Editing framework methods

Framework methods are added to an object by Object Builder. Generally, framework methods are only called by other framework methods, or by Component Broker services.

You cannot edit the signature of a framework method. Some methods are added or deleted based on your selections in the component wizards.

By default, the implementations of framework methods are read-only. To edit the implementation of a framework method (not recommended), follow these steps:

1. In the Tasks and Objects pane, click on the object whose framework methods you want to edit.
2. In the Methods pane, locate the framework method under the Framework Methods folder.
3. From the pop-up menu of the attribute, click **Properties**. The Method Implementation wizard appears, open to the Implementation page.
4. Click **Use the implementation defined in the Source pane**.

You could also click **Use an external file**, and select an external file that contained the method implementation.

As long as this option is selected, the method’s implementation will be determined by what is provided in the Source pane. The implementation will **not** be automatically updated in response to design changes. The implementation must be updated by hand.

5. Click **Finish**.

At any time, you can reset the implementation by opening the Method Implementation wizard and clicking **Return to Default**. If you want to return to using only the default, click **Use the implementation provided by Object Builder** to put the implementation back into read-only mode.

The framework methods `create()`, `retrieve()`, `update()`, `del()`, and `setConnection()` are special framework methods of the data object implementation and persistent object. Unless you provide your own implementation, the special framework method implementations are defined based on the mapping of the data object to the persistent object.

#### **RELATED CONCEPTS**

“Framework methods” on page 757  
“Special framework methods”

#### **RELATED TASKS**

“Working with methods” on page 750  
“Importing edited source files” on page 385  
“Editing special framework methods”

## **Special framework methods**

Data objects and persistent objects that access a schema have the special framework methods `insert`, `update`, `retrieve`, `del`, and `setConnection`. The implementations for these methods are calculated based on the mapping between the persistent object methods and the data object methods.

The order in which the data object methods call their equivalent methods in persistent objects can affect the integrity of the references.

#### **RELATED TASKS**

“Editing special framework methods”  
“Customizing referential integrity” on page 714

## **Editing special framework methods**

Data objects and persistent objects that access a schema have the special framework methods `insert()`, `update()`, `retrieve()`, `del()`, and `setConnection()`. The implementations for these methods are calculated based on the mapping between the persistent object methods and the data object methods.

By default, the method implementations are read-only in the Source pane. To override the calculated method implementations in the Source pane, follow these steps:

1. Locate the method in the Methods pane.
2. From the method’s pop-up menu, click **Properties**.

3. In the Method Implementation wizard, select where you want to get the implementation from: the tool-provided implementation, the Source pane, or an external file.
4. If you select the Source pane as the source, the implementation becomes editable in the Source pane, and whatever changes you make will be preserved.

**Note the following points:**

- If you edited a special framework method, you can switch back to the calculated method body at any time, by changing the setting in the wizard.
- To change the calculated method body for a special framework method, change the mapping of the persistent object method to the data object method on the Methods Mapping page of the Data Object Implementation wizard.
- For most special framework methods, the method bodies that Object Builder provides should be sufficient.
- Most views are read-only. By default, persistent objects are read-only for views, and read-write for tables (schemas), but you can change these settings. For persistent objects that represent read-only views in the database, Object Builder emits the insert(), update() and del() methods with empty method bodies.

**RELATED CONCEPTS**

“Special framework methods” on page 758

**RELATED TASKS**

“Working with methods” on page 750

“Importing edited source files” on page 385

“Customizing referential integrity” on page 714

## Push-down methods

Push-down methods are those that are “pushed down” from the business object to the data object, and finally to the persistent object. Depending on whether they are used along with DB persistent objects, or PA persistent objects, it is either the *relational database application adaptor*, or the *procedural application adaptor* (PAA) that handles the pass-through processing.

In Object Builder, push-down methods are used to expose functionality to the client. In particular, they are used for the following purposes:

- To transmit transactional data of existing applications (when they are used with PA persistent objects)
- To transmit data that is contained in databases that is used by existing applications (when they are used with DB persistent objects, or when they are used as stored procedures).

**Note:** You can edit push-down methods in Object Builder at the data object implementation level.

In Enterprise Access Builder (EAB), a push-down method is one that is written on the procedural adaptor bean. In Object Builder, it is a mapping to the implementation of the method in EAB.

When these methods are executed using the HOD mechanism for accessing IMS applications, the changes resulting from their execution are visible to other sessions.

When these methods are executed using the ECI mechanism for accessing CICS applications, the changes resulting from their execution are not visible to other sessions.

**Note:** When you are creating a data object from a PA persistent object, only push-down methods of the *char* type can be pushed up to the data object.

#### **RELATED CONCEPTS**

“Enterprise Access Builder (EAB)” on page 158

Persistent object (*Programming Guide*)

Data object (*Programming Guide*)

An overview of application adaptors (*Programming Guide*)

#### **RELATED TASKS**

“Using push-down methods with PA persistent objects” on page 762

“Using push-down methods with DB persistent objects” on page 761

“Handling exceptions thrown by PA bean push-down methods” on page 163

## **Working with PA bean push-down methods**

When you add a data object from a PA persistent object, some methods may be automatically defined for you on the data object. These methods correspond to the push-down methods on the PA persistent object (and their PA bean). The implementation of these methods in the data object will cause the corresponding PA bean method to be called. You can add your own methods to the data object, but unlike the push-down methods, they will not be used to call the methods on the PA persistent object.

You can take the following actions on push - down methods:

- Deletion
- Renaming

### **Deletion**

You can delete the push-down methods from the list if you do not want them to be available on the data object. If you accidentally delete one, it is made

available for you to add back to the list by the **Selectoption**. The **Selectoption** is only available in the pop-up menu when there are push-down methods that you have removed from the list.

### **Renaming**

You can also rename the push-down methods, and change the names of their parameters. These changes only affect the method that is defined in the data object; not the one defined in the PA persistent object.

**Restriction:**The following properties of push-down methods are determined by the PA bean, which you imported , and you cannot change them:

- the types of the parameters
- the order of the parameters
- the number of parameters

Push-down methods also show an additional folder which indicates which method on the PA bean the data object method maps to. You cannot change the name of the method on the PA bean.

The following tasks also deal with push-down methods of the PA bean:

- “Using push-down methods with PA persistent objects” on page 762
- “Handling exceptions thrown by PA bean push-down methods” on page 163

#### **RELATED CONCEPTS**

Data object (*Programming Guide*)

Persistent object (*Programming Guide*)

Application adaptor (*Programming Guide*)

## **Using push-down methods with DB persistent objects**

When push-down methods are used with database (DB) persistent objects to transmit data that is contained in databases that are used by existing applications, they can only be used when you map a business object to a data object; they cannot be used when you map a data object to a persistent object.

To employ push-down methods when you map a business object to a DB persistent object, follow these steps:

1. Import the .sql file corresponding to the schema. The DB schema is created in the DBA-Defined Schemas folder.
2. Add a persistent object to the schema. From the pop-up menu of the schema, select **Add Persistent Object**. The Add Persistent Object wizard opens to the Names and Attributes page. Accept the defaults. Ensure that at least one attribute is selected as the persistent object key. The persistent object is created in the folder, beneath the schema.

3. Add a data object to the persistent object. From the pop-up menu of the persistent object, select **Add Data Object**. The Add Data Object wizard opens to the Names and Attributes page. Accept the defaults. Ensure that at least one attribute is selected as the persistent object key. The persistent object is created in the folder, beneath the schema.
4. On the Methods page of the Add Data Object wizard, define the method (along with any parameters) that maps to the push-down method associated with the DB persistent object.
5. Create a business object.
6. At the business object interface level, define the method to correspond with the push-down method..
7. Add an implementation for the business object. Select the option **Add or select one later**.
8. From the pop-up menu of the business object implementation, select **Select a Data Object Interface**. On the Selection page, ensure that you select the data object interface, which was created when you added the data object from the DB persistent object. The data object interface, implementation, and the DB persistent object and DB schema are added to the business object implementation in the User-Defined Business Objects folder.
9. Turn to the Methods Mapping page. The Business Object Methods folder shows the method.
10. Select the method, and from its pop-up menu, select **Add**. The Data Object Method field appears.
11. Click the list button, and select the data object method to be mapped to the business object method.
12. Click **Finish**.

#### **RELATED CONCEPTS**

“Push-down methods” on page 759  
 Persistent object (*Programming Guide*)  
 Data object (*Programming Guide*)  
 Application adaptor (*Programming Guide*)

#### **RELATED TASKS**

“Mapping a business object to a data object” on page 787  
 “Mapping a data object to a DB persistent object” on page 703

## **Using push-down methods with PA persistent objects**

When push-down methods are used to transmit transactional data of existing applications, they have to be used with PA persistent objects. The method of



using them differs, depending on whether you are mapping a business object to a data object, or whether you are mapping a data object to a persistent object.

### Method 1 (Mapping a business object to a data object)

1. Import the PA bean (See Creating a PA schema by importing a PA bean). The PA schema is created in the User-Defined PA Schemas folder along with the PA persistent object.
2. Add a data object to the persistent object.
3. On the Methods page of the Add Data Object wizard, define the method *debit* (of type *long*, with parameter *amount* that maps to the push-down method associated with the PA persistent object, for example, `iBeCashAcctPAOPO.debit`)  
**Note:** The type of the data object method must be the same as the type of the method defined in the PA bean.
4. Create a business object.
5. At the business object interface level, define method *debit* (return type `void`) with a parameter *amount* (of type `long`).  
**Note:** You must define the method signature to match the one you created for your PA bean.
6. Add an implementation for the business object. Select the option **Add or select one later**.
7. From the pop-up menu of the business object implementation, select **Select a Data Object Interface**. On the Selection page, ensure that you select the data object interface, which was created when you added the data object from the PA persistent object (`BeCashAcctPAODO`). The data object interface, implementation, and the PA persistent object and PA schema are added to the business object implementation in the User-Defined Business Objects folder.
8. Turn to the Methods Mapping page. The Business Object Methods folder shows the method (*debit*).
9. Select the method, and from its pop-up menu, select **Add**. The Data Object Method field appears.
10. Click the list button, and select the data object method (*debit*) to be mapped to the business object method.
11. Click **Finish**.

### Method 2 (Mapping a data object to a persistent object)

1. At the business object interface level, define method *debit* (return type `void`) with a parameter *amount* (of type `long`)
2. Add an implementation for the business object.
3. On the Data Object Interface page, specify the attributes of the business object that are to be data object attributes.

4. Turn to the Data Object Methods page, and select the methods of the business object to be pushed down to the data object.
5. Add a data object implementation, selecting **Procedural Adaptors** for the implementation.
6. Associate a PA persistent object with the implementation.
7. Turn to the Methods Mapping page.
8. The method you defined for the data object (*debit*) appears in the User-Defined Methods folder, and you can map it to the corresponding persistent object push-down method.

#### **RELATED CONCEPTS**

“Enterprise Access Builder (EAB)” on page 158

“Push-down methods” on page 759

Persistent object (*Programming Guide*)

Data object (*Programming Guide*)

Application adaptor (*Programming Guide*)

#### **RELATED TASKS**

Creating a PA schema by importing a PA bean

“Working with methods” on page 750

“Mapping a business object to a data object” on page 787

“Mapping a data object to a PA persistent object” on page 708

“Handling exceptions thrown by PA bean push-down methods” on page 163

## **Relationship methods**

When you define a relationship from one business object to another, there is a set of methods created for the relationship. They are the `add()`, `list()`, and `remove()` methods, that enable you to access the relationship.

**Note:** Read-only relationships do not have the `add()` and the `remove()` methods.

For example, if you have a one to many relationship between the Policy business object and the Claim object, you can use the `add()` method to add claims for a policy, the `list()` method to list claims in the policy, and the `remove()` method to remove claims from a policy.

There are different kinds of relationship methods: reference collection methods, foreign key tool-defined methods, and those that you implement yourself by providing your own code. However, you can only customize the implementation of the `list()` method.

#### **RELATED CONCEPTS**

Business object (*Programming Guide*)

#### RELATED TASKS

“Adding a business object implementation and data object interface” on page 780

“Defining relationships” on page 278

“Customizing business object OOSQL implementation methods”

## Customizing business object OOSQL implementation methods

If you define a relationship from one business object to another, and you choose to implement the relationship using OOSQL queries, which you indicate on the Object Relationships page of the Business Object Implementation wizard, you can provide the OOSQL code for the `list()` method of the business object implementation. Object Builder will not validate the code you provide. Your code will overwrite the default tool-generated code for this method.

Follow these steps to customize the implementation of the `list()` method:

1. In the Tasks and Objects pane, select the business object implementation for which you have defined the relationship whose implementation is to use OOSQL queries.
2. The User-Defined Relationships folder in the Methods pane shows the relationship you have defined. Expand the relationship node to view the add, list and remove methods for the relationship that are being implemented for the business object implementation.
3. Select the list method.
4. From its pop-up menu, select **Properties**. The Method Implementation wizard opens to the Implementation page.
5. Accept the defaults and click **Next** to advance to the OOSQL Customization page.
6. Select the **Provide your own OOSQL code** check box.
7. The tool-generated code is cleared from the panel, and it becomes editable. Type in your code for the method and click **Finish**.

You can view the code you provided in the Source pane, when you select the `list()` method in the Methods pane.

#### RELATED CONCEPTS

Business object (*Programming Guide*)

“Relationship methods” on page 764

#### RELATED TASKS

“Working with methods” on page 750

“Adding a business object implementation and data object interface” on page 780

“Defining relationships” on page 278

## Customizing persistent object ESQL framework methods

You can use the ESQL Customization page of the Method Implementation wizard to provide your own embedded SQL code for the special framework methods of any persistent object in the model that uses embedded SQL.

### Note the following points:

- The persistent object’s special framework methods that you can customize are the insert(), update(), retrieve(), and del() methods; not the setConnection() method.
- Object Builder will not validate the code you provide. Your code will overwrite the default tool-generated code for this method.

Follow these steps to customize the ESQL clauses for the special framework methods of the persistent object:

1. Select the persistent object that uses embedded SQL in the Tasks and Objects pane. (You can select it from any one of these folders: User-Defined Business Objects folder, User-Defined Data Objects folder, DBA-Defined Schemas folder.) You will see the persistent object’s methods in the Methods pane.
2. Select the persistent object’s special framework method that you want to customize from the Framework Methods folder.
3. From its pop-up menu, select **Properties**. The Method Implementation wizard opens to the Implementation page.
4. Ensure that the **Use the implementation provided by Object Builder** option is selected.
5. Click **Next**, or click the arrow to the left of the page name, and select ESQL Customization page from the list. The page opens.
6. Select the check box **Provide your own ESQL code**. This makes the panel that shows the code for the method editable, and you can either edit the code provided by Object Builder, or type in entirely different code for the method. Object Builder will not validate the code you provide.
7. Click **Finish**.

**Note:** You can overwrite the code you provide with Object Builder’s code for the method by clearing the **Provide your own ESQL code** check box.

### RELATED CONCEPTS

Persistent object (*Programming Guide*)

“Special framework methods” on page 758

**RELATED TASKS**

“Working with methods” on page 750

“Working with DB persistent objects” on page 832

## Deleting a method

To delete a user-defined method, follow these steps:

1. In the User-Defined Business Objects folder, locate the object (business object interface, data object interface, business object implementation, or data object implementation) that defines the method.
2. From the pop-up menu of the interface, click **Properties** to open the object’s wizard.
3. Click the page title and turn to the Methods page.
4. Locate the method under the Methods folder.
5. From the pop-up menu of the method, click **Delete**.
6. Click **Finish**.

If the method was defined in an interface, then it is automatically deleted from any associated implementations.

Get and set methods are deleted automatically when the attribute they represent is deleted.

**RELATED CONCEPTS**

“User-defined methods” on page 751

**RELATED TASKS**

“Working with methods” on page 750

---

## Method mapping properties

The following topics describe the details of specific method mappings:

- “Special Framework Methods” on page 768
- “User-Defined Methods” on page 768
- “Method Reordering” on page 769

**RELATED CONCEPTS**

“Object Builder” on page 1

**RELATED TASKS**

“Working with methods” on page 750

## Special Framework Methods

The **Special Framework Methods** folder contains the special framework methods - `insert()`, `update()`, `retrieve()`, `del()`, and `setConnection()` - that are implemented for the data object. You can specify the processing order of the framework methods with the method reordering controls (see the reference section: “Method Reordering” on page 769), and can thus customize the implementation and referential integrity. For each of the special framework methods, the order of selection of the persistent object methods is the calling sequence and determines the order of processing. To specify the processing order, select the special framework method for which you want to define the implementation and select **Add Mapping** from its pop-up menu.

### RELATED CONCEPTS

Persistent object (*Programming Guide*)

“Framework methods” on page 757

“Special framework methods” on page 758

### RELATED TASKS

“Customizing referential integrity” on page 714

“Working with methods” on page 750

### RELATED REFERENCES

“Method Reordering” on page 769

## User-Defined Methods

When you are working with the Procedural Application Adaptor, the methods you defined for the data object using the Methods page of the Data Object Interface wizard appear in the User-Defined Methods folder, and you can map them to the corresponding persistent object push-down method.

### RELATED CONCEPTS

Data object (*Programming Guide*)

Persistent object (*Programming Guide*)

“User-defined methods” on page 751

“Push-down methods” on page 759

### RELATED TASKS

“Working with methods” on page 750

“Using push-down methods with PA persistent objects” on page 762

“Customizing referential integrity” on page 714

“Adding a business object implementation and data object interface” on page 780

“Creating a data object interface” on page 801

“Mapping a data object to a DB persistent object” on page 703

“Mapping a data object to a PA persistent object” on page 708

## Method Reordering

When a special framework method calls more than one persistent object method, the order in which they are called determines the order of processing. You can use the following buttons to reorder the methods:

- **Move Up**

This button is available for selection when you have selected any of the persistent object methods except the first one, from the Special Framework Methods folder. Each time you use this button, the selected method shifts one level higher in the tree.

- **Move Down**

This button is available for selection when you have selected any of the persistent object methods except the last one, from the Special Framework Methods folder. Each time you use this button, the selected method shifts one level lower in the tree.

### RELATED CONCEPTS

Persistent object (*Programming Guide*)

“Framework methods” on page 757

“Special framework methods” on page 758

### RELATED TASKS

“Customizing referential integrity” on page 714

“Working with methods” on page 750

---

## Working with constructs

Constructs (constants, enumerations, exceptions, typedefs, structures, and unions) can be defined at the file, module, or interface level of a business object interface or data object interface, or at the file or module level of a composition.

You can define them directly in Object Builder, or define them in Rose and export them to Object Builder.

The following tasks deal with constructs:

- “Defining constructs with file scope” on page 770
- “Defining constructs with module scope” on page 771
- “Defining constructs with interface scope” on page 772
- “Editing a construct” on page 773
- “Deleting a construct” on page 774

#### RELATED CONCEPTS

“Constructs”

#### RELATED TASKS

“Working with methods” on page 750

“Working with attributes” on page 697

#### RELATED REFERENCES

“Constructs in Rose” on page 101

## Constructs

In Object Builder, the following items are referred to throughout as constructs:

- Constant
- Enumeration
- Exception
- Typedef
- Structure
- Union

They can be defined at the file, module, or interface level of a business object interface, data object interface, or local-only object, or at the file or module level of a composition.

#### RELATED TASKS

“Working with constructs” on page 769

## Defining constructs with file scope

You can define constructs with file scope using the Business Object File wizard, Data Object File wizard, or Composition File wizard. You can add the constructs when you create a new file, or by modifying the properties of an existing file (from the file’s pop-up menu, select **Properties**).

**Warning:** File scope constructs are not qualified. In C++ implementations, they pollute the namespace and may conflict with other definitions. In Java implementations they are placed in the “unnamed” package, and may produce compilation errors (processing of classes in the unnamed package is implementation dependent). It is recommended that all constructs be defined at module or interface scope.

To define a construct, follow these steps:

1. In the wizard, click the page title and turn to the Constructs page.



2. From the pop-up menu of the Constructs folder, select the construct you want to add. You can select from the following options:
  - Constant
  - Enumeration
  - Exception
  - Typedef
  - Structure
  - Union

If the construct is an enumeration, exception, structure, or union, you must define at least one member for it. To define a member for a construct, follow these steps:

- a. Under the construct in the tree view, select the Members folder.
- b. From the Members pop-up menu, click **Add**.
- c. Type a name for the member and any other requested information. The information is saved when you click another item, select another action, or leave the page.

**Note:** To use the construct as a type within another construct, you must first click **Finish** and then re-open the wizard before you can use the type. The construct is not added to the current model until you click **Finish**.

3. Complete the rest of the pages, or click **Finish**.

#### **RELATED TASKS**

“Working with constructs” on page 769

“Creating a business object file” on page 775

“Creating a data object file” on page 797

“Creating a composition file” on page 885

## **Defining constructs with module scope**

You can define constructs with module scope using the Business Object Module wizard, Data Object Module wizard, or Composition Module wizard. You can add the constructs when you create a new module, or by modifying the properties of an existing module (from the module’s pop-up menu, select **Properties**).

To define a construct, follow these steps:

1. In the wizard, click the page title and turn to the Constructs page.
2. From the Constructs pop-up menu, select the construct you want to add. You can select from the following options:
  - Constant
  - Enumeration

- Exception
- Typedef
- Structure
- Union

If the construct is an enumeration, exception, structure, or union, you must define at least one member for it. To define a member for a construct, follow these steps:

- Under the construct in the tree view, select the Members folder.
- From the Members pop-up menu, select **Add Member**.
- Type a name for the member and any other requested information. The information is saved when you click another item, select another action, or leave the page.

**Note:** To use the construct as a type within another construct, you must first click **Finish** and then re-open the wizard before you can use the type. The construct is not added to the current model until you click **Finish**.

- Complete the rest of the pages, or click **Finish**.

#### **RELATED TASKS**

“Working with constructs” on page 769

“Adding a business object module” on page 777

“Adding a data object module” on page 800

“Adding a composition module” on page 886

## **Defining constructs with interface scope**

You can define constructs with interface scope using the Business Object Interface wizard or Data Object Interface wizard. You can add the constructs when you create a new interface, or by modifying the properties of an existing interface (from the interface’s pop-up menu, select **Properties**).

To define a construct, follow these steps:

- In the wizard, click the page title and turn to the Constructs page.
- From the Constructs pop-up menu, select the construct you want to add. You can select from the following options:
  - Constant
  - Enumeration
  - Exception
  - Typedef
  - Structure
  - Union

If the construct is an enumeration, exception, structure, or union, you must define at least one member for it. To define a member for a construct, follow these steps:

- a. Under the construct in the tree view, select the **Members** folder.
  - b. From the **Members** pop-up menu, select **Add Member**.
  - c. Type a name for the member and any other requested information. The information is saved when you click another item, select another action, or leave the page.
3. Complete the rest of the pages, or click **Finish**.

**Note:** To use the construct as the type of an attribute, method return, method exception, or construct member, you must first click **Finish** and then re-open the wizard before you can use the type. The construct is not added to the current model until you click **Finish**.

#### **RELATED TASKS**

*"Working with constructs" on page 769*

*"Adding a business object interface" on page 777*

*"Creating a data object interface" on page 801*

## **Editing a construct**

To edit a construct, follow these steps:

1. Locate the file, module, or interface that defines the construct in the User-Defined Business Objects folder.
2. From the pop-up menu of the item, click **Properties**. The item's wizard opens.
3. Click the page title and turn to the **Constructs** page.
4. Under the **Constructs** folder, locate the construct, and select it.
5. Edit the construct. If the construct has a **Members** folder, you can add, delete, or edit the members.
6. Click **Finish**.

The construct is changed. Any methods that used the construct as their method return type or as a parameter, and any attributes that had the construct as their type, are automatically updated.

#### **RELATED CONCEPTS**

*"Constructs" on page 770*

#### **RELATED TASKS**

*"Working with constructs" on page 769*

## Deleting a construct

To delete a construct, follow these steps:

1. Locate the file, module, or interface that defines the construct in the User-Defined Business Objects folder.
2. From the pop-up menu of the item, click **Properties**. The item's wizard opens.
3. Click the page title and turn to the Constructs page.
4. Under the Constructs folder, locate the construct.
5. From the pop-up menu of the construct, click **Delete**.
6. Click **Finish**.

The construct is deleted. Any methods that used the construct as their method return type or as a parameter, and any attributes that had the construct as their type, are automatically modified to refer to `invalidType`.

### RELATED CONCEPTS

"Constructs" on page 770

### RELATED TASKS

"Working with constructs" on page 769

---

## Working with business objects

Business objects are defined in the User-Defined Business Objects folder, and are presented in terms of four objects:

- The business object file (which contains one or more interfaces, optionally organized into modules)
- The business object module, if any (which contains one or more interfaces)
- The business object interface (which has one or more implementations)
- The business object implementation (which has its own file, defined on the first page of its wizard)

The four objects are created and edited separately, but collectively form a single business object. Each business object (each set of business object file, module, interface, and implementation) typically has its own data object.

The following tasks deal with business objects:

- "Creating a business object file" on page 775
- "Adding a business object module" on page 777
- "Adding a business object interface" on page 777
- "Creating a business object by importing an IDL file" on page 780

- “Adding a business object implementation and data object interface” on page 780
- “Adding a business object from a data object” on page 784
- “Mapping a business object to a data object” on page 787
- “Editing a business object interface” on page 791
- “Editing a business object implementation” on page 792
- “Editing a business object implementation file” on page 793
- “Customizing business object OOSQL implementation methods” on page 765
- “Deleting a business object interface” on page 794
- “Deleting a business object implementation” on page 794

#### RELATED CONCEPTS

Business object (*Programming Guide*)

Data object (*Programming Guide*)

#### RELATED REFERENCES


“Naming objects” on page 128

“Internationalization of data” on page 132

## Creating a business object file

A business object file (IDL) is a container for your business object interfaces. Although a file can hold multiple business object interfaces, which you may organize into modules, you typically add one interface to each file.

To create a business object file, follow these steps:

1. From the Tasks and Objects pane, select the **User-Defined Business Objects** folder.
2. From the folder’s pop-up menu, select **Add File**. The Business Object File wizard opens to the Name page.
3. Type a name for the file (for example, an insurance application might have a file named Policy).
4.  Accept the default name for the IR file, or type one. This is the executable that the DDL will run when System Management loads. It populates the Interface Repository with interfaces that are required for the client.

Object Builder generates the default name using the business object’s filename, with the suffix `_IR`. Object Builder does not add an extension for this file. Whether you accept the default name, or create a new name for the file, you must make sure that its primary name does not exceed eight characters.

**Note:** For each business object interface, `idlC - eir` is run against the business object IDL file to generate a C++ source file. This source file is then compiled and linked, resulting in the IR executable.

5. In the Deployment Platforms section, select the platforms on which the business object is to be deployed.
6. Click **Next**. The Constructs page opens.  
Use the Constructs pop-up menu to add constants, enumerations, exceptions, structures, typedefs, and unions. Any constructs you add are scoped to every interface in the file.  
**Note:** To use the construct as a type within another construct, you must first click **Finish** and then re-open the wizard before you can use the type. The construct is not added to the current model until you click **Finish**.
7. Click **Next**. The Files to Include page opens.  
`IManagedClient` is included by default. This is the correct choice for a component that represents a base class in your design. If your component has a parent, you would specify the business object file of the parent component in this field. For example, if the `CarPolicy` component inherits from the `Policy` component, then you would specify the business object file for `Policy` on this page. Also include the business object files for any referenced or related components. For example, if `CarPolicy` has an attribute of type `Claim`, you would need to include the business object file for `Claim` on this page.
8. Click **Next**. The Comments page opens. Type any comments you want to include as comment lines in your generated IDL code.
9. Click **Finish**. The wizard closes, and your file is added to the User-Defined Business Objects folder. You can now add modules or interfaces to the file.

Once you have created the file, you can modify it by selecting **Properties** from its pop-up menu. The Business Object File wizard opens again, with your selections preserved.

#### **RELATED CONCEPTS**

Business object (*Programming Guide*)

#### **RELATED TASKS**

“Working with business objects” on page 774

“Defining constructs with file scope” on page 770

“Adding a business object module” on page 777

“Adding a business object interface” on page 777

#### **RELATED REFERENCES**

“Internationalization of data” on page 132

“Naming objects” on page 128

## Adding a business object module

You can group your interfaces into modules to provide them with a unique namespace. Generally, all your interfaces should be in modules to reduce the chance of naming conflicts once they are deployed on a network. Any constructs you add to a module are scoped only to the interfaces within that module. To add a module to a file, follow these steps:

1. From the User-Defined Business Objects folder, select your business object file.
2. From the file's pop-up menu, select **Add Module**. The Business Object Module wizard opens to the Name page.
3. Type a name for the module.
4. Click **Next**. The Constructs page opens.

Use the Constructs pop-up menu to add enumerations, exceptions, structures and so on.

**Note:** To use the construct as a type within another construct, you must first click **Finish** and then re-open the wizard before you can use the type. The construct is not added to the current model until you click **Finish**.

5. Click **Next**. The Comments page opens. Type any comments you want to include as comment lines in your generated code.
6. Click **Finish**. The wizard closes, and your module is added to the User-Defined Business Objects folder, underneath the file.

You can now add business object interfaces to the module.

### RELATED CONCEPTS

Business object (*Programming Guide*)

### RELATED TASKS

"Working with business objects" on page 774

"Defining constructs with module scope" on page 771

"Adding a business object interface"

### RELATED REFERENCES

"Internationalization of data" on page 132

"Naming objects" on page 128

## Adding a business object interface



You can add multiple business object interfaces to the same file or module. However, the more interfaces a file contains, the slower the compilation process will be.

**Recommendation:** To improve performance, limit the number of interfaces per file to less than ten.

To add a business object interface to a file (or module), follow these steps:

1. From the User-Defined Business Objects folder, select the file or module that will contain the interface.
2. From the pop-up menu for the file or module, select **Add Interface**. The Business Object Interface wizard opens to the Name page.
3. Type a name for the interface (for example, CarPolicy).
4. Select whether the interface will be queryable or not.

If you select this option, the generated code for the managed object contains the dynamic dispatch method call `IMethodByName`, which allows the Query Service to call the methods of the managed object. You should also configure the managed object with a queryable home.

5. Click **Next**. The Constructs page opens.

Use the pop-up menu of the Constructs folder to add constants, enumerations, exceptions, typedefs, structures, or unions. Any constructs you add are scoped to this interface only.

**Note:** To use the construct as the type of an attribute, method return, method exception, or construct member, you must first click **Finish** and then reopen the wizard before you can use the type. The construct is not added to the current model until you click **Finish**.


6. Click **Next**. The Interface Inheritance page opens.

By default, the interface inherits from `IManagedClient::IManageable`. This is the correct choice for a component that represents a base class in your design. If your component had a parent, you would specify the business object interface of the parent component on this page.

For applications that use MQSeries application adaptors to send or receive messages to or from a queue, you need to change the inheritance to one of the following:

- `IMessage IMessage::InboundMessage` - for an interface that receives inbound messages from queues that are managed by an MQSeries application adaptor; or
- `IMessage IMessage::OutboundMessage` - for an interface that writes message to outbound queues that are managed by an MQSeries application adaptor.

Do **not** select any of the home inheritance options. These are only appropriate if you are creating a specialized home.

 If the business object is deployable to OS/390, the `IPolymorphicHome` class is not available for selection as a parent interface.

7. Click **Next**. The Attributes page opens.

To specify attributes for your interface, select **Add** from the Attributes pop-up menu (for example, the CarPolicy interface could have the



attributes “make” and “model”).

You can view how all the different data types, even those from other projects, are used within the model. (Use the **Browse** button to open the Type Browser.)

**Note:** For most attribute types, a default initializer value is provided. When there is no suitable default (for example, an attribute whose type is an enumeration), you should assign your own initializer value, if necessary.

All attributes you specify will be exposed as part of the public interface of the component. You can specify protected-access attributes in the business object implementation.

8. Click **Next**. The Methods page opens.

To specify methods for your interface, select **Add** from the Methods pop-up menu. For example, the CarPolicy interface could have the method “riskQuotient”.

9. Click **Next**. The Object Relationships page opens.

To specify any relationships that this class has to other classes, select **Add** from the Objects pop-up menu. The relationships will be 1-n. You can specify the implementation details for the relationship when you add a business object implementation to this interface.

10. Click **Next**. The Comments page opens. Type any comments you want to include as comment lines in your generated code.
11. Click **Finish**. Your new interface is added to the User-Defined Business Objects folder, with the attributes and methods you specified.

You should now see your interface in the Inheritance pane, and any methods you defined for your interface should appear under the User-Defined Methods folder in the Methods pane.

#### **RELATED CONCEPTS**

Business object (*Programming Guide*)

Query service (*Advanced Programming Guide*)

#### **RELATED TASKS**

“Working with business objects” on page 774

“Tracking data types in models” on page 235

“Defining constructs with interface scope” on page 772

“Creating a specialized home” on page 876

“Adding a key” on page 826

#### **RELATED REFERENCES**

“Internationalization of data” on page 132

“Naming objects” on page 128

## Creating a business object by importing an IDL file

If you have code already in IDL files, you can parse the code into Object Builder, and incorporate the classes, relationships, and code in the IDL files into your Object Builder application. A relationship is imported as methods.

**Note:** The IDL must be CORBA 2.2-compliant without IDL extensions. You can make sure the IDL files you are importing are valid by compiling them first. Object Builder will only import IDL files that are considered valid by the compiler.

To import an existing IDL file, follow these steps:

1. Under Tasks and Objects, select the User-Defined Business Objects folder.
2. From the folder's pop-up menu, select **Import IDL**. The Import IDL wizard opens to the File Selection page.
3. Browse for and select the files you want to import. The files you select, and any files they include, will be parsed and imported into Object Builder.
4. Click **Next**. The Search Paths for Nested Files page opens.
5. From the Include directories pop-up menu, select **Add**. Browse for the directories you want searched.

When you import a file that includes other files (that is, a file with nested files), the import process will search for the other files in the directories you specify here.

6. Click **Finish**. The selected files (and files they include) are parsed into Object Builder, and the information in the IDL is added to the current project model as business object files, business object modules, and business object interfaces.

### RELATED CONCEPTS

Interface Definition Language (*Programming Guide*)

### RELATED TASKS

"Importing IDL from the command line" on page 666

"Creating a data object by importing an IDL file" on page 804

"Creating a local-only object by importing an IDL file" on page 669

## Adding a business object implementation and data object interface



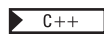
Once you have created a business object interface, you can add one or more implementations for that business object, and also create the data object interface that provides your business object with access to data. You can accomplish both tasks using the Business Object Implementation wizard. Ensure that you have added a key and a copy helper to the business object interface before proceeding with this task.

To create the business object implementation, and its associated data object interface, follow these steps:


1. From the User-Defined Business Objects folder, select the business object interface you want to implement.
2. From the interface's pop-up menu, click **Add Implementation**. The Business Object Implementation wizard opens to the Name and Data Access Pattern page.
3. Appropriate implementation names are filled in for you (the business object file name and interface name plus BO: for example, CPFile::CarPolicy gets an implementation named CPFileBO::CarPolicyBO). You can accept these defaults or replace them with your own names.
4. Set the types of behavior you want your business object to have. You can set them in the following sections:
  - "Pattern for Handling State Data" on page 245
  - "Object Reference" on page 246
  - "Data Object Interface" on page 247
  - "Session Service" on page 248
  - "Deployment platforms" on page 423

**Note:** If you create a new implementation file when you create the business object implementation, Object Builder associates the deployment platforms that you specify with that file. If you create a new business object implementation, and add it to an existing file (that is, you specify the name of a business object implementation file that already exists in the model in the **File Name** field, Object Builder checks whether the deployment platforms that you specified for the implementation are the same as those that are selected for the existing implementation file. If they are not the same, you must follow one of these procedures:

- Change the existing implementation file's platforms to be the same as those for the implementation you are adding, before adding the new implementation: From the pop-up menu of the existing business object implementation, select **File Properties**, and change the deployment platforms on the Name page of the Business Object Implementation File wizard.
  - Change the implementation's platforms to match those of the existing file, on this page (the Name and Data Access Pattern page).
5. Click **Next**. If C++ is your default implementation language, then the Files to Include page opens.




If you are developing a C++ business object, you can specify files to include here.

 If you are developing a Java business object, ignore this page. It will be removed from the wizard once you select **Java** as the implementation language (on the Implementation Language page).

6. Click **Next**. The Implementation Inheritance page opens.
7. Make sure that `IManagedClient::IManageable` is listed as a parent under the Parent Class folder.


You can also select any parent business object implementations you want to inherit behavior from.


 If the business object is deployable to OS/390, the `ISpecializedPolymorphicHome` class is not available for selection as a parent interface.

8. Click **Next**. The Implementation Language page opens. Select the language you want the business object to be implemented in. You can select either Java or C++.

The default for this page is set in the Preferences notebook, on the Tasks and Objects page.

9. Click **Next**. The Attributes page opens. Specify any attributes you want to add to the business object implementation (in addition to the attributes you already specified in the business object interface).

 For C++ business objects, if you want to specify attributes of type `Object`, you must use the type alone, and not the corresponding `_ptr` type. This is because Object Builder automatically does the `_ptr` tagging for C++ business objects.

 For Java business objects, you can specify the `_ptr` type for attributes of type `Object`.

10. Click **Next**. The Methods page opens. Specify any methods you want to add to the business object implementation (in addition to the methods you already specified in the business object interface).
11. Click **Next**. The Key and Copy Helper page opens. Select a key and, optionally, copy helper that you have created for this business object (for example, `CarPolicyKey` and `CarPolicyCopy`).

**Note:** Any primary key or copy helper in the model that contain attributes from another business object implementation will not be available for selection.

12. Click **Next**. The Handle Selection page opens.

You can select a handle for the business object implementation. If you select a handle, then the framework method `getHandleString` is implemented, which overrides the `getHandleString` method of `IManagedClient::IManageable`. The method provides a way to encapsulate the business object implementation, by returning a string that represents a reference to the business object. The handle you select

determines the pattern that is used to form the string (that is, to turn the reference into a string, or to swizzle the pointer).

13. Click **Next**. If the business object implementation has parent classes with overrideable attributes, then the Attributes to Override page opens.  
You can use this page to select which of the parent class's attributes you want to override.
14. Click **Next**. If the business object implementation has parent classes with overrideable methods, then the Methods to Override page opens.  
You can use this page to select which of the parent class's methods you want to override.
15. Click **Next**. If the business object implementation has parent classes with overrideable relationships, then the Relationships to Override page opens.  
You can use this page to select which of the parent class's relationships you want to override.
16. Click **Next**. If the business object interface defines 1-n relationships, then the Object Relationships page opens.  
You can use this page to set the way that the object relationship will be implemented.
17. Click **Next**. The Data Object Interface page opens.  
**Note:**This page does not open if, on the first page, you chose not to create a new data object.  
Appropriate data object names are filled in for you (the business object file name and interface name plus DO: for example, CPFile::CarPolicy gets the data object interface CPFileDO::CarPolicyDO). You can accept these defaults or replace them with your own names.
18. Select the attributes that you want preserved in the data object. These attributes constitute the state data for the component.  
If you implemented a one-to-many relationship as a **Local persistent reference**, then an attribute representing it appears here, so you can select to preserve it in the data object.
19. Click **Next**. The Data Object Methods page opens.  
**Note:**This page does not open if, on the first page, you chose not to create a new data object.
20. Select which business object methods you want to push down to the data object (that is, call equivalent methods to be defined in the data object).
21. Click **Next**. The Summary of Framework Methods page opens.  
Based on your selections on the previous pages of the wizard, this page displays the methods that your object implements. For example, if you selected a caching pattern to handle the essential state of your business object (on the first page), this list includes the synchToDataObject method required to keep the two sets of attributes synchronized. No action is needed.

22. Click **Finish**. The business object implementation and data object interface appear in the User-Defined Business Objects folder, under your business object interface. The data object interface also appears in the User-Defined Data Objects folder.

Now that the business object implementation is defined, you can specify the implementation code for each user-defined method.

#### RELATED CONCEPTS

Business object (*Programming Guide*)

Data object (*Programming Guide*)

Session Service (*Advanced Programming Guide*)

#### RELATED TASKS

“Implementing methods” on page 752

“Adding resource methods to a sessional business object” on page 164

“Adding a data object implementation” on page 807

“Defining a 1-n relationship” on page 281

#### RELATED REFERENCES

“Internationalization of data” on page 132

“Naming objects” on page 128

## Adding a business object from a data object

**Note:** You can add a business object to a data object only if the data object has no associated business object.

To add a business object directly from a data object, follow these steps:

1. Select the data object interface in the User-Defined Data Objects folder.
2. From the pop-up menu of the object, select **Add Business Object**. The Add Business Object wizard opens to the Name page.  
Type the names for the business object file, interface, and IR file, and indicate whether the interface is to be queryable. Then select the platforms on which the object is to be deployed. The defaults are set by the platform selections you make from the **Platform > Constrain** menu.
3. Click **Next**. The Interface Inheritance page opens. By default, the interface inherits from `IManagedClient::IManageable`. This is the correct choice for a component that represents a base class in your design. If your component has a parent, you would specify the business object interface of the parent component on this page.
4. Click **Next**. The Name and Data Access Pattern page opens. Appropriate implementation names are filled in for you (the business object file name and interface name plus BO: for example, `CPFile::CarPolicy` gets an

implementation named `CPFileBO::CarPolicyBO`). You can accept these defaults or replace them with your own names.

Set the types of behavior you want your business object to have. You can set the following properties:

- “Pattern for Handling State Data” on page 245
- “Object Reference” on page 246
- “Session Service” on page 248

5. Click **Next**. The Implementation Language page opens. You can select either C++ or Java as the implementation language for the business object. The objects’ classes will be implemented in the language you select.

The default for this page is set in the Preferences notebook, on the Tasks and Objects page.

6. Click **Next**. The Handle Selection page opens.

You can select a handle for the business object implementation. If you select a handle, then the framework method `getHandleString` is implemented, which overrides the `getHandleString` method of `IManagedClient::IManageable`. The method provides a way to encapsulate the business object implementation, by returning a string that represents a reference to the business object. The handle you select determines the pattern used to form the string (that is, to turn the reference into a string, or to swizzle the pointer).

7. Click **Next**. The Attributes page opens. Here, you can indicate whether you want to push a selected attribute of the data object interface up to the business object interface, or the business object implementation (using the **Define Attribute in** section), whether the selected attribute that will be an attribute of the key class, or of the copy helper class, or of both (using the **Add Attribute to** section), and you can also add new attributes for either the business object interface, or the business object implementation (again, indicating the level at which the attribute is to be defined using the **Define in** section). If you select **data object interface** in the **Define Attribute in** section, the attribute is not added to either the business object interface, or its implementation.

If the highlighted attribute is a string, the **String Behavior** section appears. You can indicate whether the string is to be a CORBA string, if trailing blank spaces in the string are to be stripped off, or if the string is to be padded with spaces.

Among the other properties that you can specify are whether the attribute is read-only, overrideable by a subclass, static, and if the get method for the attribute is made a constant (`const`), which indicates that it will not change the state of the business object.

8. Click **Next**. The Methods page opens. Here, you can indicate whether you want to push a selected method up to the business object interface, or the business object implementation (using the **Define Method in** section), and

you can also add new methods for either the business object interface, or the business object implementation (again, indicating the level at which the method is to be defined using the **Define Method in** section). If you select **data object interface** in the **Define Method in** section, the method is not added to either the business object interface, or its implementation.

#### 9. Click **Finish**.

The business object appears in the User-Defined Business Objects folder, complete with its file, interface, implementation, and its key and copy helper. (A key and copy helper appear only if you marked some of the attributes on the Attributes page as “key” or “copy helper” in the **Add attribute to** section.)

The following objects inherit from default classes:

- the business object implementation (inherits from `IManagedClient::IManageable`)
- the key (inherits from `IManagedLocal::IPrimaryKey`)
- the copy helper (inherits from `IManagedLocal::INonManageable`)

You can edit the properties of either the business object interface, or the business object implementation (**Properties** from the pop-up menu of the object) to change the inheritance, as well as perform the following tasks:

- Override attributes and methods
- Indicate the way in which relationships with other objects are to be implemented by modifying the properties of the business object implementation.
- Add constructs for the business object

**Note:** You cannot add constructs while you are creating this business object interface. Once the interface is created, you can add constructs at the different scoping levels (file, module, and interface) when you modify the business object’s properties (select **Properties** from the pop-up menu of its file, module, or interface).

You can now add a managed object for the business object before you can deploy it.

**Note:** An indirect way of adding a business object to a data object is to create the business object separately, and then associate it with either a new or existing data object. (Select the option **Add or select one later** from the Name and Data Access Pattern page of the Business Object Implementation wizard.)

#### **RELATED CONCEPTS**

Data object (*Programming Guide*)

Business object (*Programming Guide*)



#### RELATED TASKS

- “Creating a component for existing DB data” on page 139
- “Working with business objects” on page 774
- “Working with data objects” on page 795
- “Creating a business object file” on page 775
- “Adding a business object module” on page 777
- “Adding a business object interface” on page 777
- “Mapping a business object to a data object”

#### RELATED REFERENCES

- “Internationalization of data” on page 132
- “Naming objects” on page 128

## Mapping a business object to a data object

Once you have added a business object and its implementation, and have specified that it uses the meet-in-the-middle approach, you can map it to an existing data object. You can map both attributes and methods of one object to the other. To do so, follow these steps:

1. From the business object implementation’s pop-up menu, click **Select Data Object Interface**. The Data Object Interface Connection wizard opens to the Selection page.
2. In the **Data Object Interface Name** field, type the name of an existing data object interface in the form *data\_object\_name data\_object\_interface\_name*, or select one of the interface names from the list.

**Note:** If you select a data object, which was created using the bottom-up approach (that is, one created from a persistent object), it is recommended that you initialize the attributes of the data object before you map the attributes of the business object to those of the data object. (They are not initialized by default.) So, follow these steps:

- a. Click **Finish**. This adds the selected data object interface to the User-Defined Business Objects folder, beneath the business object implementation.
- b. Initialize the attributes of the data object. Follow these steps:
  - 1) Select the data object interface in the User-Defined Data Objects folder.
  - 2) From the pop-up menu of the data object interface, select **Properties**. The Data Object Interface wizard opens.
  - 3) Click **Next**, or click the arrow to the left of the page name, and select Attributes page from the list. The page opens.
  - 4) Select the data object attributes from the Attributes folder and type the initial value for the attribute in the **Initializer** field.

- c. Continue with the mapping: from the pop-up menu of the data object interface (which was selected for the business object), select **Properties**. The Data Object Interface Connection wizard opens.
  - d. Click **Next**, or click the arrow to the left of the page name, and select Attributes Mapping page from the list. The page opens.
  - e. Follow step 4.
3. Click **Next**. The Attributes Mapping page opens. On this page, you can map the business object attributes to the data object attributes available from the business object interface. A business object attribute can map to a single data object attribute of the same data type: the mapping is one-to-one.
- Note:** Object Builder does the default mapping between business object attributes and data object attributes if the following properties hold true:
- The attributes are of the same name
  - The attributes are of the same type

Of course, you can modify these mappings, if you want to.

To map business object attributes to data object attributes (those that are not mapped), follow these steps:

- a. From the pop-up menu of the Business Object Attributes folder, select **Add**.
  - b. Type the name of an attribute of the data object interface you specified earlier, or select one from the **Data Object Attribute** field's list.
4. Click **Next**. The Methods Mapping page opens. On this page, you can map the business object methods to the data object methods. You can only have a one-to-one mapping between any business object methods and data object methods, and the signatures of the two methods must be the same.
- Note:** Object Builder does the default mapping between business object methods and data object methods if the following properties hold true:
- The method names are the same
  - The method return types are the same
  - The methods have the same number of parameters, and they are of the same type. (The names of the parameters are irrelevant.)

Of course, you can modify these mappings, if you want to.

Object Builder always provides the default mapping for the insert(), update(), del(), retrieve(), and setConnection() methods.

To map business object methods to data object methods (those that are not mapped), follow these steps:

- a. From the pop-up menu of the Business Object Methods folder, select **Add**.
  - b. Select a method of the data object from the **Data Object Methods** field's list. This list contains only those methods of the data object that you defined for its interface.
5. Click **Next**, and add any comments about the mapping on the Comments page.

The current mapping takes effect when you proceed to map the next business object attribute, or when you click **Finish**. The selected data object interface appears in the User-Defined Business Objects folder, under the business object implementation node.

**Note:** After you have mapped a business object to a data object, you can view and edit the mapping of the attributes on the Attributes Mapping page and the Methods Mapping page of the Business Object Implementation wizard. This wizard will no longer have the Data Object Interface page.

#### RELATED CONCEPTS

Business object (*Programming Guide*)

Data object (*Programming Guide*)

#### RELATED TASKS

"Working with business objects" on page 774

"Working with data objects" on page 795

#### RELATED REFERENCES

"DB2 data type mappings" on page 142

"Oracle data type mappings" on page 146

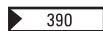
## Editing a business object file

You can edit a business object file using either of the following methods:

- Using **Properties** from the pop-up menu of the business object file
- Using **File Properties** from the pop-up menu of either the business object module, or the business object interface

To edit a business object file using the file itself, follow these steps:

1. Select the business object file in the User-Defined Business Objects folder.
2. From its pop-up menu, select **Properties**. The Business Object File wizard opens to the Name page. You can rename the file, if you want to.



You can also specify another IR file name (the executable file that the DDL will run when System Management loads).

3. Click **Next**. The Constructs page opens. Add new constructs, or modify, or delete existing ones. You can add constants, enumerations, exceptions, structures, typedefs, and unions. Any constructs you add are scoped to every interface in the file.

**Note:** To use the construct as a type within another construct, you must first click **Finish** and then reopen the wizard before you can use the type. The construct is not added to the current model until you click **Finish**.

Click **Next**. The Files to Include page opens. Add new header files if you want to, or delete existing ones.

**Note:** If you add new files to be included in your business object file, you must include these files as file adornment prefixes for keys and data object implementations that are already associated with this business object, and for those that you will define later. The key and data object implementation IDL files will not automatically include the headers that you specify for the business object file. Include them by selecting **IDL** as the file location on the Adornment Details page. See the task Adding file adornments.

4. Click **Next**. The Contents Ordering page opens. Use this page to view or set the order of constructs and interfaces in the IDL file.
5. Click **Next**. The Comments page opens. Type any comments you want to include as comment lines in your generated IDL code.
6. Click **Finish**. The wizard closes. Any modules or interfaces that you add for this object, will have the modifications that you made propagated to them.

To edit the business object file using either a business object module, or a business object interface, follow these steps:

1. From the pop-up menu of either the business object module, or the business object file, select **File Properties**. The Business Object File wizard opens to the Name page.
2. You can type a new name for the business object file, and select a different set of deployment platforms for the object.  
**Note:** The remaining pages are the same as when you edit the business object file using the file itself: the Constructs page, the Files to Include page, the Contents Ordering page, and the Comments page. Change these properties, if you want to.
3. Click **Finish** to apply your changes.

#### **RELATED CONCEPTS**

Business object (*Programming Guide*)

#### **RELATED TASKS**

“Working with business objects” on page 774

“Defining constructs with file scope” on page 770

“Adding a business object module” on page 777  
“Adding a business object interface” on page 777  
“Adding file adornments” on page 240

#### RELATED REFERENCES

“Internationalization of data” on page 132  
“Naming objects” on page 128

## Editing a business object interface

Business object interfaces are defined in the User-Defined Business Objects folder, where they are shown under the file (and module, if any) in which they are defined. You can edit the file, module, and business object interface as separate objects, following these steps:

1. From the pop-up menu of the file, module, or interface, click **Properties** to open the appropriate wizard.
2. Click the page title to select a page to turn to.
3. Change your selections as necessary.
4. Click **Finish** to apply your changes.

#### Note the following points:

- To change the deployment platforms for the object, you must edit the object’s file properties. See the task: “Setting platform constraints” on page 421.
- If you want to specify a parent for the interface after you have defined the implementation for the business object, follow these steps:
  1. Add the parent to the Parents folder on the Interface Inheritance page of the Business Object Interface wizard
  2. Open the Business Object Implementation wizard, and on the Name and Data Access Pattern page specify the pattern for handling state data as **Same as parent’s**.
  3. Click **Next**.
  4. Add the implementation parent on the Implementation Inheritance page.
- If you want to change the order of the business object file contents, follow these steps:
  1. Open the file’s wizard.
  2. Click the page title and turn to the Contents Ordering page.
  3. Move elements into the new order.
  4. Click **Order by Dependency** to validate the new order.
  5. Click **Finish**.

#### RELATED CONCEPTS

Business object (*Programming Guide*)

“Dependencies within an IDL file” on page 292

#### RELATED TASKS

“Working with business objects” on page 774

“Setting platform constraints” on page 421

#### RELATED REFERENCES

“Internationalization of data” on page 132

“Naming objects” on page 128

## Editing a business object implementation



Business object implementations are defined in the User-Defined Business Objects folder, where they are shown under the business object interface they were added to.

You can edit a business object implementation by following these steps:

1. From the pop-up menu of the business object implementation, click **Properties**. The Business Object Implementation wizard opens to the Name and Data Access Pattern page.
2. Click the page title to select another page to turn to.
3. Change your selections as necessary.
4. Click **Finish** to apply your changes.

#### Note the following points:

- To change the deployment platforms for the object, you must edit the object's file properties. This is described in the task: “Setting platform constraints” on page 421.
- The **Same as parent's** option is selected by default if the interface for this business object inherits from another business object interface. However, you still have to indicate the implementation parent on the Implementation Inheritance page of this wizard, after you delete the default parent for business object implementations, which is `IManagedClient` `IManagedClient::IManageable`.
- You cannot change the business object implementation file name, and the deployment platforms using the Name and Data Access Patterns page. To change these properties of the implementation, select **File Properties** from the pop-up menu of the business object implementation. The Business Object Implementation File wizard opens to the Name page, and you can make the changes.

- On the Key and Copy Helper page, only those keys and copy helpers that do not contain attributes from other business object implementations are available for selection.

#### RELATED CONCEPTS

Business object (*Programming Guide*)

#### RELATED TASKS

“Working with business objects” on page 774

“Setting platform constraints” on page 421

#### RELATED REFERENCES

“Internationalization of data” on page 132

“Naming objects” on page 128

## Editing a business object implementation file

To edit a business object implementation file, follow these steps:

1. From the pop-up menu of the business object implementation, select **File Properties**. The Business Object Implementation File wizard opens to the Name page.
2. You can type a new name for the business object implementation file, and select a different set of deployment platforms for the object.  
**Note:** You can select or clear only check boxes that correspond to platforms that are selected on the **Platform > Constrain** menu. The other check boxes will be disabled.
3. Click **Next**. The Contents Ordering page opens. Use this page to view or set the order of constructs and interfaces in the IDL file.
4. Click **Finish** to apply your changes.

### Editing a business object implementation file to match a new business object implementation

If you create a new implementation file when you create a business object implementation, Object Builder associates the deployment platforms that you specify with that file. If you create a new business object implementation, and add it to an existing file (that is, you specify the name of a business object implementation file that already exists in the model in the **File Name** field), Object Builder checks whether the deployment platforms that you specified for the implementation are the same as those that are selected for the existing implementation file. If they are not the same, you can change the existing business object implementation file’s platforms to be the same as those for the implementation that you are adding, before adding the new implementation: From the pop-up menu of the existing business object implementation, select **File Properties**, and change the deployment platforms on the Name page of the Business Object Implementation File wizard.

#### RELATED CONCEPTS

Business object (*Programming Guide*)  
“Multi-platform development” on page 419

#### RELATED TASKS

“Working with business objects” on page 774  
“Setting platform constraints” on page 421

#### RELATED REFERENCES

“Deployment platforms” on page 423  
“Platform differences” on page 425  
“Internationalization of data” on page 132  
“Naming objects” on page 128

## Deleting a business object interface

To delete a business object interface, follow these steps:

1. Locate the business object interface in the User-Defined Business Objects folder.
2. Delete any managed objects defined off of the interface’s business object implementation.
3. Delete or remove the data object interface defined off of the business object implementation.
4. Delete the business object implementation.
5. Delete any keys and copy helpers defined off of the interface.
6. From the pop-up menu of the business object interface, click **Delete**.

When you delete a business object interface, any methods, attributes, constructs, or one-to-many relationships that use it as a type have the type changed to `invalidType`. For example, if you delete the interface of `Agent`, then an attribute `Agent custAgent` becomes attribute `invalidType custAgent`. You can find all occurrences of `invalidType` by running the model consistency checker.

#### RELATED CONCEPTS

Business object (*Programming Guide*)

#### RELATED TASKS

“Working with business objects” on page 774  
“Checking a model for consistency” on page 31

## Deleting a business object implementation



To delete a business object implementation, follow these steps:



1. Locate the business object implementation in the User-Defined Business Objects folder.
2. Delete any managed objects defined off of the business object implementation.
3. Delete or remove the data object interface defined off of the business object implementation.
4. From the pop-up menu of the business object implementation, click **Delete**.

**Note:**When you delete a business object implementation, any of its attributes that are used in its associated keys, or copy helpers are automatically deleted.

#### **RELATED CONCEPTS**

Business object (*Programming Guide*)

#### **RELATED TASKS**

“Working with business objects” on page 774

“Deleting a managed object” on page 874

“Deleting a data object interface” on page 824

“Working with keys” on page 825 “Working with copy helpers” on page 829

---

## **Working with data objects**

A data object manages the state of a business object. It encapsulates the object’s persistent behavior, if there is any.

In the User-Defined Data Objects folder, a data object is fully presented in terms of four objects:

- The data object file (which contains one or more interfaces, optionally organized into modules)
- The data object module, if any (which contains one or more interfaces)
- The data object interface (which has one or more implementations)
- The data object implementation (which has its own file, defined on the first page of its wizard)

You can create these objects separately when you create a data object interface that is not connected to a business object. See the task “Creating a data object interface” on page 801. Collectively, these objects form a single data object. Each data object (each set of data object file, module, interface, and implementation) typically has its own persistent object.

The objects are created automatically in the User-Defined Data Objects folder when you perform one of the following tasks:

- “Adding a data object from a DB persistent object” on page 814

- “Adding a data object from a PA persistent object” on page 815
- “Adding a data object implementation” on page 807

The following tasks deal with data objects:

- “Creating a data object file” on page 797
- “Editing a data object file” on page 798
- “Adding a data object module” on page 800
- “Creating a data object interface” on page 801
- “Adding a business object implementation and data object interface” on page 780
- “Creating a data object by importing an IDL file” on page 804
- “Adding a data object implementation” on page 807
- “Adding a data object from a business object” on page 813
- “Adding a data object from a DB persistent object” on page 814
- “Adding a data object from a PA persistent object” on page 815
- “Mapping a business object to a data object” on page 787
- “Mapping a data object to a DB persistent object” on page 703
- “Mapping a data object to a PA persistent object” on page 708
- “Editing a data object interface” on page 817
- “Editing a data object implementation” on page 819
- “Editing a data object implementation file” on page 823
- “Deleting a data object interface” on page 824
- “Deleting a data object implementation” on page 824

**Note:** In the User-Defined Business Objects folder, the DBA-Defined Schemas folder and the User-Defined PA Schemas folder, the data object is only presented in terms of its interface and implementation.

#### **RELATED CONCEPTS**

Data object (*Programming Guide*)

Persistent object (*Programming Guide*)

Business object (*Programming Guide*)

#### **RELATED TASKS**

“Working with components” on page 697

#### **RELATED REFERENCES**

“Naming objects” on page 128

“Internationalization of data” on page 132

## Creating a data object file

A data object file (IDL) can hold multiple data object interfaces, which you may organize into modules. However, you typically add one interface to each file.

To create a data object file, follow these steps:

1. From the Tasks and Objects pane, select the **User-Defined Data Objects** folder.
2. From the folder's pop-up menu, select **Add File**. The Data Object File wizard opens to the Name page.
3. Type a name for the file.
4. Click **Next**. The Constructs page opens.

Use the Constructs pop-up menu to add constants, enumerations, exceptions, typedefs, structures, or unions. Any constructs you add are scoped to this file only.

**Note:** To use the construct as the type of an attribute, method return, method exception, or as a type within another construct, you must first click **Finish** and then reopen the wizard before you can use the type. The construct is not added to the current model until you click **Finish**.

5. Click **Next**. The Files to Include page opens.

If your component had a parent, you would specify the data object file of the parent component in this field. For example, if the CarPolicy component inherits from the Policy component, then you would specify the data object file for Policy on this page. The data object files for any referenced or related components are automatically added to the Include Files folder. For example, if the CarPolicy interface has an attribute of type Claim, the data object file for Claim is automatically included on this page as soon as the attribute is defined for the interface. You can also use this page to add headers to the data object interface's IDL file. These headers can be in the form of `#include` statements that include other IDL files that have definitions that you want to use.

6. Click **Next**.

**Note:** If the file has constructs or other interfaces defined in it, continue with step 6; otherwise, continue with step 7.

7. The Contents Ordering page opens. This page enables you to view the order of elements within the IDL file. You can also change the order of these constructs and interfaces within the file by using either the **Move Up** and **Move Down** buttons, or the **Order by Dependency** button. When you are satisfied with the order of elements, click **Next**.
8. The Comments page opens. Type any comments you want to include as comment lines in your generated IDL code.

9. Click **Finish**. The wizard closes, and your file is added to the User-Defined Data Objects folder. You can now add modules or interfaces to the file.

Once you have created the file, you can modify it by selecting **Properties** from its pop-up menu. The Data Object File wizard opens again, with your selections preserved. You can change the name of the file and the construct names or their types only if the file was not defined in another model. (If the file was defined in another model, and you specify that model as a project dependency when you open the current project, the project dependency model is read-only, and can be used for inheritance purposes, and for reuse of interfaces, attribute types and constructs.)

#### **RELATED CONCEPTS**

Data object (*Programming Guide*)

#### **RELATED TASKS**

“Working with data objects” on page 795

“Adding a business object module” on page 777

“Creating a data object interface” on page 801

#### **RELATED REFERENCES**

“Internationalization of data” on page 132

“Naming objects” on page 128

## **Editing a data object file**

You can edit a data object file using either of the following methods:

- Using **Properties** from the pop-up menu of the data object file
- Using **File Properties** from the pop-up menu of either the data object module, or the data object interface

To edit a data object file using the file itself, follow these steps:

1. From the Tasks and Objects pane, select the User-Defined Data Objects folder.
2. From the folder’s pop-up menu, select **Properties**. The Data Object File wizard opens to the Name page.
3. Type a new name for the file.

**Note:** You can change the name of the file, and the names of constructs or their types only if the file was not defined in another model. (If the file was defined in another model, and you specify that model as a project dependency when you open the current project, the project dependency model is read-only, and can be used for inheritance purposes, and for reuse of interfaces, attribute types and constructs.)

4. Click **Next**. The Constructs page opens.

Use the Constructs pop-up menu to add constants, enumerations, exceptions, typedefs, structures, or unions. Any constructs you add are scoped to this file only.

**Note:** To use the construct as the type of an attribute, method return, method exception, or as a type within another construct, you must first click **Finish** and then reopen the wizard before you can use the type. The construct is not added to the current model until you click **Finish**.

5. Click **Next**. The Files to Include page opens.

If your component had a parent, you would specify the data object file of the parent component in this field. For example, if the CarPolicy component inherits from the Policy component, then you would specify the data object file for Policy on this page. The data object files for any referenced or related components are automatically added to the Include Files folder. For example, if the CarPolicy interface has an attribute of type Claim, the data object file for Claim is automatically included on this page as soon as the attribute is defined for the interface. You can also use this page to add headers to the data object's IDL file. These headers can be in the form of `#include` statements that include other IDL files that have definitions that you want to use: for example, exceptions that are defined, and that are to be thrown in push-down methods.

6. Click **Next**.

**Note:** If the file has constructs or other interfaces defined in it, continue with step 6; otherwise, continue with step 7.

7. The Contents Ordering page opens. This page enables you to view the order of elements within the IDL file. You can also change the order of these constructs and interfaces within the file by using either the **Move Up** and **Move Down** buttons, or the **Order by Dependency** button. When you are satisfied with the order of elements, click **Next**.
8. The Comments page opens. Type any comments you want to include as comment lines in your generated IDL code.
9. Click **Finish**. The wizard closes, and the file is saved with your changes.

To edit the data object file using either a data object module, or a data object interface, follow these steps:

1. From the pop-up menu of either the data object module, or the data object file, select **File Properties**. The Data Object File wizard opens to the Name page.
2. You can type a new name for the data object file, and select a different set of deployment platforms for the object.

**Note:** The remaining pages are the same as when you edit the data object file using the file itself: the Constructs page, the Files to Include page, the Contents Ordering page, and the Comments page. Change these properties, if you want to.

3. Click **Finish** to apply your changes.

**RELATED CONCEPTS**

Data object (*Programming Guide*)

**RELATED TASKS**

“Working with data objects” on page 795

“Adding a business object module” on page 777

“Creating a data object interface” on page 801

**RELATED REFERENCES**

“Internationalization of data” on page 132

“Naming objects” on page 128

## Adding a data object module

If you plan to add multiple data object interfaces to a single file, you may want to store the interfaces in separate modules. To add a module to a file, follow these steps:

1. From the User-Defined Data Objects folder, select your data object file.
2. From the file’s pop-up menu, select **Add Module**. The Data Object Module wizard opens to the Name page.
3. Type a name for the module.
4. Click **Next**. The Constructs page opens.

Use the Constructs pop-up menu to add constants, enumerations, exceptions, typedefs, structures, or unions. Any constructs you add are scoped to this module only.

**Note:** To use the construct as the type of an attribute, method return, method exception, or as a type within another construct, you must first click **Finish** and then reopen the wizard before you can use the type. The construct is not added to the current model until you click **Finish**.

5. Click **Next**. The Comments page opens. Type any comments you want to include as comment lines in your generated code.
6. Click **Finish**. The wizard closes, and your module is added to the User-Defined Data Objects folder, underneath the file.

You can now add data object interfaces to the module.

**RELATED CONCEPTS**

Data object (*Programming Guide*)

**RELATED TASKS**

“Working with data objects” on page 795

“Creating a data object interface” on page 801

## RELATED REFERENCES

“Internationalization of data” on page 132

“Naming objects” on page 128

## Creating a data object interface

You can add a data object interface in different situations, from the following folders:

- User-Defined Business Objects folder
- User-Defined Data Objects folder
- DBA-Defined Schemas folder

### From the User-Defined Business Objects folder

Once you have deleted a data object interface that was created along with a business object implementation, you can create a new one to be associated with the implementation. Follow these steps:

1. From the pop-up menu of the unassociated business object implementation in the User-Defined Business Objects folder, select **Add New Data Object Interface**. The Add New Data Object wizard opens to the Data Object Interface page.
2. Accept the default data object file name and interface name, or rename them, and select those attributes of the business object to be used as state data in the data object.
3. Click **Next**. The Data Object Methods page opens.
4. Select the methods of the business object that are to be delegated to the data object. The methods you select form part of the data object’s interface.
5. Click **Next**. The Constructs page opens. Use the pop-up menu of the Constructs folder to add constants, enumerations, exceptions, typedefs, structures, or unions. Any constructs you add are scoped to this interface only.

**Note:** To use the construct as the type of an attribute, method return, or method exception, you must first click **Finish** and then reopen the wizard and define the attribute. The construct is not added to the current model until you click **Finish**.

6. The Interface Inheritance page opens. Here you can specify one or more classes from which the interface can inherit. Click the list button of the **Parent Interface** and select a parent from the list of available classes, or type the interface name using the following syntax: *filename interface\_name*
7. When you have specified all the parents for this interface, click **Next**. The Comments page opens. Type any comments you want to include as comment lines in your generated code.
8. Click **Finish**. The interface is added to the folder.

Once the data object interface is created, its file properties (**File Properties** from the pop-up menu of the object) will include the platforms that you set using the **Platforms > Constrain** menu.

### **From the User-Defined Data Objects folder**

In this folder, you can create a data object interface in the following ways:

- From a data object file
- From a data object module
- By importing an IDL file

### **From a data object file**

1. Select the User-Defined Data Objects folder.
2. From its pop-up menu, select **Add File**. The Data Object File wizard opens to the Name page.
3. Specify a name for the data object file.
4. Click **Finish**. The data object file is added to the folder.
5. Select the file, and from its pop-up menu, select **Add Interface**. The Data Object Interface wizard opens to the Name page.
6. Specify a name for the interface.
7. Click **Next** if you want to define constructs at the interface level.
8. Go to the Interface Inheritance page if you want this interface to inherit from an existing one.
9. Go to the Attributes page if you want to define attributes that are specific to the data object interface.
10. Go to the Methods page if you want to define methods for the interface.
11. Add any comments you want to, on the Comments page.
12. Click **Finish**.

The data object interface is added to the folder, and appears as a node beneath the file.

### **From a data object module**

If you want the data object interface to be scoped within a module, follow this method.

Follow steps 1 to 4 that are outlined in the previous method. Once the data object file is created, follow these steps:

1. Select the file, and from its pop-up menu, select **Add Data Object Module**. The Data Object Module wizard opens to the Name page.
2. Type a name for the module.
3. Click **Next** if you want to add constructs at the module level.



4. Go to the Comments page if you want to add comments about the module.
5. Click **Finish**. The data object module is created, and appears as a node beneath the data object file.
6. Select the module in the folder, and from its pop-up menu, select **Add Interface**. The Data Object Interface wizard opens to the Name page.

Follow steps 6 to 12 as when you added the interface from the file. The data object interface is added to the folder, and appears as a node beneath the module.

### By importing an IDL file

This is actually a method of reusing an existing data object interface. You add it, or create it within Object Builder by importing it. Follow these steps:

1. Select the User-Defined Data Objects folder.
2. From its pop-up menu, select **Import > IDL**. The Import IDL wizard opens to the IDL File Selection page.
3. From the IDL Files folder's pop-up menu, select **Add**. You can then specify the IDL file to be imported in the **File Name** field. You can also use the **Browse** button to open the File to Import dialog box. Use it to view the contents of the different drives and find the exact path for the IDL file to be imported.

**Note:** The IDL must be CORBA 2.2-compliant without IDL extensions. You can make sure the IDL files you are importing are valid by compiling them first. Object Builder will only import IDL files that are considered valid by the compiler.

4. Click **Next**. The Search Paths for Nested Files page opens.  
Include files (that is, those that are nested in another file), which are not already in the model, have to be imported. Other include files, (for example, IManagedClient), which exist in the model, do not have to be imported.
5. Indicate the directories that must be searched for the include files that are specified in the IDL file to be imported: from the pop-up menu of the Directories folder, select **Add**. Type the include directory in the **Directory** field.
6. Click **Finish**.

The data object interface appears in the folder beneath the file that contains it, or if it is scoped within a module, beneath the module that contains it.

### From the DBA-Defined Schemas folder

This method assumes you have imported an SQL file into Object Builder, and added a persistent object from it. Follow these steps to add a data object interface from the persistent object:

1. Select the persistent object that you added to the imported schema.
2. From its pop-up menu, select **Add Data Object**.
3. The Add Data Object wizard opens to the Names page.
4. Specify the names for the data object interface and its file, and for the associated data object implementation and its file.
5. Click **Next** to go to the Methods page, if you want to add methods for the data object.
6. Click **Finish**.

The data object interface appears beneath its file in the User-Defined Data Objects folder, with the data object implementation beneath it, with the implementation connected to the persistent object, which in turn is connected to the schema.

#### **RELATED CONCEPTS**

Data object (*Programming Guide*)

Query Service (*Advanced Programming Guide*)

#### **RELATED TASKS**

“Working with data objects” on page 795

“Creating a DB schema by importing an SQL file” on page 844

“Adding a persistent object from a DB schema” on page 837

“Creating a data object by importing an IDL file”

#### **RELATED REFERENCES**

“Internationalization of data” on page 132

“Naming objects” on page 128

## **Creating a data object by importing an IDL file**

If you have code already in IDL files, you can parse the code into Object Builder, and incorporate the classes, relationships, and code in the IDL files into your Object Builder application.

**Note:** The IDL must be CORBA 2.2-compliant without IDL extensions. You can make sure the IDL files you are importing are valid by compiling them first. Object Builder will only import IDL files that are considered valid by the compiler.

To import an existing data object interface IDL file, follow these steps:

1. Under Tasks and Objects, select the User-Defined Data Objects folder.
2. From the folder’s pop-up menu, select **Import IDL**. The Import IDL wizard opens to the IDL File Selection page.

3. Browse for, and select the files you want to import. The files you select, and any files they include, will be parsed and imported into Object Builder.  
**Note:** The files will be imported under the User-Defined Business Objects folder. You will have to delete the data object interface files from the business object interface.
4. Click **Next**. The Search Paths for Nested Files page opens.
5. From the **Directories** pop-up menu, select **Add**. Browse for the directories you want searched.  
When you import a file that includes other files (that is, a file with nested files), the import process will search for the other files in the directories you specify here.
6. Click **Finish**. The selected files (and files they include) are parsed into Object Builder, and the information in the IDL is added to the current project model as data object files, data object modules, and data object interfaces.

### **Consequences of importing an interface that has dependencies on other interfaces**

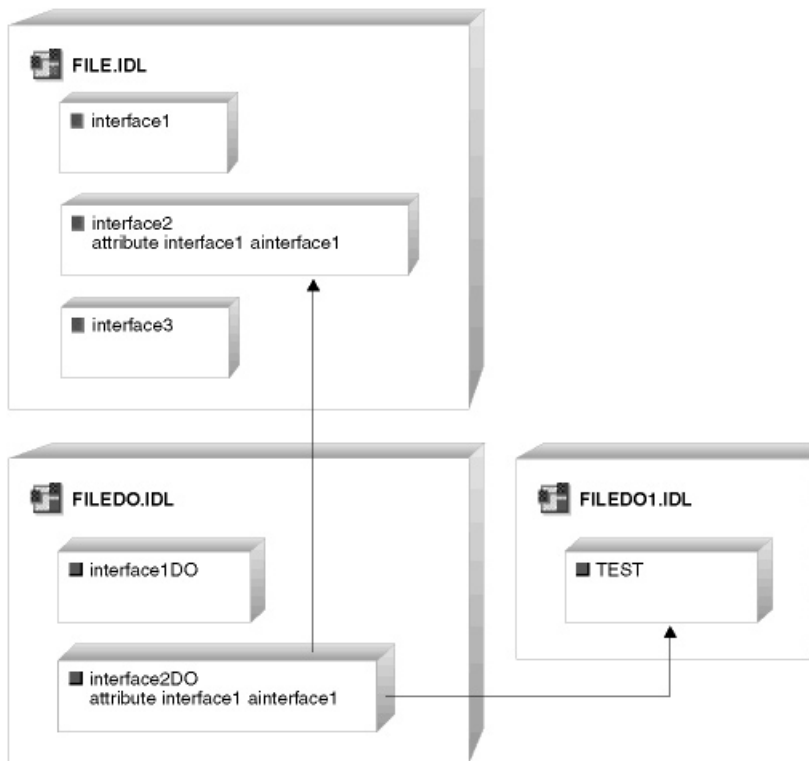
If the data object interface that you import inherits from another object - either a business object or another data object, the interfaces of those objects are imported into the model as well, and appear under the User-Defined Business Objects folder.

Follow these steps to complete the import process:

1. Specify the directory the parent interfaces are in, on the Include Files page of this wizard.
2. Delete the extraneous data object interfaces from the User-Defined Business Objects folder (from the pop-up menu of the interface, select **Delete**).
3. Import the same data object interfaces again into the User-Defined Data Objects folder.  
**Note:** Even after you import the data object interfaces, inheritance will no longer work because the definition of the interface changes as soon as the objects are deleted from the User-Defined Business Objects folder. The IDL file that you generate from this interface will not contain the include statements. So, follow step 4.
4. Open the Data Object Interface wizard and add the parent interfaces on the Interface Inheritance page.

### **Example**

1. You have the following IDL files:



1. FileD0.idl has an interface that inherits from the test interface of the IDL file FileD01.idl , and another one that has an attribute of a type defined in an interface of the file File.idl .
2. You try to import FileD0.idl (Select **Import IDL** from the pop-up menu of the User-Defined Data Objects folder and specify FileD0.idl as the file to be imported.)
3. The business object IDL file File is imported into the User-Defined Business Objects folder, along with its three interfaces. The data object IDL file FileD0 is also imported into the User-Defined Business Objects folder, along with its interface.
4. You must delete the FileD01 from the User-Defined Business Objects folder.
5. Select **Import IDL** from the pop-up menu of the User-Defined Data Objects folder and this time, specify FileD01.idl as the file to be imported.
6. Open FileD0's Data Object Interface wizard. Turn to the Interface Inheritance page and select interface2 and test as the parent interfaces.
7. Click **Finish**.

## Work-around

If you have an IDL file you want imported, and you know the interfaces it depends on (the other data object interfaces it inherits from), and the other .idl files from which it wants to use information (for example, you want to use the exceptions defined in some business object interface IDL file), you can import the data object IDL file without the business object interface and other supporting data object interfaces being imported into the User-Defined Business Objects folder, if you follow these steps:

1. Import the data object interfaces from which this interface is to inherit, before importing the interface itself (on the IDL File Selection page).
2. Include the business object interface IDL files which have the information required for the interface, on the Include Files page of the same wizard.

**Note:** If a data object has an interface and you create a new interface for it, when you generate code for the data object (file level), the interface you just created will appear last in the code. If you want the old interface to inherit from the new one, and you specify this on the Interface Inheritance page of the Data Object Interface wizard, and then generate code from the data object file, Object Builder places the definition of the new interface in the beginning, before the definition of the interfaces that inherit from it.

### RELATED CONCEPTS

Interface Definition Language (*Programming Guide*)

### RELATED TASKS

“Importing IDL from the command line” on page 666

“Creating a business object by importing an IDL file” on page 780

“Creating a local-only object by importing an IDL file” on page 669

## Adding a data object implementation



Once you have either created or selected the data object interface for your business object, you can create a data object implementation. The business object is dependent on the data object interface, but not on its implementation. However, for the business object to be of any use, the data object interface must be implemented. The data object implementation can emulate the real application environment with a full client-server setup.

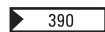
To create a data object implementation, follow these steps:

1. From the Tasks and Objects pane, select the data object interface for which you want to create an implementation (for example, CarPolicyDO).

2. From the pop-up menu of the interface, select **Add Implementation**. The Data Object Implementation wizard opens to the Name and Platform page.
3. Type the name of the implementation class and its file, and optionally the implementation module; or, accept the default names (for example FileDOImpl for the implementation file name, IntDOImpl for the implementation interface name, and ModDOImpl for the implementation module name). If the data object interface is contained in a module, specify the module name in the **Module Name** field.
4. Set the deployment platforms (the platforms on which this data object will be deployed). This determines the development options that are selectable (you can only select options that are available on all selected platforms). By default, the data object is deployable to the set of platforms defined in the **Platforms > Constrain** menu. You cannot select platforms that are **not** already selected in the **Platforms > Constrain** menu.

**Note:** If you create a new implementation file when you create the data object implementation, Object Builder associates the deployment platforms that you specify with that file. If you create a new data object implementation, and add it to an existing file (that is, you specify the name of a data object implementation file that already exists in the model in the **File Name** field, Object Builder checks whether the deployment platforms that you specified for the implementation are the same as those that are selected for the existing implementation file. If they are not the same, you must follow one of these procedures:

  - Change the existing implementation file's platforms to be the same as those for the implementation you are adding, before adding the new implementation: From the pop-up menu of the existing data object implementation, select **File Properties**, and change the deployment platforms on the Name page of the Data Object Implementation File wizard.
  - Change the implementation's platforms to match those of the existing file, on this page (the Name and Platform page).
5. Click **Next**. The Behavior page opens.
6. Select the environment for testing the object in the **Environment** section. See the reference section: "Environment" on page 249.
7. Select the type of persistence from the **Type of Persistence** section of the same page. See the reference section: Type of Persistence. All options in this section are available for selection only if you have selected **BOIM with any key** in step 6.




The **Cache Service** option is not available when the target platform is OS/390.

The **MQSeries Adaptor** option is not selectable if one of the deployment platforms is either AIX, or OS/390.

8. Select the data access pattern to be used in the data object implementation from the **Data Access Pattern** section. See the reference section: “Data Access Pattern” on page 254. This section is available for selection only if you have selected **BOIM with any key** in step 6. The access pattern can be either **Delegating**, which is the default option, or **Local copy**.
9. Select the handle for storing references from the **Handle for Storing Pointers** section. See the reference section: “Handle for Storing Pointers” on page 255.  
**Note:** Your selection of a handle does not affect the default mapping of object references on the Attributes Mapping page of this wizard. It is relevant only if you override the default setting of the persistent object and schema’s attribute mapping from **Key Home** to **Primitive**.
10. Click **Next**. The Implementation Inheritance page opens. Click the list button of the **Parent Class** field and select a parent, or accept the default, which is provided by Object Builder.
11. Click **Next**. The Attributes page opens. You can use this page to add more attributes for the data object. These attributes will be specific to this implementation.
12. Click **Next**. The Methods page opens. Here, you can add implementation-specific methods for the data object. These methods can access the implementation-specific attributes you added on the previous page. These methods can also be called from within other methods that you define for the data object interface.
13. Click **Next**. The Key and Copy Helper page opens. Select the key and the copy helper to use with this implementation.  
**Note:** If you selected **BOIM with any key** in step 5, and there is at least one persistent object in the model that has the same type of persistence as that for the implementation, which you specify in step 6, the Associated Persistent Objects page is added to the wizard, and you can continue with step 14; if not, continue with step 17.
14. Click **Next**. The Associated Persistent Objects page opens. You can specify existing persistent object instances that are to be associated with the data object. These instances are stored as protected members of the data object implementation.

**Note the following points:**

- At this point, you can also finish the wizard and add a persistent object and schema from the data object implementation. The persistent object you create is automatically associated with the data object and appears in the Persistent Object Instances folder on the Associated Persistent Objects page.

- If you associated one or more existing persistent object instances with the data object, the Attributes Mapping page and the Methods Mapping page are added to the wizard. You can map the data object to the persistent object: continue with step 15. If you did not associate any persistent object with the data object, continue with step 17.
  - You can also finish the wizard, and from the data object implementation's pop-up menu, select **Select Persistent Object and Schema** to associate an existing schema and persistent object with the implementation, and to map attributes and methods of the data object to those of the persistent object.
15. Click **Next**. The Attributes Mapping page opens. You can specify the mapping between the data object attributes and the persistent object attributes.
  16. Click **Next**. The Methods Mapping page opens. For each of the special framework methods associated with the implementation, you can specify the persistent object methods that it calls. The order of the methods in the tree determines the order in which they are called.  
**Note:** If you have associated any Procedural Adaptor (PA) persistent objects with this implementation, this page has the User-Defined Methods folder as well. This folder contains the methods you defined for the data object using the Methods page of the Data Object Interface wizard. You can map each of these methods to the corresponding push-down method of the persistent object, if you want these methods to be called immediately.
  17. Click **Next**. The Discriminators page opens.  
 This page is available for polymorphic home support only if the constraint platforms do not include either OS/390 or HP-UX.  
 If you are using a single, backend table, you can use this page to specify the discriminator predicate (typecode) that will allow the data object implementation to isolate appropriate rows from the table. See the task Using customized discriminator predicates
  18. Click **Next**. The Summary of Framework Methods page opens. On this page you can review the framework methods that are implemented by Object Builder for this class. These methods have to be implemented for the data object to work properly in the server.
  19. Click **Finish**. The data object implementation appears in the Tasks and Objects pane, with the appropriate methods added in the Method List pane. You can now add your own code to implement those methods. None of the framework methods have implementations at this point. Object Builder provides the code for these methods as soon as there is a persistent object associated with this data object.

Sometimes, you will have an already-existing data object implementation, associated with a non-PA persistent object (for example, if you started



working against a database backend to get things up and running), which you now want to associate with a PA persistent object. To change the persistent object associated with a data object implementation, go to the Associated Persistent Object page in the data object implementation's Properties wizard. See "Associating a PA persistent object with an existing data object implementation" on page 822 for more information.

#### **RELATED CONCEPTS**

Data object (*Programming Guide*)  
Persistent object (*Programming Guide*)  
Object relationships (*Programming Guide*)  
Application adaptor (*Programming Guide*)  
Data object customization for cardinality relationships (*Programming Guide*)  
State data (*Programming Guide*)  
Data object implementation (*Programming Guide*)  
"Container" on page 578  
"Home" on page 581  
Using handles (*Programming Guide*)  
Naming Service (*Advanced Programming Guide*)  
Cache Service (*Advanced Programming Guide*)  
"Special framework methods" on page 758  
"Framework methods" on page 757  
Using Sets of Objects (Using Reference Collections) (*Programming Guide*)

#### **RELATED TASKS**

"Working with data objects" on page 795  
Using customized discriminator predicates  
"Adding a persistent object and schema" on page 833  
"Setting platform constraints" on page 421  
"Customizing referential integrity" on page 714  
"Creating a container instance" on page 578  
"Configuring a managed object" on page 588  
"Working with attributes" on page 697  
"Working with methods" on page 750  
"Implementing methods" on page 752  
"Adding resource methods to a sessional business object" on page 164  
"Associating a PA persistent object with an existing data object implementation" on page 822

#### **RELATED REFERENCES**

"Data Object Implementation Inheritance" on page 257  
"Internationalization of data" on page 132  
"Naming objects" on page 128

## **Associating a persistent object with a data object**

You can associate a persistent object with a data object in the following ways:

- When you define the data object implementation. See the task “Adding a data object implementation” on page 807.
- After you have defined the data object implementation, when you edit its properties. See the task “Editing a data object implementation” on page 819.
- After you have defined the data object implementation, by selecting a pop-up menu option. This option cannot be used to associate a PA persistent object with a data object implementation. Follow these steps:
  1. Select the data object implementation, and from its pop-up menu, select **Select Persistent Object and Schema**. The Select Persistent Object and Schema wizard opens to the Associated Persistent Objects page, where you can select one or more existing persistent object instances to be associated with the data object implementation. These instances are stored as protected members of the data object implementation.
  2. Click **Next**. The Attributes Mapping page opens. You can specify the mapping between the data object attributes and the persistent object attributes.
  3. Click **Next**. The Methods Mapping page opens. For each of the special framework methods associated with the implementation, you can specify the persistent object methods that it calls. The order of the methods in the tree determines the order in which they are called.
  4. Click **Next**. The Discriminators page opens. You can specify the discriminator predicate that will be used if this data object implementation will be configured against a polymorphic home.
  5. Click **Next**. The Summary of Framework Methods page opens. On this page you can review the framework methods that are implemented by Object Builder for this class. These methods have to be implemented for the data object to work properly in the server.
  6. Click **Finish**. The data object implementation appears in the Tasks and Objects pane, with the appropriate methods added in the Methods pane. You can now add your own code to implement those methods. Object Builder provides the code for the framework methods as soon as there is a persistent object associated with this data object.

#### **RELATED CONCEPTS**

Data object (*Programming Guide*)  
 Persistent object (*Programming Guide*)  
 State data (*Programming Guide*)  
 Data object implementation (*Programming Guide*)  
 “Special framework methods” on page 758  
 “Framework methods” on page 757  
 Using sets of objects (Using reference collections) (*Programming Guide*)

#### **RELATED TASKS**

“Working with data objects” on page 795

“Adding a data object implementation” on page 807  
“Editing a data object implementation” on page 819.  
“Adding a persistent object and schema” on page 833  
Associating a PA persistent object with an existing data object implementation  
“Customizing referential integrity” on page 714  
“Working with attributes” on page 697  
“Working with methods” on page 750  
“Implementing methods” on page 752  
“Adding resource methods to a sessional business object” on page 164

#### RELATED REFERENCES

“Data Object Implementation Inheritance” on page 257

## Adding a data object from a business object

A business object is normally created with its own data object. If however, you delete the associated data object, or select the **Add or select one later** option when you add the business object implementation (in the section “Data Object Interface” on page 247, on the Name and Data Access Pattern page of the Business Object Implementation wizard), you can either select a data object interface that exists in the model, or define an entirely new one for the business object.

To add a data object from a business object, follow these steps:

1. From the pop-up menu of the business object, select Add Data Object Interface. The Add New Data Object wizard opens to the Data Object Interface page.

Appropriate data object names are filled in for you (the business object file name and interface name plus DO: for example, CarPolicy gets the data object interface CarPolicyDO). You can accept these defaults or replace them with your own names.

Select the business object attributes that you want preserved in the data object. These attributes constitute the state data for the component.

2. Click **Next** to open the Constructs page. Use the Constructs pop-up menu to add constants, enumerations, exceptions, typedefs, structures, or unions. Any constructs you add are scoped to this interface only.

**Note:** To use the construct as a type within another construct, you must first click **Finish** and then reopen the wizard before you can use the type. The construct is not added to the current model until you click **Finish**.

3. Click **Next** to open the Interface Inheritance page. Here you can specify one or more classes from which the interface can inherit. Click the list button of the **Parent Interface** and select a parent from the list of available classes, or type the interface name using the following syntaxes:

filename interface\_name (if the interface is stand-alone)  
filename module\_name::interface\_name (if the interface derives from a module)

4. When you have specified all the parents for this interface, click **Next**.
5. Type any comments you need to add, on the Comments page.
6. Click **Finish**. The interface is added to the folder.

#### **RELATED CONCEPTS**

Business object (*Programming Guide*)  
Data object (*Programming Guide*)

#### **RELATED TASKS**

“Working with data objects” on page 795

#### **RELATED REFERENCES**

“Internationalization of data” on page 132  
“Naming objects” on page 128

## **Adding a data object from a DB persistent object**

To add a data object to a persistent object, follow these steps:

1. From the pop-up menu of the persistent object in the DBA-Defined Schemas folder, select **Add Data Object**. The Add Data Object wizard opens to the Names page.
2. Type a name for the interface file in the **Interface File Name** field, or accept the default.
3. Type a name for the data object interface in the **Interface Name** field, or accept the default.
4. Type a filename for the data object implementation in the **Implementation File Name** field, or accept the default.
5. Type a name for the data object implementation in the **Implementation Name** field, or accept the default.
6. Click **Finish**.

The data object appears in the User-Defined Data Objects folder under the data object filename you provided. The data object interface exists in the tree as a child of the data object file, and the data object implementation exists as a child of the interface. The data object implementation has the persistent object as its child node and the persistent object has the schema as its child node.

#### **Note the following points:**

- A default mapping is generated between the attributes of the data object and the persistent object. If there are mappings generated that need a

mapping helper, Object Builder will inform you for which pairs of attributes it is required. You can then follow these steps:

1. Select the data object implementation that was just created.
  2. Select **Properties** from its pop-up menu. The Data Object Implementation wizard opens.
  3. Click **Next**, or click the arrow to the left of the page name, and select Attributes Mapping page from the list. The page opens.
  4. Select **Map using a helper class** and provide the mapping helper class and method names for each pair of attributes.
- You can initialize the attributes of the data object interface that is created. (They are not initialized by default.) Follow these steps:
    1. Select the data object interface in the User-Defined Data Objects folder.
    2. From the pop-up menu of the data object interface, select **Properties**. The Data Object Interface wizard opens.
    3. Click **Next**, or click the arrow to the left of the page name, and select Attributes page from the list. The page opens.
    4. Select the data object attributes from the Attributes folder and type the initial value for the attribute in the **Initializer** field.

#### **RELATED CONCEPTS**

Data object (*Programming Guide*)

Persistent object (*Programming Guide*)

Schema (*Programming Guide*)

#### **RELATED TASKS**

“Working with data objects” on page 795

“Working with DB persistent objects” on page 832

“Mapping attributes using a mapping helper” on page 738

#### **RELATED REFERENCES**

“DB2 data type mappings” on page 142

“Oracle data type mappings” on page 146

“Internationalization of data” on page 132

“Naming objects” on page 128

## **Adding a data object from a PA persistent object**

To add a data object from a PA persistent object, follow these steps:

1. From the pop-up menu of the PA persistent object in the User-Defined PA Schemas folder, select **Add Data Object**. The Add Data Object wizard opens to the Names page.
2. Type a name for the interface file in the **Interface File Name** field, or accept the default.

3. Type a name for the data object interface in the **Interface Name** field, or accept the default.
4. Type a filename for the data object implementation in the **Implementation File Name** field, or accept the default.
5. Type a name for the data object implementation in the **Implementation Name** field, or accept the default.
6. Click **Next**. The Methods page opens. To specify methods for your interface, select **Add** from the Methods folder's pop-up menu. You can add new user-defined methods to the data object, and also modify them (except, you cannot map them to existing persistent object methods).

The data object appears in the User-Defined Data Objects folder under the data object filename you provided. The data object interface exists in the tree as a child of the data object file, and the data object implementation exists as a child of the interface. The data object implementation has the PA persistent object as its child node and the persistent object has the PA schema object as its child node.

**Note the following points:**

- A default mapping is generated between the attributes of the data object and the persistent object. If there are mappings generated that need a mapping helper, Object Builder will inform you for which pairs of attributes it is required. You can then follow these steps:
  1. Select the data object implementation that is just created.
  2. Select **Properties** from its pop-up menu. The Data Object Implementation wizard opens.
  3. Click the arrow to the left of the page name, and select **Attributes Mapping** page from the list.
  4. Select **Map using a helper class** and provide the mapping helper class and method names for each pair of attributes.
- You can initialize the attributes of the data object interface that is created. (They are not initialized by default.) Follow these steps:
  1. Select the data object interface in the User-Defined Data Objects folder.
  2. From the pop-up menu of the data object interface, select **Properties**. The Data Object Interface wizard opens.
  3. Click **Next**, or click the arrow to the left of the page name, and select **Attributes** page from the list. The page opens.
  4. Select the data object attributes from the **Attributes** folder and type the initial value for the attribute in the **Initializer** field.

#### RELATED CONCEPTS

Data object (*Programming Guide*)  
Persistent object (*Programming Guide*)  
Schema (*Programming Guide*)

#### RELATED TASKS

“Creating a component for PA data” on page 157  
“Working with data objects” on page 795  
“Working with PA persistent objects” on page 860  
“Mapping attributes using a mapping helper” on page 738

## Editing a data object interface

You can edit a data object interface, whether it was created with a business object or created by itself.

To edit a data object interface that was created with a business object, follow these steps:

1. From the pop-up menu of the data object interface in the **User-Defined Business Objects** folder, select **Properties**. The Data Object Interface wizard opens to the Name page. You cannot change the name of the data object file, or module. You can rename the data object interface: specify a new name in the **Name** field.  
**Restriction:** You cannot change the name of the interface if it inherits from another.
2. Click **Next**. The Interface Inheritance page opens. You can add new parents for the interface, delete the ones specified earlier, or rename them.
3. Click **Next**. The Methods page opens. Make changes as required.
4. Click **Next**. The Comments page opens. Type any remarks, if you want to.
5. Click **Finish**. The changes you made to the interface are saved and can be viewed later by examining the same wizard using the same option from the data object interface’s pop-up menu.

#### Note the following points:

- To change the deployment platforms for the object, you must edit the object’s file properties. See the task “Setting platform constraints” on page 421.
- To ensure that valid code is generated after a rename, use **Generate > All** instead of **Generate > Selected** from the pop-up menu of the object.
- If the data object has dependent objects such as a business object, key, or copy helper, and you change an attribute of the data object, you must make the change in the dependent objects as well. For each of the objects, follow these steps:
  1. Open the object’s wizard.

## 2. Click **Finish**.

To edit a stand-alone data object interface, follow these steps:

1. From the pop-up menu of the data object interface in the **User-Defined Data Objects** folder, select **Properties**. The Data Object Interface wizard opens to the Name page. You can rename the data object interface if it does not inherit from another.
2. Click **Next**. The Interface Inheritance page opens. You can add new parents for the interface, delete the ones specified earlier, or rename them.
3. Click **Next**. The Attributes page opens. You can add new attributes for the interface, or delete or rename the ones specified earlier.

### **Note the following points:**

- You can only access the Attributes page when you are viewing the properties of the data object interface from the **User-Defined Data Objects** folder.
  - If there is a data object implementation created from this data object interface, follow these steps to refresh the model after you modify the interface in any way:
    - a. From the pop-up menu of the data object implementation, select **Properties**. The Data Object Implementation wizard opens to the Name and Platform page.
    - b. Click **Finish**, or turn to any other page of the wizard and click **Finish**.
  - If there is a persistent object associated with the data object that has a mapping defined between the attributes of these objects, and you change any of the data object's attributes, these attributes lose the defined mapping. You will either have to remap the attributes of the data object to those of the persistent object (on the Attributes Mapping page of the Data Object Implementation wizard), or you can delete the persistent object and schema, and create a new one, which will automatically use the changed attributes.
4. Click **Next**. The Methods page opens. Add new methods or make changes as required.
  5. Click **Next**. The Comments page opens. Type any remarks, if required.
  6. Click **Finish**. The changes you made to the interface are saved and can be viewed later by opening the same wizard using the same option from the data object interface's pop-up menu.

Whichever data object interface you are editing, if you want to change the order of the file's contents (modules, interfaces and constructs within the file), follow these steps:

1. Open the data object file's wizard.



2. Click the arrow to the left of the page name, and select Contents Ordering page from the list.
3. Move elements into the new order.
4. Click **Order by Dependency** to validate the new order.
5. Click **Finish**.

#### RELATED CONCEPTS

Data object (*Programming Guide*)

“Dependencies within an IDL file” on page 292

#### RELATED TASKS

“Working with data objects” on page 795

#### RELATED REFERENCES

“Internationalization of data” on page 132

“Naming objects” on page 128

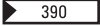
## Editing a data object implementation



To edit a data object implementation, follow these steps:

1. Select the data object implementation in either the User-Defined Business Objects folder or the User-Defined Data Objects folder. From its pop-up menu, select **Properties**. The Data Object Implementation wizard opens to the Name and Platform page. Change the names of the data object implementation’s interface, or module, if you want to.  
**Restriction:** You cannot change the names of the objects if the implementation inherits from another data object implementation.  
**Note the following points:**
  - You cannot change the data object implementation file name, and the deployment platforms using this page. To change these properties of the implementation, select **File Properties** from the pop-up menu of the data object implementation. The Data Object Implementation File wizard opens to the Name page, and you can make the changes.
  - To ensure that valid code is generated after a rename, use **Generate > All** instead of **Generate > Selected** from the pop-up menu of the object.
2. Click **Next** to turn to the Behavior page.
3. You can modify the type of implementation. You can select to test your business object either in a distributed environment, or as a stand-alone. Select one of the different options in the **Environment** section. See the reference topic: “Environment” on page 249.
4. You can change the data object behavior and its implementation in the **Type of Persistence** section of the same page. See the reference topic: Type of Persistence. All options in this section are available for selection

only if you have selected **BOIM with any key** in step 3.

 The **Cache Service** option is not available when the target platform is OS/390.

5. The choices available in the next section, the **Data Access Pattern** section (See the reference section: “Data Access Pattern” on page 254.) are determined by your selection of the type of persistence in step 4. The access pattern is either **Delegating** or **Local copy**.
6. Select the handle for storing references to objects in the **Handle for Storing Pointers** section. See the reference section: “Handle for Storing Pointers” on page 255.
7. Click **Next**. The Implementation Inheritance page opens. You can specify new parent classes for the implementation to inherit from, or delete existing ones.
8. Click **Next**. The Attributes page opens. You can use this page to add more attributes for the data object. These attributes will be specific to this implementation.
9. Click **Next**. The Methods page opens. Here, you can add implementation-specific methods for the data object.
10. Click **Next**. The Select Key and Copy Helper page opens. Select a different key and copy helper, if necessary.

**Note:** If you selected **BOIM with any key** in step 3, and there is at least one persistent object in the model that has the same type of persistence as that for the implementation, which you specify in step 4, the Associated Persistent Objects page is added to the wizard, and you can follow steps 11 through 15; if not, continue with step 14.

11. Click **Next**. The Associated Persistent Objects page opens. You can specify new persistent object instances that are to be stored as protected members of the data object implementation, and edit or delete existing ones.

**Note:** If the PA persistent object you are looking for does not appear in this page, or if the page does not appear at all, there is likely an error in the PA schema definition or the data object implementation definition. Check to make sure the deployment platforms selected for the data object implementation are correct, and that they match up with the platforms that are supported by the PA schema’s connector type. For example, you cannot associate a persistent object that uses LU 6.2 (APPC) connector with a data object implementation that is targeted for OS/390 (or multiple platforms including OS/390), because Component Broker does not support APPC as a connector type for OS/390. For more information, see “Associating a PA persistent object with an existing data object implementation” on page 822.

12. Click **Next**. The Attributes Mapping page opens. You can specify or change the mapping between the data object attributes and the persistent object attributes.

13. Click **Next**. The Methods Mapping page opens. For each of the special framework methods associated with the implementation, you can specify the persistent object methods that it calls. You can change the mappings by adding new persistent object methods, deleting existing ones or changing the order of the methods in the tree. Their order determines the order in which they are called.

**Note:** If you have associated any PA persistent objects with this implementation, this page has the User-Defined Methods folder as well. This folder contains methods you defined at the data object interface level. When the business object uses Session Service, you can provide your own code to be called during some of the normal processing for those services. You can then map each of these methods to the corresponding push-down method of the persistent object. When a business object uses Session Service, you can provide your own code to be called during some of the normal processing for those services. You can do this by calling the `endResource()`, the `checkpointResource()`, and the `resetResource()` method that you define on the business object, in both C++ and Java implementations.

14. Click **Next**. The Discriminators page opens. If you are using a single, backend table, you can use this page to set or change the type of the the discriminator predicate (typecode) that will allow the data object implementation to isolate appropriate rows from the table.
15. Click **Next**. The Summary of Framework Methods page opens. This page shows you the framework methods that Object Builder implements for this data object implementation. You cannot edit this page.
16. Click **Finish**. The data object implementation appears in the Tasks and Objects pane, with the changes to its properties.

**Note:** If there is no persistent object associated with the data object, none of the special framework methods will have implementations. You can “Implementing methods” on page 752 that you want to implement, or for those for which you do not want to use the implementation provided by Object Builder.

#### **RELATED CONCEPTS**

Data object (*Programming Guide*)

Persistent object (*Programming Guide*)

Application adaptor (*Programming Guide*)

Data object customization for cardinality relationships (*Programming Guide*)

Object relationships (*Programming Guide*)

Data object customization (Storage options) (*Programming Guide*)

“Container” on page 578

“Home” on page 581

State data (*Programming Guide*)

Handles (*Programming Guide*)

Naming Service (*Advanced Programming Guide*)  
Cache Service (*Advanced Programming Guide*)  
“Special framework methods” on page 758  
“Framework methods” on page 757  
Using sets of objects (Using reference collections) (*Programming Guide*)

#### RELATED TASKS

“Working with data objects” on page 795  
“Adding a persistent object and schema” on page 833  
“Setting platform constraints” on page 421  
“Customizing referential integrity” on page 714  
“Creating a container instance” on page 578  
“Configuring a managed object” on page 588  
“Working with methods” on page 750  
“Implementing methods” on page 752  
“Adding resource methods to a sessional business object” on page 164  
“Associating a PA persistent object with an existing data object implementation”

#### RELATED REFERENCES

“Data Object Implementation Inheritance” on page 257  
“Internationalization of data” on page 132  
“Naming objects” on page 128

## Associating a PA persistent object with an existing data object implementation



Sometimes you need to change the persistent object associated with an existing data object implementation. For example, you may have started development using a persistent object created from a database schema, to get things up and running. If your final scenario changes to one that connects to a backend such as CICS or IMS, the data object implementation will need to be associated with a PA persistent object.

To change the association:

1. Select **Properties** from the data object implementation’s pop-up menu.
2. Go to the Behavior page. In the **Type of Persistence** section, ensure that **Procedural Adaptors** is selected.
3. Go to the Associated Persistent Object page.
4. Select the desired PA persistent object.

If the persistent object you are looking for does not appear in the Associated Persistent Object page, or if the page does not appear at all, there is likely an error in the PA schema definition or the data object implementation definition. Check to make sure the deployment platforms selected for the

data object implementation are correct, and that they match up with the platforms that are supported by the PA schema's connector type. For example, you cannot associate a persistent object that uses LU 6.2 (APPC) connector with a data object implementation that is targeted for OS/390 (or multiple platforms including OS/390), because Component Broker does not support APPC as a connector type for OS/390.

Note that the only way to associate a PA persistent object with an existing data object implementation is by using the data object implementation's Properties wizard.

#### **RELATED CONCEPTS**

Data object (*Programming Guide*)

Persistent object (*Programming Guide*)

Application adaptor (*Programming Guide*)

#### **RELATED TASKS**

"Editing a data object implementation" on page 819

"Working with data objects" on page 795

"Adding a persistent object and schema" on page 833

"Setting platform constraints" on page 421

## **Editing a data object implementation file**

To edit a data object implementation file, follow these steps:

1. From the pop-up menu of the data object implementation, select **File Properties**. The Data Object Implementation File wizard opens to the Name page.
2. You can type a new name for the data object implementation file, and select a different set of deployment platforms for the object.  
**Note:** You can select or clear only check boxes that correspond to platforms that are selected on the **Platform > Constrain** menu. The other check boxes will be disabled.
3. Click **Next**. The Contents Ordering page opens. Use this page to view or set the order of constructs and interfaces in the IDL file.
4. Click **Finish** to apply your changes.

### **Editing a data object implementation file to match a new data object implementation**

If you create a new implementation file when you create a data object implementation, Object Builder associates the deployment platforms that you specify with that file. If you create a new data object implementation, and add it to an existing file (that is, you specify the name of a data object implementation file that already exists in the model in the **File Name** field,

Object Builder checks whether the deployment platforms that you specified for the implementation are the same as those that are selected for the existing implementation file. If they are not the same, you can change the existing data object implementation file's platforms to be the same as those for the implementation that you are adding, before adding the new implementation: From the pop-up menu of the existing data object implementation, select **File Properties**, and change the deployment platforms on the Name page of the Data Object Implementation File wizard.

#### **RELATED CONCEPTS**

Data object (*Programming Guide*)

"Multi-platform development" on page 419

#### **RELATED TASKS**

"Working with data objects" on page 795

"Setting platform constraints" on page 421

#### **RELATED REFERENCES**

"Deployment platforms" on page 423

"Platform differences" on page 425

"Internationalization of data" on page 132

"Naming objects" on page 128

## **Deleting a data object interface**

To delete a data object interface, follow these steps:

1. If there is a data object implementation created for the data object, you must first delete the implementation before you can delete the data object interface.
2. Select the data object interface in either the User-Defined Business Objects folder or the User-Defined Data Objects folder.
3. From the pop-up menu of the data object interface, select **Delete**. The interface is deleted from both folders.

#### **RELATED CONCEPTS**

Data object (*Programming Guide*)

#### **RELATED TASKS**

"Working with data objects" on page 795

## **Deleting a data object implementation**

To delete a data object implementation, follow these steps:

1. Select the data object implementation in either the User-Defined Business Objects folder or the User-Defined Data Objects folder.

2. From the pop-up menu of the data object implementation, select **Delete**.

**Note:** If the data object implementation has a persistent object and schema associated with it, these objects no longer appear in these folders. However, the persistent object and schema are not deleted, and still exist in the DBA-Defined Schemas folder.

#### **RELATED CONCEPTS**

Data object (*Programming Guide*)

#### **RELATED TASKS**

“Working with data objects” on page 795

---

## **Working with keys**

Keys are defined in the User-Defined Business Objects folder, where you can add them from the pop-up menu of a business object interface.

The key provides a way for the client to locate a specific instance of a component on the server.

You can add multiple keys to a business object interface, but each component you configure can only have one key.

The following tasks deal with keys:

- “Adding a key” on page 826
- “Editing a key” on page 828
- “Deleting a key” on page 829

**Note the following points when you select attributes for the key:**

- All business object attributes that you select as primary key attributes will become read-only, even if they were read-write attributes before.
- After you either edit a key by removing some of its attributes, or delete a key, the business object attributes that you had selected as key attributes when you defined the key remain read-only attributes of the business object. You can change their state to read-write using the Attributes page of the Business Object Interface wizard as long as these attributes do not compose any other key of the business object.

#### **RELATED CONCEPTS**

Key (*Programming Guide*)

#### **RELATED TASKS**

“Working with components” on page 697

## RELATED REFERENCES

“Naming objects” on page 128

“Internationalization of data” on page 132

## Adding a key

► CHANGED

Each component must include a primary key class that contains enough information to uniquely identify the component. The key is used when new instances of the component are created or when existing instances need to be found.

Object Builder supports two types of primary keys:

- Those consisting strictly of attributes from a business object client interface (according to the traditional Component Broker Programming Model).
- Those consisting of attributes from a single business object server implementation and, optionally, from the business object client interface (according to the traditional EJB Programming Model).

To add a key, follow these steps:

1. From the User-Defined Business Objects folder, select your business object interface (for example, CarPolicy).
2. From the object’s pop-up menu, select **Add Key**. The Key wizard opens to the Name and Key Attributes page.
3. Appropriate key names are filled in for you (the business object file name and interface name plus Key: for example, CPFile::CarPolicy gets a key named CPFileKey::CarPolicyKey). You can accept these defaults or replace them with your own names.

Select the business object attributes that make up the primary key. If the business object has a parent interface, you can also select from the parent interface’s attributes (you should **not** select attributes of the parent interface if you are planning to inherit from the parent interface’s key).

### Note the following points about key attributes:

- Attributes that you select for the primary key will become read-only, even if you had defined them as read-write attributes when you created the business object.
  - If you have an attribute in the key that is either an unbounded string (a string whose size is not specified), or a bounded string with length in excess of 4000 characters, and you are creating a persistent object and schema for the data object, Object Builder does not provide a mapping helper for the mapping of that attribute as the string does not contain the proper size information. You will have to provide one on the Attributes Mapping page of that wizard.
4. Click **Next**. The Implementation Inheritance page opens.



On this page, you can specify the type of key (primary or unique), and inherit from the appropriate parent class (IPrimaryKey or IUniqueKey) .

If the key has a parent, you can specify it here.

**Note:** You should not inherit from a parent key if you also selected inherited attributes on the previous page.

5. Click **Next**. The Summary of Framework Methods page opens. This page summarizes the framework methods this object implements. No action is needed.
6. Click **Next**. The Optional Framework Methods page opens. Select any additional framework methods you want to implement. Object Builder will add signatures for the methods you select, but you must provide your own implementation code. The methods you implement will override the equivalent framework methods of the parent class.  
**Note:** The Source pane will not allow you to edit these methods until you set them as editable in the Method Implementation wizard. To set a method as editable, follow these steps:
  - a. In the Methods pane, select the framework method.
  - b. From its pop-up menu, click **Properties**.
  - c. In the Method Implementation wizard, specify that you want to use the Source pane.
7. Click **Finish**. The key appears in the User-Defined Business Objects folder, under your business object interface.

If the key consists of one or more attributes from a business object implementation, Object Builder establishes this primary key as the one 'used by' the business object implementation, so long as the business object implementation is not already using another primary key.

In the Methods pane, you should see some items listed in the Framework Methods folder. Default implementation code is provided for these methods, which you can view in the Source pane by selecting a method. Normally, you will not want to edit this code (except for the code for the optional framework methods, as noted above). The code for framework methods is read-only by default.

#### **RELATED CONCEPTS**

Key (*Programming Guide*)

#### **RELATED TASKS**

"Working with keys" on page 825

"Adding a copy helper" on page 830

#### RELATED REFERENCES

Keys for enterprise beans

“Internationalization of data” on page 132

“Naming objects” on page 128

## Editing a key

▶ CHANGED

Keys are defined in the User-Defined Business Objects folder, where they are shown under the business object interface they were added to. You can edit a key by following these steps:

1. From the pop-up menu of the key, click **Properties**. The Key wizard opens to the Name and Key Attributes page.
2. Click the title to select a page to turn to.
3. Change your selections as necessary.
4. Click **Finish** to apply your changes.

**Note:** After you edit a key by removing some of its attributes, the business object attributes that you had selected as key attributes when you defined the key remain read-only attributes of the business object. You can change their state to read-write using the Attributes page of the Business Object Interface wizard as long as these attributes do not compose any other key of the business object.

If the key you edited was already used by a business object implementation, Object Builder determines if the key is still viable for the business object implementation. If the revised primary key consists of attributes from a different business object implementation than the one using it, Object Builder warns you that the relationship between the business object implementation that was originally using it, and the primary key will be dismantled. You then have the option of proceeding, or going back to the wizard to make further changes.

#### RELATED CONCEPTS

Key (*Programming Guide*)

#### RELATED TASKS

“Working with keys” on page 825

#### RELATED REFERENCES

Keys for enterprise beans

“Internationalization of data” on page 132

“Naming objects” on page 128

## Deleting a key

To delete a key, follow these steps:

1. Remove the key from any business object implementations that are configured with it.
2. Remove the key from any data object implementations that are configured with it.
3. Remove the key from any managed object configuration that uses it.
4. Locate the key in the User-Defined Business Objects folder.
5. From the key's pop-up menu, click **Delete**.

**Note:** After you delete a key, the business object attributes that you had selected as key attributes when you defined the key remain read-only attributes of the business object. You can change their state to read-write using the Attributes page of the Business Object Interface wizard as long as these attributes do not compose any other key of the business object.

### RELATED CONCEPTS

Key (*Programming Guide*)

### RELATED TASKS

“Working with keys” on page 825

“Editing a business object implementation” on page 792

“Editing a data object implementation” on page 819

“Editing a managed object configuration” on page 874

---

## Working with copy helpers

Copy helpers are defined in the User-Defined Business Objects folder, where they are shown below the business object interface they were added to.

The copy helper is an optional object that provides a way to initialize multiple attributes of a component instance with a single call to the server.

You can add multiple copy helpers to a business object interface, but each component you configure can only have one copy helper.

The following tasks deal with copy helpers:

- “Adding a copy helper” on page 830
- “Editing a copy helper” on page 831
- “Deleting a copy helper” on page 832

### RELATED CONCEPTS

Copy helper (*Programming Guide*)

#### RELATED TASKS

“Working with components” on page 697

#### RELATED REFERENCES

“Naming objects” on page 128

“Internationalization of data” on page 132

## Adding a copy helper



The copy helper is an optional component object that lets you initialize the attributes of a new component on the server with a single call. It embodies the business object attributes that you will want to initialize.

Object Builder supports two types of copy helpers:

- Those consisting strictly of attributes from a business object client interface (according to the traditional Component Broker Programming Model).
- Those consisting of attributes from a single business object server implementation and, optionally, from the business object client interface (according to the traditional EJB Programming Model).

To add a copy helper, follow these steps:

1. From the User-Defined Business Objects folder, select your business object interface (for example, CarPolicy).
2. From the object’s pop-up menu, select **Add Copy Helper**. The Copy Helper wizard opens to the Name and Attributes page.
3. Appropriate copy helper names are filled in for you (the business object file name and interface name plus Copy: for example, CPFile::CarPolicy gets a copy helper named CPFileCopy::CarPolicyCopy). You can accept these defaults or replace them with your own names.
4. Select which business object attributes to externalize in the copy helper. If the business object has a parent interface, you can also select from the parent interface’s attributes (you must not select attributes of the parent interface if you are planning to inherit from the parent interface’s copy helper).

**Restriction:** CORBA complex types such as *any* and *wstring*, and typedefs, structures, and unions, which are defined as constructs cannot be used as copy helper attributes if either OS/390 are selected as the platform constraints (**Platform > Constrain**). If the business object has such attributes, they will not appear in the **Business Object Attributes** list.

5. Click **Next**. The Implementation Inheritance page appears.

If the copy helper is for a component with a parent, you can select to inherit from the parent component’s copy helper. You should **not** inherit from a parent copy helper if you also selected inherited attributes on the previous page.

If you are not inheriting from a parent copy helper, then you can accept the default `IManagedLocal::IManagedLocal::INonManageable`.

If the data object's environment is **BOIM with UUID key** (page 250), the copy helper should inherit from

`IManagedAdvancedServer::IUUIDCopyHelperBase`. The copy helper will only be usable by other components on the server: client applications should not create UUID components on the server.

6. Click **Next**. The Summary of Framework Methods page opens. This page summarizes the framework methods this object implements. No action is needed.
7. Click **Finish**. The key appears in the User-Defined Business Objects folder, under your business object interface.

If the copy helper consists of one or more attributes from a business object implementation, Object Builder establishes this copy helper as the one 'used by' the business object implementation, so long as the business object implementation is not already using another copy helper.

In the Methods pane, you should see some items listed in the Framework Methods folder. Default implementation code is provided for these methods, which you can view in the Source pane by selecting a method. By default, this code is read-only.

#### **RELATED CONCEPTS**

Copy helper (*Programming Guide*)

#### **RELATED TASKS**

"Working with copy helpers" on page 829

"Adding a business object implementation and data object interface" on page 780

#### **RELATED REFERENCES**

"Environment" on page 249

"Internationalization of data" on page 132

"Naming objects" on page 128

## **Editing a copy helper**

Copy helpers are defined in the User-Defined Business Objects folder, where they are shown under the business object interface they were added to. You can edit a copy helper by following these steps:

1. From the pop-up menu of the copy helper, click **Properties**. The Copy Helper wizard opens to the Name and Attributes.
2. Click the page title to select a page to turn to.
3. Change your selections as necessary.

4. Click **Finish** to apply your changes.

**RELATED CONCEPTS**

Copy helper (*Programming Guide*)

**RELATED TASKS**

“Working with copy helpers” on page 829

**RELATED REFERENCES**

“Internationalization of data” on page 132

“Naming objects” on page 128

## Deleting a copy helper

To delete a copy helper, follow these steps:

1. Remove the copy helper from any data object implementations that are configured with it.
2. Remove the copy helper from any managed object configurations that use it.
3. From the pop-up menu of the copy helper, click **Delete**.

**RELATED CONCEPTS**

Copy helper (*Programming Guide*)

**RELATED TASKS**

“Working with copy helpers” on page 829

“Editing a data object implementation” on page 819

“Editing a managed object configuration” on page 874

---

## Working with DB persistent objects

DB persistent objects are defined in the DBA-Defined Schemas folder. A DB schema is created when you import an SQL DDL file into Object Builder. You can create multiple DB persistent objects for every DB schema. You can also create multiple data objects from the DB persistent object. Further, you can also associate a DB persistent object with a data object implementation, or create a DB persistent object and its associated schema from the implementation.

The following tasks deal with DB persistent objects:

- “Adding a persistent object and schema” on page 833
- “Adding a persistent object from a DB schema” on page 837
- “Mapping a data object to a DB persistent object” on page 703
- “Editing a DB persistent object” on page 838

- “Customizing persistent object ESQL framework methods” on page 766
- “Deleting a DB persistent object” on page 839

#### RELATED CONCEPTS

Persistent object (*Programming Guide*)

Schema (*Programming Guide*)

“DDL” on page 137

#### RELATED TASKS

“Working with components” on page 697

#### RELATED REFERENCES

“Naming objects” on page 128

“Internationalization of data” on page 132

## Adding a persistent object and schema

►CHANGED

Once you have added a data object implementation to the data object interface, you can either create schemas and persistent objects for the implementation, or map existing persistent objects and schemas to the implementation.

**Note:** Before you set about the task of creating the schema and persistent object, if you want either the column names in the schema, or the schema name itself to be enclosed in quotation marks, make sure that these options are set on the Tasks and Objects page, which you can access from **File > Preferences**.

To create a schema and a persistent object, follow these steps:

1. From the User-Defined Data Objects folder, select the data object implementation for which you want to create the persistent object.
2. From the data object implementation’s pop-up menu, select **Add Persistent Object and Schema**. The Add Persistent Object and Schema wizard opens to the Names page.
  - a. Type a name for the group.

**Note:** Group names can contain only alphanumeric characters, the blank space, and the underscore, and they are case-sensitive.

- b. Type a name for the database file.

**Note the following points about database names:**

- They must not exceed eight characters.
- They can contain any of the following characters: the letters a-z and A-Z, 0-9, #, @, \$.

- The first character of the name must be an alphabetic character, or one of #, @, or \$. They must not contain characters from European or Asian character sets (for example, umlauts are not allowed).
- They are not case-sensitive.

**Restriction:** A given transaction cannot access more than one Informix database per CB server. To involve two Informix databases in a transaction, you must access each database from a different server.

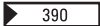
- Type the table name, or accept the default.
- Type the user name.
- Type a name for the schema file, or accept the default.

Follow these rules when you name a DB schema:

- The name must not exceed 18 characters for DB2; 30 characters for Oracle.
  - All alphanumeric characters from your database character set and the characters `_`, `$`, `#`, `@` are allowed. Characters include those from DBCS or European sets (including umlauts).
  - There is no case-sensitivity for names containing these characters, unless they are surrounded by double quotation marks.
  - Non-alphanumeric names must be enclosed in double-quotes, and their case is maintained internally.
- If you had selected **Embedded SQL** as the type of persistence, you must type a name in the **Package File** field, or accept the default.

**Note:** The name of the package file must not exceed eight characters. It must be unique for each of the persistent objects that you create, if they are to operate under the same server at run time.

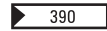
You can either type the names for the persistent object class and instance, or accept the default names.

 **390** If OS/390 is one of the deployment platforms for the data object implementation, the persistent object class name must not exceed eight characters. Object Builder validates the length of the persistent object class when you create a persistent object from a data object implementation, but if you change the deployment platform after you have created the persistent object, be sure that you follow the rule. If not, Object Builder will truncate the name to the 8.3 format. This may result in two persistent object file names becoming identical after truncation, since Object Builder assumes the object's file name to be the same as the persistent object class name.

**Note:** A schema must have a database key specified.



- g. If you had selected Cache Service on the Behavior page of the data object implementation, you can specify one of the Cache Service options: **DB2 Cache Service**, **Oracle Cache Service**, or **Informix Cache Service**.



**Cache Service** is not available when the target platform is OS/390.

**Restriction:** If you are using the Informix Cache Service, a given transaction will not be able to access more than one Informix database per CB server. To involve two Informix databases in a transaction, you must access each database from a different server.

3. Click **Next**. The Attributes Mapping page opens. Here, you can map attributes of the data object to those of the persistent object. You can also change the names of the attributes of the persistent object and the corresponding columns of the schema, and their data types, and specify the persistent object keys if necessary, and the database keys for the table.

**Note the following points:**

- The persistent object attribute name must not exceed 26 characters in length.
  - If you have an attribute in the key that is either an unbounded string (a string whose size is not specified), or a bounded string with length in excess of 4000 characters, and you are creating a persistent object and schema for the data object, Object Builder does not provide a mapping helper for the mapping of that attribute as the string does not contain the proper size information. You will have to provide one on the Attributes Mapping page.
4. Click **Next**. The Columns and Attributes page opens. Use this page to view the mapping between the schema and the persistent object.
5. Click **Next**. The Comments page opens. Use it to save any comments about the schema, any of the schema columns, or the persistent object.

The persistent object is automatically associated with the data object implementation: the persistent object instance is added to the folder on the Associated Persistent Objects page. Object Builder provides the default mapping of both attributes and methods of the data object to the persistent object. You can change the default mappings.

The persistent object appears as a node beneath the data object implementation, and the schema appears as a node beneath the persistent object in both the User-Defined Business Objects folder and the User-Defined Data Objects folder. In the DBA-Defined Schemas folder, the schema exists (with its persistent object) in the schema group you named.

**Restriction:** Even if there is a key defined for a business object and it is designated as a foreign key, when you create a persistent object and schema

for a business object referenced by the other object, it will not automatically create a foreign key in the schema. That is, the FOREIGN KEY constraint is not created in the table's .sql description file.

**Note:** In some RDBMS configurations, the .sql files that Object Builder generates from the schemas must be processed by a database administrator using a design tool such as Logic Works' ERWin version 3.5 or 3.0, before they can be used to create tables in the database catalog. In others, you may be able to bypass the design tool, and instead use command line or other procedures to populate the database catalog.

**Example:**

- In the DB2 NT 5.0 single-user environment, you can use the following sequence of commands from the DB2 command window:  
db2 connect to <name of working database>  
db2 -t -f <full path name of SQL file>
- In Oracle 8.0.4.0 script center, you can import the .sql files.

ERWin 3.0 does not support the following database systems that ERWin 3.5 supports:

- DB2 / 390 5
- DB2 / CS 2
- DB2 / UDB 5
- Oracle 8.x

If you are using ERWin 3.0 or 3.5 to generate SQL files to be imported into Object Builder, you cannot use the default options provided by ERWin for the Oracle DBMS. In ERWin, when you select **Tasks > Forward Engineer/Schema Generation**, you must change the referential integrity options for the primary key and foreign key to use the CREATE statements instead of the ALTER statements.

**RELATED CONCEPTS**

Persistent object (*Programming Guide*)

Schema (*Programming Guide*)

**RELATED TASKS**

"Working with DB persistent objects" on page 832

"Working with DB schemas" on page 843

"Adding a data object implementation" on page 807

"Mapping a data object to a DB persistent object" on page 703

"Tutorial: Creating a component for new DB data" on page 50

#### RELATED REFERENCES

“DB2 data type mappings” on page 142

“Oracle data type mappings” on page 146

Informix data type mappings

## Adding a persistent object from a DB schema

▶CHANGED

To add a persistent object to an existing schema, follow these steps:

1. From the DBA-Defined Schemas folder, select the schema to which you want to add a persistent object.
2. From the schema’s pop-up menu, select **Add Persistent Object**. The Add Persistent Object wizard opens to the Name and Attributes page.
3. Type a name for the persistent object in the **Name** field.

▶  If OS/390 is one of the deployment platforms (that is **Platform > Constrain > 390** is selected), the persistent object class name must not exceed eight characters. If the name exceeds eight characters, Object Builder truncates it to the 8.3 format. This may result in two persistent object file names becoming identical after truncation, since Object Builder assumes the object’s file name to be the same as the persistent object class name.

4. Change the setting of the **Table is updatable** check box, if you want. Your selection determines if the schema is read-only or if it can be updated. For schemas, this check box is selected by default; for views, it is not selected.
5. Select **Cache Service** (one of DB2, Oracle, or Informix is available, depending on your schema definition) or **Embedded SQL** to specify the type of persistence for the object.
6. If you select **Embedded SQL**, you must type a name for the package file, or accept the default.
7. Indicate whether a particular schema column is to be mapped to the corresponding persistent object attribute: click the **Mapped** field and select the check box for the column.  
**Restriction:** You must map all schema columns to their corresponding persistent object attributes; otherwise you may get exceptions thrown at run time if you use the Query Service.
8. Modify the name of the persistent object attribute, if required.
9. Specify whether a schema column’s corresponding persistent object attribute is to be the key for the persistent object by selecting the **PO Key** check box for the column.
10. Click **Next**. The Comments page opens. Use it to add any comments about the persistent object.

#### Restrictions:

- If your schema uses Oracle Cache Service, you can import the schema only if the columns are of the NUMBER or VARCHAR2 data types, or any of the IBM DB2 data types. Object Builder will not accept any other Oracle types such as RAW(n), LONG RAW, NCHAR(n), NVARCHAR2, and ROWID. See “Oracle data type mappings” on page 146 for a complete list.
- Even if there is a key defined for a business object and it is designated as a foreign key, when you create a persistent object and schema for a business object referenced by the other object, it will not automatically create a foreign key in the schema. That is, the FOREIGN KEY constraint is not created in the table’s .sql description file.

#### RELATED CONCEPTS

Persistent object (*Programming Guide*)

Schema (*Programming Guide*)

Cache Service (*Advanced Programming Guide*)

Query Service (*Advanced Programming Guide*)

#### RELATED TASKS

“Creating a component for existing DB data” on page 139

“Working with DB persistent objects” on page 832

“Creating a DB schema by importing an SQL file” on page 844

“Adding a persistent object and schema” on page 833

“Adding a data object from a DB persistent object” on page 814

#### RELATED REFERENCES

“DB2 data type mappings” on page 142

“Oracle data type mappings” on page 146

“Internationalization of data” on page 132

“Naming objects” on page 128

## Editing a DB persistent object



To modify a persistent object, follow these steps:

1. Select the persistent object in the Tasks and Objects pane.
2. From its pop-up menu, select **Properties**.
3. The Persistent Object wizard opens to the Persistent Object page. You can rename the persistent object and provide a new name for the package file. In the panel, you can rename the persistent object attribute: double-click in the field, and type in the new name. You can also specify different persistent object attributes as the keys for the object. Click in the **PO Key** field, and select, or clear the check box.

**Restriction:** A persistent object attribute name cannot exceed 26 characters in length.

4. Click **Next** if you want to change any comments about the persistent object.

You can also change properties for a DB persistent object in the DB schema properties window. To open the window, select the DB schema to which the PO is mapped. Select **Properties** from the schema's pop-up menu.

When you select a column from the schema columns table, the table at the bottom of the Schema properties window shows the corresponding attributes from all POs mapped to that column. Select the PO that you want to edit. Then you can edit the following PO properties:

- **Mapped** Select the check box to indicate that the selected PO maps the selected column.
- **Attribute Name** Changes the name of the PO attribute that is mapped to the selected column.
- **Attribute Type** and **Size** Changes the type and size of the attribute mapped to the selected column.
- **PO Key** Select the check box to indicate that the attribute is part of the PO key.

**Note the following points:**

- To ensure that valid code is generated after a rename, use **Generate > All** instead of **Generate > Selected** from the pop-up menu of the object.
- You cannot change the database type of the persistent object.

**RELATED CONCEPTS**

Persistent object (*Programming Guide*)

Schema (*Programming Guide*)

**RELATED TASKS**

"Working with DB persistent objects" on page 832

"Editing a DB schema" on page 855

**RELATED REFERENCES**

"Internationalization of data" on page 132

"Naming objects" on page 128

## Deleting a DB persistent object

To delete a DB persistent object, follow these steps:

1. Select the persistent object from either the User-Defined Business Objects folder, the User-Defined Data Objects folder, or the DBA-Defined Schemas folder.
2. From the pop-up menu of the persistent object, select **Delete**.

If the persistent object is not connected to a data object implementation, it is deleted from the DBA-Defined Schemas folder.

If the persistent object is associated with a data object implementation, the following deletions take place:

- the persistent object and its underlying schema are deleted from the User-Defined Business Objects folder and the User-Defined Data Objects folder.
- the persistent object is deleted from the DBA-Defined Schemas folder

**RELATED CONCEPTS**

Persistent object (*Programming Guide*)

**RELATED TASKS**

“Working with DB persistent objects” on page 832

---

## Working with DB schema groups

All DB schemas in Object Builder exist in schema groups for organizational purposes.

The following tasks deal with schema groups:

- “Creating a DB schema group”
- “Editing a DB schema group” on page 841
- “Deleting a DB schema group” on page 843

**RELATED CONCEPTS**

“DDL” on page 137

Schema (*Programming Guide*)

Schema group (*Programming Guide*)

**RELATED TASKS**

“Working with components” on page 697

“Creating a component for existing DB data” on page 139

**RELATED REFERENCES**

“Naming objects” on page 128

“Internationalization of data” on page 132

## Creating a DB schema group

You can create a schema group in Object Builder in the following ways:

- by specifying the name of the schema group when you add a persistent object and schema for a data object implementation  
See the task “Adding a persistent object and schema” on page 833.

- by specifying the name of the schema group when you create schemas by importing an SQL file into Object Builder  
See the task “Creating a DB schema by importing an SQL file” on page 844.
- by selecting **Add Schema Group** from the pop-up menu of the DBA-Defined Schemas folder

To create an empty schema group, follow these steps:

1. From the pop-up menu of the DBA-Defined Schemas folder, select **Add Schema Group**. The Schema Group wizard opens to the Schema Group Name page.
2. Type a name for the schema group in the **Schema Group Name** field.
3. Type a name of the database to be associated with this schema group in the **Database Name** field.
4. Select the type of the relational database backend for which you are creating the schema group. You can select **DB2**, **Oracle**, or **Informix**.

**Note the following points:**

- Whenever you create a schema group in Object Builder, along with the schema group name, you must specify the name of the database to be associated with the schema group.
- You can import SQL files into any of the existing schema groups.

**RELATED CONCEPTS**

“DDL” on page 137  
 Schema (*Programming Guide*)  
 Schema group (*Programming Guide*)

**RELATED TASKS**

“Working with DB schema groups” on page 840  
 “Creating a component for existing DB data” on page 139

**RELATED REFERENCES**

“Internationalization of data” on page 132  
 “Naming objects” on page 128

## Editing a DB schema group



You can edit a schema group by editing either the properties of the group or its contents.

To edit the properties of a schema group, follow these steps:

1. From the pop-up menu of the schema group, select **Properties**.
2. The Schema Group wizard opens to the Schema Group page.
3. You can rename the group and the database to be associated with the group. The name of the group must be unique.

4. In the **File to Open in Editor** section, indicate the file you want to view when you use the **Open in Editor** option from the pop-up menu of either the schema group, or any schema within the group. You can change it from the source file (the original SQL DDL file that was imported), which is the default, to the generated file, which is the SQL file that Object Builder generates (when you select **Generate** from the pop-up menu of the schema group).

**Note the following points:**

- The editor in which the SQL file is opened will be either Object Builder's default editor, or the one that you specified in the file:

▶ WIN \CBroker\bin\sqledit.bat

▶ AIX sqledit.sh

**Recommendation:** If you would like to use the key stroke recording function with DBCS characters, you must edit this file to invoke an editor that supports key stroke recording with DBCS characters.

- **Generated** is the only option available for schema groups that are created top-down.
- For a schema group, using **Generate > Selected** is the only way to emit a .sql file for the group.
- The generated file exists in the working directory and has the same name as the name of the schema group. If you want to preserve these generated files, you must rename the existing generated file before you select the **Generate** option from the schema group's pop-up menu. This is particularly important if you want to re-import the SQL source file and this file exists in the working directory, and has the same name as the group. You can re-import an SQL file using the Statements to Import page of the Import SQL DDL File wizard.

New and existing schemas within the group will be associated with the new database name.

The following tasks deal with editing the contents of a schema group:

- "Creating a DB schema by importing an SQL file" on page 844
- "Re-importing an SQL file" on page 847
- "Creating a view with the SQL View Editor" on page 850
- "Editing a view with the SQL View Editor" on page 851

**RELATED CONCEPTS**

"DDL" on page 137

Schema (*Programming Guide*)

Schema group (*Programming Guide*)



**RELATED TASKS**

- “Creating a component for existing DB data” on page 139
- “Working with DB schema groups” on page 840
- “Working with DB schemas”

**RELATED REFERENCES**

- “Internationalization of data” on page 132
- “Naming objects” on page 128

## Deleting a DB schema group

To delete a schema group, follow these steps:

1. Delete any schemas belonging to another group that reference schemas within this group.
2. From the DBA-Defined Schemas folder, select the schema group.
3. From the pop-up menu of the schema group, select **Delete**. You get a warning message informing you that if you delete the group, any associated persistent objects that are contained within that group will be deleted as well.
4. To continue with the deletion process, click **Yes**. The entire schema group is removed from the DBA-Defined Schemas folder, and any schemas and their persistent objects that were members of the group are deleted from the User-Defined Business Objects folder and the User-Defined Data Objects folder as well.

**RELATED CONCEPTS**

Schema (*Programming Guide*)

Schema group (*Programming Guide*) Persistent object (*Programming Guide*)

**RELATED TASKS**

- “Working with DB schema groups” on page 840
- “Working with DB schemas”

---

## Working with DB schemas

A schema is a structural and behavioral composition that defines data storage and data access mechanisms within the database. A schema is always related to storage. A persistent object is usually associated with the schema, and provides persistence of the data beyond the execution time of the application that instantiated the object.

A schema can be created based on a data object, or it can be created from the database definitions stored in a DDL file.

The following tasks deal with DB schemas:

- “Adding a persistent object and schema” on page 833

- “Creating a DB schema by importing an SQL file”
- “Creating a view with the SQL View Editor” on page 850
- “Using complex relationships in SQL clauses” on page 289
- “Editing a view with the SQL View Editor” on page 851
- “Editing a view” on page 853
- “Editing a DB schema” on page 855
- “Editing a generated SQL file” on page 857
- “Deleting a DB schema” on page 859

#### **RELATED CONCEPTS**

Schema (*Programming Guide*)

Persistent object (*Programming Guide*)

#### **RELATED TASKS**

“Working with components” on page 697

“Working with DB schema groups” on page 840

“Adding a persistent object from a DB schema” on page 837

#### **RELATED REFERENCES**

“Naming objects” on page 128

“Internationalization of data” on page 132

## **Creating a DB schema by importing an SQL file**

When you import an SQL DDL file, you import the description of the database schema as it is defined in a relational database. Schemas imported from an SQL file exist in Object Builder within a schema group.

The DDL file is an ASCII file containing SQL statements that describe the schema. A schema can have tables. Each row must be uniquely identifiable: each table must have a unique primary key. A row of a table (or of a view) is represented as a single persistent object instance. All rows in the table can be represented by a single persistent object class but each row is a separate instance of that class.

### **Restrictions:**

- This version of Object Builder supports SQL DDL files only from the following database types and versions:

- Oracle:



Oracle 8.0.5 databases are supported only on the Windows NT, AIX and Solaris platforms.

    Oracle 8.1.6 (Oracle 8i Release 2) databases are supported on the Windows NT, AIX, Solaris, and HP-UX platforms.

- DB2: MVS 4.1 databases, and Object Builder tolerates UDB syntax of versions 5.0 and 6.1. This means that you may not be able to import a 5.0 or 6.1 DDL file, but if you have a DDL file containing 4.1 DDL with a few 5.0-, or 6.1-specific lines, you will be able to import the 4.1 DDL lines from that file.
- Informix:  
A given transaction cannot access more than one Informix database per CB server. To involve two Informix databases in a transaction, you must access each database from a different server.

  Informix Dynamic Server version 7.30 files are supported only on AIX and Solaris.

   Informix Dynamic Server version 7.31 files are supported only on Windows NT, AIX and Solaris.

- SQL files larger than 2 MB are not recommended.

To import an existing DDL file, follow these steps:

1. From the pop-up menu of DBA-Defined Schemas folder or the schema group, select **Import SQL**. The Import SQL DDL File wizard opens to the SQL File Selection page.
2. Type the name of the DDL file (.sql file) or click **Find** to specify the path and select from a list of files.  
**Note:** It is recommended that the SQL source file be placed in a directory other than the Working directory. This is to avoid having the file overwritten when you select either **Generate > Selected** or **Generate > All** from the schema group's pop-up menu.
3. Type a name for the database to be associated with the schema being imported, or accept the default in the **Database Name** field.
4. Type a name for the group to contain the schemas being imported or accept the default in the **Group Name** field. The schemas appear in the DBA-Defined Schemas, folder beneath the the group.
5. Click **Next**. The Statements to Import page opens with all the SQL statements in the imported file selected for parsing. To clear all the selections, click **Undo Selection**. You can then select the specific ones you want parsed. Multiple selections are possible. To select all the statements, click **Select All**.

**Note:** At least one CREATE TABLE statement must be selected for the import process to succeed.

**Restriction:** Currently, the only SQL statements supported are DROP, CREATE TABLE, CREATE VIEW, ALTER TABLE, and COMMENT ON. None of

these statements must contain expressions or column functions. The CREATE VIEW statement must contain only a simple query (SELECT statement). Currently there is no support for unnamed columns, expressions, functions, or sub-selects in CREATE VIEW.

6. Click **Finish**. Schemas thus imported into Object Builder, appear in the DBA-Defined Schemas folder, within a group that gets its name from the .sql file.

**Note:** You can re-import a schema group to include different tables, or to modify or delete existing tables. To do so, follow these steps:

1. Select the schema group in the DBA-Defined Schemas folder. From the pop-up menu of the schema group, select Import SQL. The Import SQL DDL File wizard opens to the Statements to Import page.
2. All the SQL statements in the imported file are selected for parsing. To clear all the statements, click **Undo Selection**. You can select the specific ones you want parsed. Multiple selections are possible. To select all the statements, click **Select All**. The schema group is overwritten.

Importing an SQL file is the first step in the bottom-up scenario, when you can reuse existing data. The scenario continues with the following steps:

1. Using the schema information, create a persistent object.
2. Create a data object that corresponds to the persistent object.
3. Create a business object and select the data object (created in step 2) to be used by the business object. The data object uses the mapping information you provide when you select it, to manage the business object's persistent state.

#### **Restrictions:**

- A table that is associated with a schema of one schema group cannot reference a foreign key defined in a table within another schema group.
- Object Builder lets you import schemas for which no primary keys have been defined. However, these schemas can result in exceptions thrown at run time if you use Query Services. To avoid this happening, you can follow any one of these steps:
  1. After you import the SQL file, select **Properties** from the pop-up menu of the schema, and select any of the schema columns as the database key. (Select the **DB Key** check box.)
  2. Before you import the SQL file, edit the source file and add a PRIMARY KEY constraint for at least one of the tables.
  3. Edit the primaryKey entry in the table MappedType.*tablename*\_Table, in the System Management Data Definition Language (SM DDL) file generated from the Application Family. This file's primary name will have the term *Specific* before the family name you specified and its extension will be .ddl. For tables that do not have at least one primary key defined, the primaryKey entry will be void (""). Edit it to include

the names of all the table columns that comprise the primary key you want to define. For example, if you want to indicate that the columns COMP, PLAT, SEQ and ATTEMPT comprise the primary key, this would be the entry: `primaryKey = "\"COMP\", \"PLAT\", \"SEQ\", \"ATTEMPT\"";`

**Note:** Most views are read-only, but you can update some of them. Object Builder emits `insert()`, `update()` and `del()` with empty method bodies on associated persistent objects that have been marked read-only. By default, persistent objects are read-only for views, and updatable for tables, but you can change this setting.

#### RELATED CONCEPTS

“DDL” on page 137  
Schema (*Programming Guide*)  
Persistent object (*Programming Guide*)  
“Special framework methods” on page 758

#### RELATED TASKS

“Creating a component for existing DB data” on page 139  
“Working with DB schemas” on page 843  
“Working with DB persistent objects” on page 832  
“Editing a DB schema group” on page 841  
“Editing special framework methods” on page 758  
“Generating code” on page 551

#### RELATED REFERENCES

“Internationalization of data” on page 132  
“Naming objects” on page 128

## Re-importing an SQL file

To re-import an SQL file, follow these steps:

1. From the pop-up menu of DBA-Defined Schemas folder or the schema group, select **Import SQL**. The Import SQL DDL File wizard opens to the SQL File Selection page.
2. The name of the DDL file (.sql file) previously imported appears in the **Last File Name Imported** field.

**Note:** It is recommended that the SQL source file be placed in a directory other than the subdirectories of the Working directory, which are named according to the platform for which you are generating code. This is to avoid having the file overwritten when you select either **Generate > Selected** or **Generate > All** from the schema group’s pop-up menu.

3. The name of the database previously associated with the schema being imported is shown in the **Database Name** field. This entry cannot be changed.
4. The name of the group shown in the **Group Name** field too cannot be changed. The schemas appear in the DBA-Defined Schemas folder beneath the group.
5. Click **Next**. The Statements to Import page opens with all the SQL statements in the imported file selected for parsing. To deselect all the statements, click **Undo Selection**. You can select the specific ones you want parsed. Multiple selections are possible. To select all the statements, click **Select All**.

**Restriction:** Currently, the only SQL statements supported are DROP, CREATE TABLE, CREATE VIEW, ALTER TABLE, and COMMENT ON. None of these statements must contain expressions or column functions. The CREATE VIEW statement must contain only a simple query (SELECT statement). Currently there is no support for unnamed columns, expressions, functions, or sub-selects in CREATE VIEW.

6. Click **Finish**.

If you use **Import SQL** from the folder, the schemas imported into Object Builder appear in the DBA-Defined Schemas folder, within a group whose default name is the name of the .sql file. You can change the name of the group and the default name of the database as well. If you use **Import SQL** from a schema group, you can neither change the name of the group nor that of the database.

If you select statements that act on existing tables, Object Builder warns you that the tables will be overwritten.

#### **RELATED CONCEPTS**

- "DDL" on page 137
- Schema (*Programming Guide*)
- Schema group (*Programming Guide*)
- "The SQL View Editor" on page 849

#### **RELATED TASKS**

- "Creating a DB schema by importing an SQL file" on page 844
- "Working with DB schemas" on page 843
- "Adding a persistent object from a DB schema" on page 837
- "Editing a DB schema" on page 855
- "Editing a generated SQL file" on page 857

## The SQL View Editor

The SQL View Editor is a tool that enables you to create and modify views from within Object Builder. You can invoke it from the pop-up menu of a schema group in the DBA-Defined Schemas folder when you want to create a view, or from the pop-up menu of a view when you want to edit the view.

The SQL View Editor has the following notebook pages:

- View Properties
- View Work Area
- View Summary

### View Properties

This page enables you to provide identification for the view you are adding. You can review details about the view, and edit any comments you added when you created the view, when you use the Editor to edit a view.

### View Work Area

This page is where most of your interaction with the View Editor takes place. This area is further subdivided into the following panes:

- **Schemas:** This pane lists the schemas that belong to the schema group. In the Graphic view, it also shows foreign key relationships among the schemas.
- **Columns:** This pane lists the columns of the schema that you select in the Schemas pane. You can select these columns for inclusion in SQL clauses.
- **Clauses:** This pane enables you to construct or modify clauses to define the view, with the columns that you select. It is further divided into the following panes for defining the clauses:
  - Selected Columns
  - Where
  - Group By
  - Having

### View Summary

Use this section to view the SQL code for the different clauses you defined using the **Clauses** pane in the **View Work Area**. You can view the code either as a single SQL statement, or clause by clause.

#### RELATED CONCEPTS

Schema (*Programming Guide*)

Schema group (*Programming Guide*)

#### RELATED TASKS

“Creating a view with the SQL View Editor” on page 850

“Editing a view with the SQL View Editor” on page 851  
“Using complex relationships in SQL clauses” on page 289

#### RELATED REFERENCES

Query Service (*Advanced Programming Guide*)

## Working with the SQL View Editor

You can perform the following tasks with the SQL View Editor.

- “Creating a view with the SQL View Editor”
- “Editing a view with the SQL View Editor” on page 851
- “Using complex relationships in SQL clauses” on page 289

#### RELATED CONCEPTS

Schema (*Programming Guide*)

Object relationships (*Programming Guide*)

#### RELATED TASKS

“Creating a component for existing DB data” on page 139

“Working with components” on page 697

“Editing a DB schema group” on page 841

“Storing an object reference as a handle” on page 288

## Creating a view with the SQL View Editor

To create an SQL view in Object Builder, follow these steps:

1. Select the schema group from which you want to create the view, in the DBA-Defined Schemas folder.  
**Note:** The schema group must contain the schemas from which you want to create the view.
2. From its pop-up menu, select **Add SQL View**. The SQL View Editor opens.
3. Click the **View Properties** tab of the editor. On the View Properties page, type a name for the view, and optionally specify a userid and add any comments.
4. Click the **Selected Columns** tab of the Clauses pane.
5. Select a schema to be used for the view in the Schemas pane. The schema columns and their details appear in the Columns pane.
6. Select the columns you require for the view from the Columns pane. As you select each column, the column name and the table it belongs to appear in the Selected Columns page.
7. Repeat steps 4, 5 and 6 for each of the schemas whose columns you want to include in the view.



8. Click the **Where** tab of the Clauses pane. On the Where page, you can specify conditions that have to be met by the various schema columns, for inclusion in the view.
9. Click the **Group By** tab of the Clauses pane. On the Group By page, you can specify the columns, based on whose values the order of occurrence of the view's rows is determined: Click the **Select All** button. All columns that you specified as selected columns on the Selected Columns page are set as the grouping columns. The first grouping column determines the initial grouping. Subsequent grouping columns are used to resolve the order of the rows when there is a tie within a group of rows formed by its predecessor.  
**Note:** You cannot select a subset of the selected columns to group the view by. You have to use the **Select All** button.
10. Click the **Having** tab of the Clauses pane. On the Having page, you can apply a qualifying condition to the groups created with the `<span style="text-transform: uppercase">GROUP BY</span>` clause. Only those groups that meet the HAVING condition are included in the view.  
**Note:** From the Where page and the Having page, you can bring up the Organize Logical Combination dialog box, where you can manually arrange the predicates to be combined for the view.
11. Click the **View Summary** tab to open the View Summary page of the Clauses pane. Use this page to view the SQL clauses you defined for the view.
12. When you have finished reviewing your definition, click **Finish**.

The view appears in the schema group in the DBA-Defined Schemas folder.

#### RELATED CONCEPTS

Schema (*Programming Guide*)

Schema group (*Programming Guide*)

#### RELATED TASKS

"Working with DB schemas" on page 843

"Working with the SQL View Editor" on page 850

"Creating a DB schema group" on page 840

"Editing a view with the SQL View Editor"

#### RELATED REFERENCES

"Internationalization of data" on page 132

"Naming objects" on page 128

## Editing a view with the SQL View Editor

To edit an SQL view in Object Builder, follow these steps:

1. Select the view in the DBA-Defined Schemas folder.

2. From its pop-up menu, select **SQL View Editor**. The SQL View Editor opens.
3. Click the **View Properties** tab. You cannot change the name and user of the view. However, you can modify the comments on the View Properties page.
4. Click **Next**, or click the **View Work Area** tab. Columns of the view appear in the Selected Columns page of the Clauses pane. As you select the different schemas in the Schemas pane, their schema columns and corresponding details appear in the Columns pane. To add a new column to the view, click on it in the Columns pane. To remove a column from the view, right-click on any field in column's row in the Clauses pane, and select **Remove** from the pop-up menu. To rename a view column, right-click on the name in the **View Column** of the Selected Columns page, and select **Change Value** from the pop-up menu. The Change Column Name dialog box appears, and you can type a new name in the field.

**Note:** On all the other pages of this pane, you can view the previous settings and make changes if you want. To remove an entry in a field on either the Where page or the Having page, right click in the field and select **Remove**. To specify a new entry for the Table/Column field in the predicate section, select the column from the Columns pane; to specify a new entry in the search conditions' Table/Column section, first click in the field, and then, select a column from the Columns pane.

5. Click the **Where** tab of the Clauses pane. On the Where page, you can view the conditions that were previously set for the various schema columns, for inclusion in the view.
6. Click the **Group By** tab of the Clauses pane. On the Group By page, if no GROUP BY clause had been specified for the view, you can specify the columns, based on whose values the order of occurrence of the view's rows is determined: Click the **Select All** button. All columns that you specified as selected columns on the Selected Columns page are set as the grouping columns. If the view's previous definition included a GROUP BY clause, the only modification you can make on this page is to deselect all grouping columns: click on **Clear All**. The view will not have an orderly grouping for its rows.
7. Click the **Having** tab of the Clauses pane. On the Having page, you can apply a qualifying condition to the groups created with the GROUP BY clause. Only those groups that meet the HAVING condition are included in the view.

**Note:** From the Where page and the Having page, you can bring up the Organize Logical Combination dialog box, where you can manually arrange the predicates to be combined for the view.

8. Click the **View Summary** tab to open the View Summary page of the Clauses pane. Use this page to view the SQL clauses you redefined for the view.
9. When you have finished reviewing your definition, click **Finish**.

The view will be redefined according to your modifications.

#### **RELATED CONCEPTS**

Schema (*Programming Guide*)

Schema group (*Programming Guide*)

#### **RELATED TASKS**

“Working with the SQL View Editor” on page 850

“Creating a DB schema group” on page 840

“Creating a view with the SQL View Editor” on page 850

#### **RELATED REFERENCES**

“Internationalization of data” on page 132

“Naming objects” on page 128

## **Editing a view**

You can modify a view that exists in Object Builder using the Schema page of the Schema wizard the same way you modify a schema in Object Builder.

To change the identification for a view, follow these steps:

1. In the DBA-Defined Schemas folder, select the view. From the pop-up menu of the view, select **Properties**. The Schema page opens.
2. You can modify the user ID, table name and schema filename.
3. Click **Finish**.

A new, stand-alone view is created only when the old view is referenced under another view. If the view is not under another view, it is renamed. So, even a view that has associated persistent objects will be renamed, not copied. A copy of the view, with the new name and properties is created only for views that are under other views.

**Note:** To ensure that valid code is generated after a rename, use **Generate > All** instead of **Generate > Selected** from the pop-up menu of the object.

You can modify the structure of a view by editing the Schema page, and the Clause Summary page.

To change the structure of a view using the Schema page, follow these steps:

1. From the pop-up menu of the view object, select **Properties**. The Schema page opens.

2. You can edit the **ForBitData**, **DB Key** and **Not Null** fields. The existing view is overwritten with the changes.

To change the structure of a view using the Clause Summary page, follow these steps:

1. From the pop-up menu of the view object, select **Properties**. The Schema wizard opens to the Schema page. You can edit the **ForBitData**, **DB Key** and **Not Null** fields.
2. Click the arrow to the left of the page name, and select Clause Summary page from the list. By default, the text panel on this page is read-only, and you can select the radio buttons associated with the different SQL clauses to see their definition.

**Attention:** It is not recommended that you edit the Object Builder-generated SQL clauses for the view definition. The changes you make affect the DDL that Object Builder generates and you can access the original code only by redefining the view, or importing once again into Object Builder the SQL file that contains the definition of the view. However, if you must edit some of the view's definition clauses, follow step 3; otherwise proceed with step 4.

3. Select the **Provide your own SQL for the clause** check box. Once you select this box, the text panel containing the clauses becomes editable, and you can overwrite the definition provided by Object Builder. You can overwrite one clause at a time.
4. Turn to the Comments page. Here, you can type comments for the view, as well as for the schema columns that are used in the definition of the view.
5. Click **Finish**.

The existing view is overwritten with the changes.

**Note:** To delete a view that has an associated persistent object, you must first delete the persistent object. To delete a view that is used to create other views, you must first delete the view that is created from it.

#### **RELATED CONCEPTS**

Schema (*Programming Guide*)

Persistent object (*Programming Guide*)

#### **RELATED TASKS**

"Working with DB schemas" on page 843

"Adding a persistent object and schema" on page 833

"Adding a persistent object from a DB schema" on page 837

"Editing a DB schema group" on page 841

#### RELATED REFERENCES

“Internationalization of data” on page 132

“Naming objects” on page 128

## Editing a DB schema

▶ CHANGED

You can modify any DB schema that exists in Object Builder, whether it was created from a data object or imported from an SQL DDL file.

To make changes to the DB schema, select **Properties** from its pop-up menu. The Schema page opens. On the schema page, you can make changes to the following properties:

- Identification (page 855)
- Column properties (page 855)
- Schema structure (page 856)

You can also change persistent object attribute mapping properties on this page.

Changes you make here will be propagated up through the persistent objects and the data object implementation to persistent object mappings for the schema. Check the resulting mapping changes to ensure they are what you expect.

### Identification

Use the **Schema Name**, **Table**, and **Schema File** fields to change what the schema is called. You cannot modify the schema access type (the database type of the schema).

Click **Next** to go to the Comments page, where you can edit comments about the schema.

In the case of a view, you can also modify the clauses that define the view (Clause Summary page). When you edit the view definition, a new, stand-alone schema is created only if the old schema is referenced under another view. If the schema is not referenced by another view, the view you are editing is simply renamed. So, even a schema that has foreign key relationships or associated persistent objects will be renamed, not copied. A copy of the schema, with a new name and new properties is created only for schemas that are referenced under other views.

**Note:** To ensure that valid code is generated after a rename, use **Generate > All** instead of **Generate > Selected** from the pop-up menu of the object.

## Column properties

Each column in the schema is shown in the table, along with the properties for that schema. To edit a column property, double-click on the corresponding field.

- **Column Name** Changes the name of the column in the schema.
- **SQL Type** Changes the SQL type associated with the column. If you change the column type to a type for which Object Builder “understands” how to properly map (for example, string to varchar), a reasonable mapping will be propagated to the PO and DOImpl. Otherwise, the previous mapping will be kept (which may result in unexpected run-time behavior). Check the propagated type mapping in case you want to change it.
- **Length** and **Scale** For numeric types such as DECIMAL, changes the length and scale of the type.
- **ForBitData** Select the check box if the schema column is used to store an object reference. Object references are stored as strings.
- **DB Key** Select the check box to specify this column as being part of the database key.
- **Not Null** Select the check box to indicate that this column is not nullable.
- **Foreign key** fields

**Note:** All columns that you indicate as DB keys have to be “not null”. This specification cannot be changed.

## Schema structure

While you may modify schema structure by re-importing the SQL file, by doing so, you will lose changes you have made. Instead, you can make schema structure modifications in the Schema properties window. The existing schema is overwritten with the changes.

To add a column:

1. Select **Add Schema Column** from the table’s pop-up menu. A column with some default values will be added to the table.
2. Edit the values as appropriate.

To delete a column:

1. Select the column in the table.
2. Select **Delete Schema Column** from the pop-up menu.

The column is not actually deleted until you exit the properties window. Instead, a red X will appear beside the column, indicating your intention to delete it. If you decide to keep it, select **Undo Delete** from its pop-up menu.

**Note:** If you choose to change the structure of a schema by re-importing the SQL file, follow these steps:

1. From the pop-up menu of the schema group, select **Import SQL**. The Statements to Import page opens.
2. Select those ALTER TABLE statements that refer to the schema that you want to modify in this group.

**Warning:** Do not re-import CREATE statements.

These changes will be propagated up through the POs and DOImpl-PO mappings, using default mappings which you may want to confirm before generating.

#### RELATED CONCEPTS

Schema (*Programming Guide*)

Persistent object (*Programming Guide*)

#### RELATED TASKS

“Working with DB schemas” on page 843

“Adding a persistent object and schema” on page 833

“Adding a persistent object from a DB schema” on page 837

“Editing a DB schema group” on page 841

“Editing a DB persistent object” on page 838

#### RELATED REFERENCES

“Internationalization of data” on page 132

“Naming objects” on page 128

## Editing a generated SQL file

▶CHANGED

When you generate a schema that is created from a data object, using either **Generate > Selected** or **Generate > All** from the pop-up menu of the schema, or **Generate > All** from the pop-up menu of its containing schema group, the resulting .sql file cannot be used as such by DB2 to create tables.

### Note the following points:

- In some RDBMS configurations, the .sql files that Object Builder generates from the schemas must be processed by a database administrator using a design tool such as Logic Works' ERWin version 3.5 or 3.0, before they can be used to create tables in the database catalog. In others, you may be able to bypass the design tool, and instead use command line or other procedures to populate the database catalog.

#### Example:

In the DB2 NT 5.0 single-user environment, you can use the following sequence of commands from the DB2 command window:

```
db2 connect to "name of working database"
```

```
db2 -t -f "SQL filename with the path"
```

In Oracle 8.0.4.0 script center, you can import the .sql files.

- ERWin 3.0 does not support the following database systems that ERWin 3.5 supports:
  - DB2 / 390 5
  - DB2 / CS 2
  - DB2 / UDB 5
  - Oracle 8.x

If you are using ERWin 3.0 or 3.5 to generate SQL files to be imported into Object Builder, you cannot use the default options provided by ERWin for the Oracle DBMS. In ERWin, when you select **Tasks > Forward Engineer/Schema Generation**, you must change the Referential Integrity Options for the Primary Key and Foreign Key to use the CREATE statements instead of the ALTER statements.

- If the design tool you use includes a startup command that takes a DDL file as an argument, you can place that command in the file named sqlparse.cmd (on NT) and sqlparse.sh (on AIX). The **Open in Editor** option from the pop-up menu of the schema, would then launch the tool.
- The SQL DDL files that you create using a database design tool can be imported into Object Builder.

To launch ERWin from Object Builder, follow these steps:

1. Open the file sqledit.bat (which is located in the bin subdirectory of Component Broker tools installation directory, which is usually \CBroker\bin in your current drive).
2. Comment out (add rem before) the command that launches the LPEX editor (or, any other editor of your choice, if you had earlier modified this statement):
 

```
rem @start evfxlxpm %sqleditargs%
```
3. Delete rem, which is at the beginning of the command that launches ERWin
 

```
@start mmopn32 %sqleditargs%
```

The sqledit.bat file should have the following entries:

```
rem @start evfxlxpm %sqleditargs%
@start mmopn32 %sqleditargs%
```
4. From the DBA-Defined Schemas folder or the User-Defined Data Objects folder, select the schema.
5. From the pop-up menu of the schema, select **Open in Editor**. This launches ERWin .

 ERWin does not run on AIX, but if you still want to use it as the design tool, follow these steps:


1. Transfer the generated .sql file from AIX to NT.



2. Process the .sql file using ERWin against an NT DB2 client installation, which is backed by an AIX DB2 server.

With AIX, too you can specify the editor of your choice. This is done in the file `sqledit.sh`.

1. Comment out (add # before) the command line that launches the vi editor  
# `dtterm -e vi $* &`
2. Include the editor of your choice instead of vi (for example, LPEx: `lpex $* &`)

 ERWin runs only on Windows NT.

**Note:** LPEx is available only if SDE/6000 is installed.

The `sqledit.sh` file should have the following entries:

```
dtterm -e vi $* &
```

```
lpex $* &
```

**Hint:** When you use ERWin to create table columns that will hold object references, you can specify the VARCHAR FOR BIT DATA type.

#### RELATED CONCEPTS

Schema group (*Programming Guide*)  
Object relationships

#### RELATED TASKS

- “Creating a component for existing DB data” on page 139
- “Working with DB schemas” on page 843
- “Editing a DB schema group” on page 841
- “Storing an object reference as a handle” on page 288

## Deleting a DB schema

To delete a DB schema, follow these steps:

1. Delete any persistent objects that are associated with this DB schema.

**Note:** If you delete the persistent objects from the the User-Defined Data Objects folder, or the User-Defined Business Objects folder, the schemas are automatically deleted from these folders. You still have to delete it from the DBA-Defined Schemas folder. (Follow step 4.)

2. Delete any views that use this schema.
3. Delete any other schemas that reference this schema.

4. From the pop-up menu of the schema in the DBA-Defined Schemas folder, select **Delete**.

#### **RELATED CONCEPTS**

Schema (*Programming Guide*)

Persistent object (*Programming Guide*)

#### **RELATED TASKS**

“Working with DB schemas” on page 843

“Deleting a DB persistent object” on page 839

---

## **Working with PA persistent objects**

PA persistent objects are defined in the User-Defined PA Schemas folder. A PA persistent object is created along with every PA schema that is created in Object Builder, as a result of importing a PA bean.

You can create additional PA persistent objects for every PA schema. You can also create multiple data objects from the PA persistent object. Further, you can also associate a PA persistent object with a data object implementation.

The following tasks deal with PA persistent objects:

- “Adding a persistent object from a PA schema”
- “Editing a PA persistent object” on page 861
- “Adding a data object from a PA persistent object” on page 815
- “Mapping a data object to a PA persistent object” on page 708
- “Deleting a PA persistent object” on page 861

#### **RELATED CONCEPTS**

Persistent object (*Programming Guide*)

Schema (*Programming Guide*)

“Procedural adaptor bean (PA bean)” on page 159

#### **RELATED TASKS**

“Working with components” on page 697

## **Adding a persistent object from a PA schema**

To add a persistent object to an existing PA schema, follow these steps:

1. From the User-Defined PA Schemas folder, select the schema to which you want to add a persistent object.
2. From the schema’s pop-up menu, select **Add Persistent Object**. The Add Procedural Adaptor Persistent Object wizard opens to the Attributes Mapping page.

3. Type a name for the persistent object in the **Name** field.
4. Modify the names of the persistent object attributes, if required.

#### **RELATED CONCEPTS**

Persistent object (*Programming Guide*)

Schema (*Programming Guide*)

#### **RELATED TASKS**

“Creating a component for PA data” on page 157

“Working with PA persistent objects” on page 860

“Working with PA schemas” on page 862

“Adding a data object from a PA persistent object” on page 815

### **Editing a PA persistent object**

In this release of Object Builder, you cannot edit a PA persistent object that was created for you along with the PA bean that you imported into Object Builder.

However, you can add another persistent object to the schema (from the pop-up menu of the PA schema, select **Add Persistent Object**), and you can change the names of the attributes, if you want to. Once the persistent object is created, it is not editable.

#### **RELATED CONCEPTS**

Persistent object (*Programming Guide*)

#### **RELATED TASKS**

“Working with PA persistent objects” on page 860

### **Deleting a PA persistent object**

To delete a PA persistent object, follow these steps:

1. Select the persistent object from either the User-Defined Business Objects folder, the User-Defined Data Objects folder, or the User-Defined PA Schemas folder.
2. From the pop-up menu of the persistent object, select **Delete**.

If the persistent object is not connected to a data object implementation, the following deletions take place:

- the persistent object is deleted from the User-Defined PA Schemas folder (the schema is not removed from this folder)

If the persistent object is associated with a data object implementation, the following deletions take place:

- the persistent object and its underlying schema are deleted from the User-Defined Business Objects folder (if the data object implementation is associated with a business object as in the case when you build a model bottom-up) and the User-Defined Data Objects folder.
- the persistent object is deleted from the User-Defined PA Schemas folder (the schema is not removed from this folder)

#### RELATED CONCEPTS

Persistent object (*Programming Guide*)

#### RELATED TASKS

“Working with PA persistent objects” on page 860

---

## Working with PA schemas

You can create a component for existing transactional information by importing the PA bean into Object Builder, and deriving a component from it.

The following tasks deal with PA schemas:

- “Creating a PA schema by importing a PA bean”
- “Adding a persistent object from a PA schema” on page 860
- “Editing a PA schema” on page 868
- “Deleting a PA schema” on page 869

#### RELATED CONCEPTS

Persistent object (*Programming Guide*)

Schema (*Programming Guide*)

“Procedural adaptor bean (PA bean)” on page 159

#### RELATED TASKS

“Working with components” on page 697

## Creating a PA schema by importing a PA bean

► CHANGED

**Suggestion:** It is important to have your CLASSPATH system environment variable always up-to-date with the directory into which you are importing the PA beans. If modifying the class path is a problem - especially if you want to avoid rebooting your system, which is usually required each time you update any system environment variables, in this case the class path, always follow these steps when you import PA beans into Object Builder:

1. Maintain one specific directory into which you always import PA beans.

2. Update your CLASSPATH system environment variable to include this directory path.
3. Whenever you import a PA bean from within Object Builder, place it in this directory (unzip the JAR file into this directory). If you are importing a PA bean by exporting it directly from VisualAge for Java, export it as a directory with the same name as the directory you included in the class path; do not export it as either a DAT file or a JAR file.

**Note:**The suggestion above is recommended only for importing beans; not for deploying them. When you deploy the beans, you must include their JAR files as additional executables for the application (add them on the Additional Executables page of the Application wizard). This ensures that the bean is available to the Component Broker server at run time. The JAR files that are listed as additional executables take precedence over the directory since the JAR files are added to the front of the CLASSPATH when the Component Broker server is started.

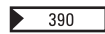
#### **Restrictions:**

- You **cannot** import PA beans that inherit from the BeanInfo superclass in VisualAge for Java.
- When you create the PA bean, ensure that you name its key the same name as the PA bean, with the suffix Key. For example, if the name of the PA bean is AccountPAO, then the name of its key must be AccountPAOKey. Only such beans can be imported into Object Builder.
- The PA beans that you import cannot have arrays as either attribute types, or method parameter types, or method return types. The only types that are supported are those listed in “Java data type mappings” on page 159.
- You cannot import into Object Builder PA beans that have two methods with the same name. You can create such PA beans with overloaded method names (methods with the same name that differ in either the type or number of their parameters, or both) using VisualAge for Java. However, since overloaded method names are not allowed in IDL, Object Builder enforces the IDL restriction.
- Procedural Adapter Object (PAO) beans, which may be created with VisualAge for Java (version 2.0 and later) are supported with this release of Component Broker.

To import a procedural adaptor schema (PA schema), follow these steps:

1. In the Tasks and Objects pane, select the User-Defined PA Schemas folder.
2. From its pop-up menu, select **Import > Bean**. The Import Procedural Adaptor Bean wizard opens to the Bean Selection page.
3. You can import the bean using one of the following methods:
  - a. Specify the name of the EntityProceduralAdapterObject bean class. Follow these steps:

- 1) Select the **Enter bean name** radio button, and type the fully qualified name of the class (package name and class name) in the field. For example, to import the BeCashAcct bean, type `paa.samples.cics.eci.acct.BeCashAcctPAO`.
- b. Specify a JAR file. Follow these steps:
  - 1) Select the **Select a bean from an existing JAR file** radio button. The **Find JAR file** button becomes active, and you can use it to locate the file.
  - 2) Once you have selected the JAR file (for example `BeCashAcct.jar`), the panel lists the classes contained in the file and you can select the one you want imported. The field just below the **Enter bean name** button shows the name of the selected object class. See the notes below this procedure for possible error conditions and their causes.
4. Click **Next**. The Names and Connectors page opens. Here, you can name the module and the persistent object to be associated with the PA schema. You can also select the connector type to be used to access objects. The connector type must match the one you used when you created the procedural adaptor bean.



When you choose OS/390 as the development (target) platform (**Platform > Constrain**), only the EXCI, OTMA, IMS APPC, and Generic connector types are available for selection.

When you select either NT and 390, or AIX and 390 as the development platforms, all the connector types (LU 6.2, HOD, SAP, and ECI, besides the types available for OS/390) are available for selection.

When you select NT and AIX, all the types except those that are specific to OS/390 (EXCI, OTMA, and IMS APPC) are available for selection.

5. Click **Next**. The Key Selection page opens with the properties of the PA schema listed in the **Properties** box.
6. Select the properties that are part of the PA bean's key class from the **Properties** box, and move them to the **Key Attributes** box.
7. Click **Next**. The Variable Type Specification page opens, and for all attributes of *string* type in your imported bean, you can specify whether they correspond to the IDL *string* type, or the IDL *wstring* type. Similarly, for *char* properties in your imported bean, you can specify whether they correspond to the *char* IDL type, or the *wchar* IDL type.
8. Click **Next**. The Method and Parameter Type Specification page opens, and you can specify whether the return type of the methods defined on the PA bean that are of character or string type are either single-byte, or multi-byte. You can also specify the same for the character and string types of these method's parameters.

**Note:** If the bean is queryable, the Query Methods page will be

dynamically added to the wizard. On this page, you can specify what each of the query methods will fetch using a WHERE clause.

9. Click **Finish**. The bean will be imported into Object Builder. The PA schema (for example, BeCashAcctPAO) and its associated persistent object (BeCashAcctPAOPO) will now appear in the Tasks and Objects pane under the User-Defined PA Schemas folder.

**Note the following points:**

Typically, the following types of classes are referenced in the bean that you select for import:

1. The PAA run-time file sompart.zip, and the someab.jar file, and if you are using host on-demand (HOD) connections, somhod20.jar.
2. User-defined classes (any JAR files corresponding to the PA beans you created using Enterprise Access Builder (EAB), and any other classes you defined that are used by the PA bean)

If you get a message about files not being in the class path, follow these steps:

1. Ensure that the PA bean (the EntityProceduralAdapterObject) is in the class path (in the CLASSPATH environment variable).
2. If your PA bean is in a JAR file, ensure that the JAR file is in the class path.
3. If your PA bean is in .class files in a directory, ensure that the directory path is in your class path. (Do not specify the package name in the directory path.)
4. Ensure that the IBM Component Broker CICS and IMS Application Adaptor component is installed.
5. Ensure that all the user classes that are referenced in the bean are included in your system environment variable CLASSPATH. The class path must either contain the JAR file (if you archived the classes into a JAR file), or the directory under which the class files exist (if you did not create a JAR file).

If you have any other trouble while importing the bean, it could be that you created the the BeanInfo class when the **Inherit BeanInfo from superclass** option was enabled in VisualAge for Java's integrated development environment (IDE).

When you select the PA persistent object in the Tasks and Objects pane, the Methods pane shows you the attributes and methods defined on the PA schema, based on those that you had defined on the PA bean that you imported. The names of method parameters may not appear as you specified them in the VisualAge for Java Integrated Development Environment (IDE). However, this is only superficial, and it does not affect the behavior when the method is called on the bean.

#### RELATED CONCEPTS

Persistent object (*Programming Guide*)  
Schema (*Programming Guide*)

#### RELATED TASKS

“Creating a component for PA data” on page 157  
“Building the JAR files” on page 561  
“Working with PA schemas” on page 862  
“Adding a persistent object from a PA schema” on page 860  
“Adding a data object implementation” on page 807  
“Mapping query method parameters to PA bean attributes” on page 161  
“Customizing PA bean query methods” on page 160

#### RELATED REFERENCES

“WHERE clause syntax”  
“Java data type mappings” on page 159  
“Internationalization of data” on page 132  
“Naming objects” on page 128

## WHERE clause syntax



You can specify what the selected query method will return using a WHERE clause, which defines the search criteria.

### Substitution Strings

A WHERE clause may contain a substitution string. Use the percent symbol (%) for the substitution character in the **WHERE clause** field. Then, in the **Substitution String** field, specify the character that the backend uses as a substitution character (for example, ? or \*).

If you want to fetch every element in the procedural application adaptor (PAA) home, then you do not need a WHERE clause for that particular query method. In that case, you should not specify a substitution string for that method.

### WHERE Clause Checking

Object Builder performs a check for the following conditions:

1. The only logical operator that can appear in the WHERE clause is AND
2. All the predicates in the WHERE clause should be of the form

$X \text{ op } [\text{CONSTANT} | \text{PARAMETER}]$

where

- a. X is an attribute reference (e.g. MenuCustomerPO.number)
- b. op must be one of =, >, <, >=, <=, or LIKE (When LIKE is the operator, the substitution character may appear inside CONSTANT.)



- c. **CONSTANT** is any string or numerical constant

You must enclose any string constants in the **WHERE** clause in single quotes. For example, **WHERE name like 'S%'**

**Note:** Exponential and hexadecimal numbers (whose representation include a letter; for example, 1.0E-324 or 0xFFFF) are not permitted as constants in the query over the PAA OOSQL **WHERE** clause. To represent one of these values, code the constant in the PAO bean directly and do not expose it in the **WHERE** clause.

- d. **PARAMETER** represents any parameter to be taken by the transaction. Every parameter name is preceded by a colon. For example, if you define a parameter named **p1** on the query method named **queryAllItems**, then the **WHERE** clause of **queryAllItems** would refer to that parameter as **:p1**

You must use parameters when the values of the criteria will not be constant and you have to use a query.

Examples:

- a. To specify the clause **WHERE MenuCustomerP0.salary > 30000**, type **WHERE salary > 30000**.  
(You do not have to type the name of the persistent object; it is automatically added to the attribute of the bean.)
- b. To use a variable (parameter) **p1** that is predefined with value 30000, type **WHERE salary > :p1**

**Note the following points:**

- You must declare the variable named **p1** on that particular query method and map **p1** to an attribute in the bean.
- Since **salary** is a number, **p1** must also be a number. If we use name, **p1** must be a string.
- A bean attribute can be mapped to only one parameter in a query method. Each parameter that you use in the **WHERE** clause must be mapped to an attribute in the PA bean. The attribute to which the parameter is mapped must be of the same type as the attribute reference.
- Each attribute of the bean can be used in each query method even if another method has used it already.
- Each parameter can be mapped to any attribute on the bean, except the key attribute.

Besides checking the syntax of the **WHERE** clause, Object Builder also performs these checks:

- If you do not use a **WHERE** clause for a particular method, then there must be no substitution string for that method

- If you create a parameter, you must map it to an attribute on the bean, and the parameter must be of the same type as the attribute
- You cannot map an OOSQL parameter to the key on the bean. (This is because the key is read-only.)



#### RELATED CONCEPTS



“Enterprise Access Builder (EAB)” on page 158

“Procedural adaptor bean (PA bean)” on page 159

Persistent object (*Programming Guide*)

Application adaptor (*Programming Guide*)

  Query Service for AIX and Windows NT (*Advanced Programming Guide*)

  Query Service for OS/390 and Solaris (*Advanced Programming Guide*)

Session Service (*Advanced Programming Guide*)

Transaction Service (*Advanced Programming Guide*)

#### RELATED TASKS

“Creating a PA schema by importing a PA bean” on page 862

“Adding a data object implementation” on page 807

“Mapping query method parameters to PA bean attributes” on page 161


“Customizing PA bean query methods” on page 160

## Editing a PA schema

**Note:** You cannot rename the PA schema.

To edit a PA schema, follow these steps:

1. Select the PA schema in the User-Defined PA Schemas folder.
2. From its pop-up menu, select **Properties**. The PA Schema wizard opens to the Attributes page. You can change the connector type information for the schema. You can select from HOD, ECI, SAP, LU 6.2, Generic, EXCI, OTMA, or IMS APPC.

 If the deployment platform is OS/390, you can only select from among EXCI, OTMA, IMS APPC, and Generic.

3. If the PA bean associated with the PA schema is queryable, click **Next**. The Query Methods page opens.
4. Here, you can map each of the the query methods of the bean to an OOSQL WHERE clause.

**Note:** If you had associated the PA persistent object that is connected with this PA schema, with a data object implementation, you must first disassociate this PA persistent object from the data object implementation (delete the persistent

object from the Persistent Object Instances folder on the Associated Persistent Objects page of the Data Object Implementation wizard) before you can change the connector type.

#### **RELATED CONCEPTS**

Schema (*Programming Guide*)

#### **RELATED TASKS**

“Working with PA schemas” on page 862

#### **RELATED REFERENCES**

“WHERE clause syntax” on page 866

## **Deleting a PA schema**

To delete a PA schema, follow these steps:

1. Delete any persistent objects that are associated with this PA schema.  
**Note:** If you delete the persistent objects from the the User-Defined Data Objects folder, or the User-Defined Business Objects folder, the schemas are automatically deleted from these folders. You still have to delete it from the User-Defined PA Schemas folder. (Follow step 2.)
2. From the pop-up menu of the schema in the User-Defined PA Schemas folder, select **Delete**.

#### **RELATED CONCEPTS**

Schema (*Programming Guide*)

Persistent object (*Programming Guide*)

#### **RELATED TASKS**

“Working with PA schemas” on page 862

“Deleting a PA persistent object” on page 861

---

## **Working with managed objects**

Managed objects are defined in the User-Defined Business Objects folder, where they are shown under the business object implementation they were added to.

Once you configure a managed object with an application, an object representing that configuration appears in the Application Configuration folder, where it is shown under the application it was added to.

You can add multiple managed objects to each business object implementation, but each component you configure will have only one managed object. In fact, the component is defined by the configuration of the managed object.

The following tasks deal with managed objects:

- “Adding a managed object” on page 871
- “Configuring a managed object” on page 588
- “Editing a managed object” on page 872
- “Editing a managed object file” on page 873
- “Editing a managed object configuration” on page 874
- “Deleting a managed object” on page 874
- “Deleting a managed object configuration” on page 875

#### **RELATED CONCEPTS**

Managed object (*Programming Guide*)

#### **RELATED TASKS**

“Working with components” on page 697

#### **RELATED REFERENCES**

“Naming objects” on page 128

“Internationalization of data” on page 132

## **Service to Use**

When you create or edit a managed object, you can set the service it will provide for its business object. The Managed Object wizard lists two services, based on the type of persistence the component needs.

You can select one of the following services:

- Transaction Service
- Session Service

### **Transaction Service**

Select this type of service if the business object is to be associated with a DB persistent object. This is the only service that can be used even if the business object is associated with a PA persistent object, and the development platform is OS/390.

### **Session Service**

Select this type of service if the business object is to be associated with a PA persistent object.

#### RELATED CONCEPTS

Managed object (*Programming Guide*)  
Business object (*Programming Guide*)  
Persistent object (*Programming Guide*)  
Session Service (*Advanced Programming Guide*)  
Transaction Service (*Advanced Programming Guide*)

#### RELATED TASKS

“Adding a managed object”

## Adding a managed object

For a component to be installed on the server, it must have a managed object. End-user applications primarily interact with the managed object, which inherits the interface of the business object. The managed object controls the key and the copy helper, the relationship between the business object and its data object, and so on. You must create a new managed object for each of your business object implementations.

To add a managed object, follow these steps:

1. From the User-Defined Business Objects folder, select your business object implementation (for example, CarPolicyBO).
2. From the object’s pop-up menu, select **Add Managed Object**. The Managed Object wizard opens to the Names and Service page.  
Appropriate names are filled in for you (the business object file name and interface name plus MO: for example, CPFile::CarPolicy gets a managed object CPFileMO::CarPolicyMO). You can accept these defaults or replace them with your own names.
3. Set the deployment platforms (the platforms on which this managed object will be deployed). This determines the development options that are selectable (you can only select options that are available on all selected platforms). By default, the managed object is deployable to the set of platforms defined in the **Platforms > Constrain** menu. You cannot select platforms that are **not** already selected in the **Platforms > Constrain** menu.
4. Make sure the correct service is selected. The services should be appropriate for the form of persistence provided by the component’s data object implementations.

**Note:** Transaction Service is the appropriate choice for every form of persistence except for the Procedural Application Adaptor. The type of service required by the Procedural Application Adaptor depends on the connector type of your PA bean. If the connector type is HOD, SAP, or ECI, use Session Service. If your connector type is Generic, you can use either Session or Transaction Service. Other connector types require Transaction Service.

 390 If one of your deployment platforms is OS/390, you can only select Transaction Service.

5. Click **Next**. The Implementation Inheritance page opens.

By default, no inheritance is selected. If, however, the managed object is for a component that already has an inheritance tree (for example, this is the managed object for the interface CarPolicy, which inherits from Policy), then the managed object should follow the same inheritance pattern (CarPolicyMO should inherit from PolicyMO).

**Note:** The options available in the **Parent Class** drop-down list are for defining a container or home (specialized forms of managed objects). There are separate instructions for these tasks. Do not use the list's options when creating a simple managed object.

 390 If the managed object is deployable to OS/390, the ISpecializedPolymorphicHomeManagedObject class is not available for selection as a parent interface.

6. Click **Finish**. The managed object appears in the User-Defined Business Objects folder, under your business object implementation.

#### RELATED CONCEPTS

Managed object (*Programming Guide*)

An overview of application adaptors (*Programming Guide*)

#### RELATED TASKS

"Working with managed objects" on page 869

"Setting platform constraints" on page 421

"Configuring builds" on page 549

"Configuring a managed object" on page 588

#### RELATED REFERENCES

"Internationalization of data" on page 132

"Naming objects" on page 128

## Editing a managed object

Managed objects are defined in the User-Defined Business Objects folder, where they are shown under the business object implementation they were added to. The configuration of a managed object with an application is represented by a separate object, in the Application Configuration folder, where it is shown under the application it was configured with.

You can change the services used by the managed object by following these steps:

1. From the pop-up menu of the managed object, click **Properties**. The Managed Object wizard opens to the Name and Services page.

2. Change your selections as necessary.
3. Click **Finish** to apply your changes.

You can change the name of the managed object file, and the platforms on which the managed object is to be deployed by following these steps:

1. From the pop-up menu of the managed object, select **File Properties**. The Managed Object File wizard opens to the Name page.
2. You can type a new name for the managed object, or select a different set of deployment platforms.
3. Click **Finish** to apply your changes.

#### RELATED CONCEPTS

Managed object (*Programming Guide*)

#### RELATED TASKS

“Working with managed objects” on page 869

#### RELATED REFERENCES

“Internationalization of data” on page 132

“Naming objects” on page 128

## Editing a managed object file

To edit a managed object file, follow these steps:

1. From the pop-up menu of the managed object, select **File Properties**. The Managed Object File wizard opens to the Name page.
2. You can type a new name for the managed object file, and select a different set of deployment platforms for the object.

WIN  AIX  SOLARIS  HP-UX Object Builder generates the IR executable based on the managed object IDL file name. `idlcc -eir` is run against the managed object IDL file to produce a C++ source file. This file is then compiled and linked to produce the IR executable. The IR executable’s name is the managed object IDL file’s root name with the suffix `_IR`. This is the executable that the DDL will run when System Management loads.

WIN This file gets the `.exe` extension only on Windows NT.

For example, given a managed object IDL file with the name `fooMO.idl`, the IR executable that is generated will be `fooMO_IR.exe` on Windows NT, and `fooMO_IR` on AIX, Solaris, and HP-UX.

3. Click **Next**. The Contents Ordering page opens. Use this page to view or set the order of constructs and interfaces in the IDL file.
4. Click **Finish** to apply your changes.

#### **RELATED CONCEPTS**

Managed object (*Programming Guide*)

#### **RELATED TASKS**

“Working with managed objects” on page 869

#### **RELATED REFERENCES**

“Internationalization of data” on page 132

“Naming objects” on page 128

## **Editing a managed object configuration**

To edit a managed object configuration, follow these steps:

1. Locate the managed object configuration in the Application Configuration folder.
2. From the pop-up menu of the configuration, click **Properties** to open the Managed Object Configuration wizard.
3. Click the page title to display the contents of the guide, and turn to a particular page.
4. Make your changes.
5. Click **Finish**.

#### **RELATED CONCEPTS**

Managed object (*Programming Guide*)

#### **RELATED TASKS**

“Working with managed objects” on page 869

## **Deleting a managed object**

To delete a managed object, follow these steps:

1. Delete its configuration, if any, in the Application Configuration folder.  
If the managed object is a specialized home, then you must also remove it from any configurations of other managed objects that use it.
2. From the pop-up menu of the managed object, click **Delete**.

#### **RELATED CONCEPTS**

Managed object (*Programming Guide*)

“Home” on page 581

#### **RELATED TASKS**

“Working with managed objects” on page 869



“Deleting a managed object configuration”

“Editing a managed object configuration” on page 874

## Deleting a managed object configuration

To delete a managed object configuration, click **Delete** from its pop-up menu.

**Note:** If the managed object configuration is part of a specialized home, then you must first remove it from any other managed object configurations that use it as their home.

### RELATED CONCEPTS

Managed object (*Programming Guide*)

### RELATED TASKS

“Working with managed objects” on page 869

“Working with specialized homes”

---

## Working with specialized homes

Customized homes, also known as specialized homes, are shown in the User-Defined Business Objects folder, and are presented in terms of five objects:

- The business object file (which contains one or more interfaces, optionally organized into modules), customized to be the file for a home.
- The business object module, if any (which contains one or more interfaces).
- The business object interface (which has one or more implementations), customized to be the interface for a home.
- The business object implementation (which has its own file, defined on the first page of its wizard), customized to be the implementation for a home.
- The managed object (which acts as an access point for the business object), customized to serve as a home.

The following tasks deal with specialized homes:

- “Creating a specialized home” on page 876
- “Editing a specialized home” on page 879
- “Deleting a specialized home” on page 879
- “Creating a specialized polymorphic home” on page 880

### RELATED CONCEPTS

“Home” on page 581

#### RELATED TASKS

“Working with components” on page 697

#### RELATED REFERENCES

“Naming objects” on page 128

“Internationalization of data” on page 132

## Creating a specialized home

When you configure a managed object with an application, you define a home instance that will be used to create and find instances of the managed object. Component Broker provides default home instances for you to base home instances on, which should be sufficient for most managed objects. However, you may want to create a home instance based on a specialized home class for the needs of a particular application adaptor type, for example, by adding customized create and find methods.

To create a specialized home, follow these steps:

1. From the pop-up menu of the User-Defined Business Objects folder, click **Add File** to open the Business Object File wizard.
2. Name the file.
3. Click the page title and turn to the Files to Include page.
4. Under the Include Files folder, click the existing file to display its information.
5. If you want a queryable home, select `IManagedAdvancedClient` from the list.
6. Complete the remaining wizard pages and click **Finish**.
7. From the pop-up menu of the file you just added, click **Add Module** to open the Business Object Module wizard.
8. Name the module and click **Finish**.
9. From the pop-up menu of the file you just added, click **Add Interface** to open the Business Object Interface wizard.
10. Click the page title and turn to the Interface Inheritance page.
11. Change the default parent interface:
  - If you want a queryable home, click the **Component Broker Specialized, Queryable Home** button, or select `IManagedAdvancedClient::IQueryableIterableHome` from the list.
  - If you want a queryable, iterable home with polymorphic properties, click the **Component Broker Specialized, Polymorphic Home**, or select `IManagedAdvancedClient::IPolymorphicHome` from the list.

- If you do not need a queryable home, click the **Component Broker Specialized Home** button to include the appropriate file.
12. Complete the remaining wizard pages and click **Finish**.
  13. From the pop-up menu of the interface you just added, click **Add Implementation** to open the Business Object Implementation wizard. On the first page, the data access and data object options are absent because this is a specialized home. The data object interface pages of the wizard are also absent.
  14. Provide any user data you want associated with the home in the central data store (CDS).
  15. Click the page title and turn to the Implementation Inheritance page. The appropriate parent implementation is selected by default.
  16. Complete the remaining wizard pages and click **Finish**.
  17. From the pop-up menu of the implementation you just added, click **Add Managed Object** to open the Managed Object wizard.
  18. Click the page title and turn to the Implementation Inheritance page.
  19. From the Parents pop-up menu, click **Add**.
  20. Select the appropriate Component Broker home class from the **Parent Class** drop-down list.
  21. Complete the remaining wizard pages and click **Finish**.
  22. Configure the specialized home, as a managed object, with your application.
 

**Note:** Make sure that the specialized home and its associated managed objects are configured with different containers. If a managed object and its home are configured with the same container, the server will not activate.

You now have a specialized home class.

Specialized homes do not require data object interfaces, data object implementations, copy helpers, or key classes. When you configure managed objects for an application, you can associate them with your specialized home (on the Managed Object Configuration wizard, Home page). An instance of the specialized home class will be defined in the generated DDL for the managed object, and used on the server to create and find instances of the managed object.

Example: Creating a specialized home

You **must** package the specialized home in the same application as the managed objects that use it.

#### RELATED CONCEPTS

“Home” on page 581  
“DDL” on page 137

#### RELATED TASKS

“Working with specialized homes” on page 875  
“Configuring a managed object” on page 588  
“Packaging applications” on page 574

#### RELATED REFERENCES

“Internationalization of data” on page 132  
“Naming objects” on page 128

### Example: Creating a specialized home



**Note:** This example is not independently reproduceable. It assumes a model where the type `Warehouse`, and corresponding managed object and business object have been created. It is just an example of how you would do it, given this type of model.

To create a specialized home, follow these steps:

1. Add a file called `warehouseHome` to the User-Defined Business Object folder.
2. Add an interface called `warehouseSpHome` to the User-Defined Business Object folder.
  - a. On the Interface Inheritance page, click the **Component Broker Specialized Home** button.
  - b. On the Method page, add a method called `create`, which returns `warehouse` of type `Warehouse`, and throws `IManagedClient IInvalidKey` and `IManagedClient IDuplicateKey`.
  - c. Add another method, called `findWarehouse`, which returns `warehouse` of type `Warehouse`; has two parameters, `parametername (id)` and type `(long)`, and throws `IManagedClient IManagedObjectKey`.
3. Right-click on the new interface, and select **Add Implementation**.
  - a. Select C++ as the implementation language.
  - b. On the Method page, add a method called `getUnique`, with a return type `long`.
4. Add a managed object to the User-Defined Business Object folder.
5. Right-click on the `warehouseSpHome` file in the User-Defined Business Object folder. Select **Generate -> All**.
6. Add `warehouseSpHomeContainer` to the Container definition folder.

7. In the Build Configuration folder, define a server DLL. Select warehouseSpHomeMO and warehouseSpHomeBO on the Server Source File page.
8. In the Application Configuration folder, add warehouseSpHomeMO to the application warehouse7. Select warehouseSpHomeContainer on the Container page.
9. In the Application Configuration, modify WarehouseMO7. Click the **Customized Home** button on the Home page. Set the **Home Name** to warehouseHomeMO warehouseSpHomeMO.
10. Generate the DDL for Wh7ApplFamily in the Application Configuration folder.
11. Create the make file by selecting **Generate->Select->C++ Default Target** from the Build Configuration folder.

## Editing a specialized home

Specialized homes are defined in the User-Defined Business Objects folder, where they are shown as a tree of business object file, module (if any), business object interface, business object implementation, and managed object. You can edit these objects by following these steps:

1. From the pop-up menu of the object, click **Properties** to display the appropriate wizard.
2. Click the page title to select a page to turn to.
3. Change your selections as necessary.
4. Click **Finish** to apply your changes.

**Note:** When you click **Finish**, the framework methods for the business object implementation are recalculated. If you made any changes to the framework method implementations (not recommended), those changes are lost.

### RELATED CONCEPTS

"Home" on page 581

### RELATED TASKS

"Working with specialized homes" on page 875

### RELATED REFERENCES

"Internationalization of data" on page 132

"Naming objects" on page 128

## Deleting a specialized home

To delete a specialized home, follow these steps:

1. Remove the specialized home from any managed object configurations that use it.
2. Delete the managed object configuration for the specialized home's managed object.
3. Delete the specialized home's managed object.
4. Delete its business object implementation.
5. Delete its business object interface.
6. If it is the only interface defined in the file, delete the business object file.

#### RELATED CONCEPTS

"Home" on page 581

#### RELATED TASKS

"Working with specialized homes" on page 875

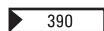
## Creating a specialized polymorphic home



When you configure a managed object with an application, you can choose to integrate polymorphic behavior into it.

To create a specialized, polymorphic home, follow these steps:


1. From the pop-up menu of the User-Defined Business Objects folder, click **Add File** to open the Business Object File wizard.
2. Name the file.
3. Click the page title and turn to the Files to Include page.
4. Under the Include Files folder, click the existing file to display its information. Object Builder generates the necessary includes automatically as you select types (such as for parent interfaces).
5. Complete the remaining wizard pages and click **Finish**.
6. From the pop-up menu of the file you just added, click **Add Module** to open the Business Object Module wizard.
7. Name the module and click **Finish**.
8. From the pop-up menu of the file you just added, click **Add Interface** to open the Business Object Interface wizard.
9. Click the page title and turn to the Interface Inheritance page.
10. Change the default parent interface:  
Click the **Component Broker Specialized, Polymorphic Home** push button. The `IManagedAdvancedClient`  
`IManagedAdvancedClient::IPolymorphicHome` class is added to the Parents folder.



If the business object interface is deployable to

OS/390, or HP-UX, this button is disabled, and the `IPolymorphicHome` class is not available for selection as a parent interface.

11. Complete the remaining wizard pages and click **Finish**.
12. From the pop-up menu of the interface you just added, click **Add Implementation** to open the Business Object Implementation wizard.  
On the first page, the data access and data object options are absent because this is a specialized home. The data object interface pages of the wizard are also absent.
13. Provide any user data you want associated with the home in the central data store (CDS).
14. Click the page title and turn to the Implementation Inheritance page.  
The appropriate parent implementation is selected by default.
15. Complete the remaining wizard pages and click **Finish**.
16. From the pop-up menu of the implementation you just added, click **Add Managed Object** to open the Managed Object wizard.
17. Click the page title and turn to the Implementation Inheritance page.  
For a polymorphic home, Object Builder selects `IManagedAdvancedServer`  
`IManagedAdvancedServer::ISpecializedPolymorphicHomeManagedObject` as the default class to inherit from.

 If the managed object is deployable to OS/390, or HP-UX, the `ISpecializedPolymorphicHomeManagedObject` class is not available for selection as a parent interface.

18. From the Parents pop-up menu, click **Add**.
19. Select the appropriate Component Broker home class from the **Parent Class** drop-down list.
20. Complete the remaining wizard pages and click **Finish**.
21. Configure the specialized polymorphic home, as a managed object, with your application. Refer to the task “Configuring a managed object” on page 588.

**Note:** Make sure that this specialized polymorphic home and its associated managed objects are configured with different containers. If a managed object and its home are configured with the same container, the server will not activate.

You now have a specialized, polymorphic home class.

Like specialized homes, specialized polymorphic homes do not require data object interfaces, data object implementations, copy helpers, or key classes. When you configure managed objects for an application, you can associate them with your specialized polymorphic home (on the Managed Object Configuration wizard, Home page). An instance of the specialized

polymorphic home class will be defined in the generated DDL for the managed object, and used on the server to create and find instances of the managed object.

You **must** package the specialized polymorphic home in the same application as the managed objects that use it.

#### **RELATED CONCEPTS**

“Home” on page 581

“Polymorphic homes” on page 581

#### **RELATED TASKS**

“Working with specialized homes” on page 875

“Configuring a managed object” on page 588

“Packaging applications” on page 574

#### **RELATED REFERENCES**

“Internationalization of data” on page 132

“Naming objects” on page 128

## **Querying abstract classes**

To query abstract classes, follow any one of these tasks:

- Provide a specialized home that has the create() and remove() methods return exceptions instead of performing their normal duties.
- Develop a component assembly (managed object assembly) of a standard managed object, a business object, a data object, and most likely, a transient data object.

This component will be configured with its specialized home, and can be built and loaded just like any other component. You can run the query against the home, and get the polymorphic behavior, but since no objects of that type can be created or deleted, the type is abstract.

**Note:** Abstract classes are not supported by VisualAge for Java, so although it will be possible to provide support for abstract classes in the EJB environment, it will have to be done outside of VisualAge for Java.

#### **RELATED CONCEPTS**

“Home” on page 581

“Container” on page 578

Managed object (*Programming Guide*)

Data object (*Programming Guide*)

Abstract base class inheritance

Naming Service (*Advanced Programming Guide*)

LifeCycle Service (*Advanced Programming Guide*)



**RELATED TASKS**

“Packaging applications” on page 574

---

## Working with container instances

Containers provide object services for components. Default containers are provided for objects with transient data. If you have objects with persistent data, or want to customize the types of service that are provided by a container, you need to define your own container instance.

The following tasks deal with containers:

- “Creating a container instance” on page 578
- “Editing a container instance”
- “Deleting a container instance” on page 884

**RELATED CONCEPTS**

“Container” on page 578

**RELATED TASKS**

“Working with components” on page 697

“Configuring a managed object” on page 588

**RELATED REFERENCES**

“Naming objects” on page 128

“Internationalization of data” on page 132

Container configuration parameters (*Programming Guide*)

Typical settings for container configuration parameters (*Programming Guide*)

Summary of supported container configurations (*Programming Guide*)

## Editing a container instance

To edit a container instance you have defined, follow these steps:

1. From the pop-up menu of your container, click **Properties**. The Container Definition wizard opens to the Name of Container and Number of Components page.
2. Click the title to select another page to turn to.
3. Change your selections as necessary.
4. Click **Finish** to apply your changes.

**RELATED CONCEPTS**

“Container” on page 578

**RELATED TASKS**

“Working with container instances” on page 883

**RELATED REFERENCES**

“Internationalization of data” on page 132

“Naming objects” on page 128

## Deleting a container instance

You cannot delete the default container instances. To delete a container instance that you have defined, follow these steps:

1. Remove the container from any managed object configurations that use it.
2. Locate the container in the Container Definition folder.
3. From the pop-up menu of the container, click **Delete**.

**RELATED CONCEPTS**

“Container” on page 578

**RELATED TASKS**

“Working with container instances” on page 883

“Packaging applications” on page 574

“Editing a managed object configuration” on page 874

---

## Working with compositions

A composition defines a combined interface for a group of components. In addition, it describes the implementation of the attributes and methods in the combined interface, which delegate to attributes and methods of the components in the group. Once you have combined the components into a composition, you can create composite components that are based on the composition.

The following tasks deal with compositions:

- “Creating a composition file” on page 885
- “Adding a composition module” on page 886
- “Adding a composition” on page 886
- “Editing a composition” on page 889

**RELATED CONCEPTS**

“Composition” on page 263

**RELATED TASKS**

“Creating a composite component” on page 261

- “Working with components” on page 697
- “Working with composite business objects” on page 891
- “Working with composite keys” on page 900

#### RELATED REFERENCES

- “Naming objects” on page 128
- “Internationalization of data” on page 132

## Creating a composition file

A composition file (IDL) is a container for your compositions. Although a file can hold multiple compositions, which you may organize into modules, you typically add one composition to each file.

To create a composition file, follow these steps:

1. From the Tasks and Objects pane, select the **User-Defined Compositions** folder.
2. From the folder’s pop-up menu, select **Add File**. The Composition File wizard opens to the Name page.
3. Type a name for the file (for example, CGFile).
4. Click **Next**. The Constructs page opens.  
Use the Constructs pop-up menu to add enumerations, exceptions, structures and so on. Any constructs you add are scoped to every interface in the file.
5. Click **Next**. The Files to Include page opens.  
IManagedClient is included by default. Do not change this.
6. Click **Next**. The Comments page opens. Type any comments you want to include as comment lines in your generated IDL code.
7. Click **Finish**. The wizard closes, and your file is added to the User-Defined Compositions folder. You can now add modules or interfaces to the file.

Once you have created the file, you can modify it by selecting **Properties** from its pop-up menu. The Composition File wizard opens again, with your selections preserved.

#### RELATED CONCEPTS

- “Composition” on page 263

#### RELATED TASKS

- “Creating a composite component” on page 261
- “Working with compositions” on page 884
- “Adding a composition module” on page 886
- “Adding a composition” on page 886

#### RELATED REFERENCES

“Internationalization of data” on page 132

“Naming objects” on page 128

## Adding a composition module

If you plan to add multiple compositions to a single file, you may want to store the compositions in separate modules. Any constructs you add to a module are scoped only to the compositions within that module. To add a module to a file, follow these steps:

1. From the User-Defined Compositions folder, select your composition file.
2. From the file’s pop-up menu, select **Add Module**. The Composition Module wizard opens to the Name page.
3. Type a name for the module.
4. Click **Next**. The Constructs page opens.  
Use the Constructs pop-up menu to add enumerations, exceptions, structures and so on.
5. Click **Next**. The Comments page opens. Type any comments you want to include as comment lines in your generated code.
6. Click **Finish**. The wizard closes, and your module is added to the User-Defined Compositions folder, underneath the file.

You can now add compositions to the module.

#### RELATED CONCEPTS

“Composition” on page 263

#### RELATED TASKS

“Creating a composite component” on page 261

“Working with compositions” on page 884

“Adding a composition”

#### RELATED REFERENCES

“Internationalization of data” on page 132

“Naming objects” on page 128

## Adding a composition

The composition is a server-side implementation object that provides a composite business object with access to its member components’ methods and data. It can also define its own methods and attributes for use by the composite business object.

To add a composition to a file (or module), follow these steps:

1. From the User-Defined Compositions folder, select the file or module that will contain the interface.
2. From the pop-up menu for the file or module, click **Add Composition**. The Composition Editor opens.
3. Click **Add** to open the Composition Palette.
4. Select the components you want to add to the composition.
5. Click **Add**, then **Close**. The components are added to the composition in the form of managed object instances, with default names based on the original component interface (for example, SavingsAccount component becomes SavingsAccount1).
6. Review the list of objects to composite in the Objects to Composite list.
7. You can rename a managed object instance by clicking on its name and then clicking **Rename** or by double-clicking on its name.
8. Select a composition style to apply to the objects. The result is applied to the list of composited attributes and methods in the Results pane. For conjunction composites you should choose the variant (that is, with or without name matching) that produces a result that is closest to what you want.
9. If you choose the **Conjunction with name matching** style, but would still like certain attributes or methods to remain separate, you can selectively reverse the name-matching and split the combined attribute or method into its separate elements. To split a combined attribute or method, select **Split** from the pop-up menu of the attribute or method.  
 Generally, key attributes should not be combined; if the **Conjunction with name matching style** has matched key attributes of the objects you are compositing (for example, SavingsAccount.accountNo and CheckingAccount.accountNo become accountNo), you should split them back out into separate attributes (savingsAccount\_accountNo and checkingAccount\_accountNo). Key attributes need to be kept separate, so that they can be delegated to by attributes of the composite key.
10. If you choose the **Conjunction without name matching** style, but would still like certain attributes or methods to be combined (as if you had chosen the **Conjunction with name matching** style), you can selectively match and combine attributes or methods. To combine multiple attributes or methods, select them by holding down the Ctrl key and clicking the left mouse button, then click on the last one with the Ctrl key plus right mouse button to display its pop-up menu, and select **Equate**.  
 Generally, you should not equate key attributes of the objects you are compositing, as noted previously.  
 You can use the **Equate** command to join attributes or methods with the same type. They do not need to have the same names.
11. Click on an attribute or method to view its republishing (delegating) behavior, in the Current Republish Value pane.

12. Click on the Properties tab to display the properties of the currently selected attribute and method. You can also double-click on the attribute or method to display its properties.
13. Only the name of the attribute or method is editable, because their definitions are based on their equivalents in the combined components.
14. Add any new attributes or methods you want to be part of the composition, that may or may not be based on combined components. You can add new attributes or methods from the pop-up menus of the folders in the Results pane. These new methods could, for example, provide extra processing of the information being combined (beyond simple delegation).

For example, a composition `AllAccounts`, which combines the components `CheckingAccount` and `SavingsAccount`, could have a private helper method `addFloats`, which can take the two original balances (`CheckingAccount1.balance` and `SavingsAccount1.balance`) as arguments, and return their sum. You can then map `AllAccounts.balance` to the helper method.

When you add a new method, you can supply its implementation (for example, `return arg1+arg2`) in Object Builder's Source pane (after you complete the composition, click on it in the Tasks and Objects pane; then select the method in the Methods pane, and complete its implementation in the Source pane).

15. Edit republishings using the pop-up menu of the current value in the pane. You can also change a republish value by simply double-clicking on it, and then selecting a new value from the drop-down list that appears.

For conjunction composites, you can map attributes and methods either to attributes and methods of the combined components, or to a *sequence* of attributes and methods (in which all listed attributes or methods are called in sequence, and the result of the last one is returned). The delegating attribute or method must have a type or return type that matches the result of the last call in the sequence.

Disjunction attributes and methods usually map to a *select* of attribute and methods (that is, a list of mutually exclusive attributes and methods, only one of which will exist at run time and be called).

You can map to attributes or methods of the combined components, or to other attributes and methods that are unique to the composition.

For example, if the composition `AllAccounts` combines `CheckingAccount` and `SavingsAccount` with the Conjunction with name matching style, then by default `AllAccounts.balance` returns a **sequence** of `CheckingAccount1.balance` and `SavingsAccount1.balance` (which simply returns the second value in the sequence). You can replace this default mapping with a more useful one that returns their sum, by adding a

private helper method `addFloats` (as described in the previous step), and changing the mapping to call the helper method, with the two original `balance` attributes as arguments.

16. Click on the parent folder (representing the composition as a whole) in the **Results** pane. By default, its name is **Untitled**.
17. Click on the **Properties** tab.
18. Type a name for the composition. The name is reflected in the **Results** pane.
19. Set the implementation language (C++ or Java).
20. Click **OK**.

#### **RELATED CONCEPTS**

“Composition” on page 263

#### **RELATED TASKS**

“Creating a composite component” on page 261

“Working with compositions” on page 884

“Adding a composite business object interface” on page 892

“Adding a composite key” on page 901

#### **RELATED REFERENCES**

“Internationalization of data” on page 132

“Naming objects” on page 128

## **Editing a composition**

Compositions are defined in the User-Defined Compositions folder, where they are shown under the file (and module, if any) in which they are defined. You can edit the file and module as separate objects, following these steps:

1. From the pop-up menu of the file or module, click **Properties** to display the appropriate wizard.
2. Click the page title to select a page to turn to.
3. Change your selections as necessary.
4. Click **Finish** to apply your changes.

To edit the attributes or methods of the composition, follow these steps:

1. From the pop-up menu of the composition group, click **Properties** to open the Composition Editor.

You can edit the delegating behavior of the methods or attributes currently in the composition, add new methods or attributes that are unique to the composition, or change the components that are combined in the composition.

2. When you are done editing, click **OK**.

From within the editor, you can turn to the Compositions page to change the components that make up the composition:

To delete a component from the composition, follow these steps:

1. Select its managed object instance in the Objects to Composite list.
2. Click **Delete**.

To add a component to the composition, follow these steps:

1. Click **Add** to open the Composition Palette.
2. Select a managed object.
3. Click **Add**, then **Close**.

To rename a component in the composition, follow these steps:

1. Select its managed object instance in the Objects to Composite list.
2. Click **Rename**.
3. Type over the old name.
4. Click elsewhere in the list to apply the new name.

Once you are done editing, click **OK** to apply your changes.

If your changes are limited to renaming or deleting elements, then your changes will automatically be reflected in the other composite component objects that are based on the composition (for example, the business object and key). If, however, you added new components to the composition, you need to provide these objects with the information necessary to locate or create instances of the new component.

When you have added new components to a composition, follow these steps for each composite component based on the composition:

1. From the pop-up menu of the composite component's key, click **Properties** to open the Key wizard.
2. Add any new key attributes that may be required and provide the composite key to component key mappings if possible.
3. Click **Finish**.
4. From the pop-up menu of the composite component's business object implementation, click **Properties** to open the Business Object Implementation wizard.
5. Click the page title and turn to the Location page.
6. Review and update the location information for any new or edited components of the composition.
7. Click the page title and turn to the Data Object Interface page.
8. Add any new key attributes to the data object interface.
9. Click **Finish**.



**RELATED CONCEPTS**

“Composition” on page 263

**RELATED TASKS**

“Working with compositions” on page 884

**RELATED REFERENCES**

“Internationalization of data” on page 132

“Naming objects” on page 128

---

## Working with composite business objects

Composite business objects are defined in the User-Defined Business Objects folder, and are presented in terms of four objects:

- The business object file (which contains one or more interfaces, optionally organized into modules)
- The business object module, if any (which contains one or more interfaces)
- The business object interface (which has one or more implementations)
- The business object implementation (which has its own file, defined on the first page of its wizard)

The four objects are created and edited separately, but collectively form a single business object. Each business object (each set of business object file, module, interface, and implementation) typically has its own data object.

A business object is composite when it is based on a composition, as set by the business object interface. A composite business object has attributes and methods based on those in the composition, which are in turn based on the composited components that make up the composition.

The following tasks deal with composite business objects:

- “Adding a composite business object interface” on page 892
- “Adding a composite business object implementation and data object interface” on page 894
- “Editing a composite business object interface” on page 898
- “Editing a composite business object implementation” on page 899

**RELATED CONCEPTS**

“Composite business object” on page 264

**RELATED TASKS**

“Working with components” on page 697

“Creating a composite component” on page 261

“Working with composite keys” on page 900

#### RELATED REFERENCES

“Naming objects” on page 128

“Internationalization of data” on page 132

## Adding a composite business object interface

Once you have defined a composition, you can create composite components that are based on the composition, starting with the business object.

First, add a business object file (and optionally module) to the User-Defined Business Objects folder.

To add a composite business object interface to a file (or module), follow these steps:

1. From the User-Defined Business Objects folder, select the file or module that will contain the interface.
2. From the pop-up menu for the file or module, select **Add Interface**. The Business Object Interface wizard opens to the Name page.
3. Type a name for the interface (for example, CompositeCustomer). Do not use the same name as the composition unless one or both are nested in modules.
4. Select the **Composite** check box.
5. From the **Composition to Use** list, select the composition you want to base the interface on.
6. Click **Next**. The Constructs page opens.
7. Use the Constructs pop-up menu to add enumerations, exceptions, structures and so on. Any constructs you add are scoped to this interface only.

**Note:** To use the construct as the type of an attribute, method return, or method exception, you must first click **Finish** and then re-open the wizard and define the attribute. The construct is not added to the current model until you click **Finish**.

8. Click **Next**. The Interface Inheritance page opens.

By default, the interface inherits from `IManagedClient::IManageable`. This is the correct choice for a component that represents a base class in your design. If your component had a parent, you would specify the business object interface of the parent component on this page.

9. Click **Next**. The Attributes page opens.

The public attributes (except for the attributes that represent references to instances of the combined components) of the composition you selected appear here, but are not editable. You can edit them (for example, change

their names or delegating behavior) in the composition where they are defined. Changes to the composition are applied to the composite business object automatically.

When you add the composite business object implementation, the get and set methods for these attributes will be implemented, and delegate to the composition helper object.

To specify additional attributes for your interface, select **Add** from the Attributes pop-up menu. When you add the business object implementation, these attributes will receive default implementations in the usual manner.

10. Click **Next**. The Methods page opens.

The public methods of the composition you selected appear here, but are not editable. You can edit them in the composition where they are defined. Changes to the composition are applied to the composite business object automatically.

When you add the composite business object implementation, the implementations for these methods will be implemented, and delegate to the composition helper object.

To specify additional methods for your interface, select **Add** from the Methods pop-up menu. When you add the business object implementation, you can provide your own implementations for the methods in the usual manner.

11. Click **Next**. The Object Relationships page opens.

To specify any relationships that this class has to other classes, select **Add** from the Objects pop-up menu. You can specify how the relationship will be implemented when you add the business object implementation.

12. Click **Next**. The Comments page opens. Type any comments you want to include as comment lines in your generated code.
13. Click **Finish**. Your new interface is added to the User-Defined Business Objects folder, with the attributes and methods of the composition you selected, as well as any additional attributes and methods you specified.

You should now see your interface in the Tasks and Objects pane. Any methods defined for your interface should appear under the **User-Defined Methods** folder in the Methods pane, and any attributes defined for your interface should appear under the **User-Defined Attributes** folder in the Methods pane.

#### **RELATED CONCEPTS**

“Composite business object” on page 264  
Business object (*Programming Guide*)

#### RELATED TASKS

“Creating a composite component” on page 261

“Working with composite business objects” on page 891

“Adding a composite key” on page 901

#### RELATED REFERENCES

“Internationalization of data” on page 132

“Naming objects” on page 128

## Adding a composite business object implementation and data object interface

Once you have created a composite business object interface, you must add one or more implementations for that composite business object, and also create its data object interface. You can accomplish both tasks using the Business Object Implementation wizard. Ensure that you have added a key to the composite business object interface before proceeding with this task.

To create the composite business object implementation, and its associated data object interface, follow these steps:

1. From the User-Defined Business Objects folder, select the composite business object interface you want to implement.
2. Display the pop-up menu for the interface and select **Add Implementation**. The Business Object Implementation wizard opens to the Name and Data Access Pattern page.

Appropriate implementation names are filled in for you (the business object file name and interface name plus BO: for example, AAFile::AllAccounts gets an implementation named AAFileBO::AllAccountsBO). You can accept these defaults or replace them with your own names.

3. Select the pattern you want to use for handling the component’s state data (that is, any attributes of the component that are **not** derived from the composition it is based on). The following patterns are available:
  - **Delegating**  
The business object delegates every request for the essential state to the data object interface.
  - **Caching**  
Both the business object and the data object have their own copies of the essential state, which are synchronized. **Lazy evaluation** is the default synchronization method, meaning that cached copies of the attributes are synchronized at first use, rather than at instantiation.
  - **Same as parent’s**  
The business object inherits its pattern from a parent interface.  
**Note:** This option is selected by default if the interface for this business

object inherits from another business object interface. However, you still have to indicate the implementation parent on the Implementation Inheritance page of this wizard.

There is also an option listed for **None**, which would generate a transient data object. This option is not available in this release.

The pattern you select will apply for any attributes you created that are unique to the composite component. It does not apply to any attributes derived from the component's composition. Attributes derived from the composition are always implemented as delegation calls to their equivalents in the composition helper object, regardless of the pattern selected here.

4. Select whether to create a new data object now, or add or select one later.
5. Click **Next**. The Implementation Inheritance page opens.
6. Make sure that `IManagedClient::IManageable` is listed as a parent under the Parent Class folder.
7. You can also select any parent business object implementations you want to inherit behavior from.
8. Click **Next**. The Implementation Language page opens. Select the language you want the business object to be implemented in. You can select either Java or C++.
9. The default for this page is set in the Preferences notebook, on the Tasks and Objects page.
10. Click **Next**. The Attributes page opens.

A private attribute with the same name as the composition being used is automatically included. This attribute is used to access the composition helper object in the delegating implementations of composite attributes and methods (for example, the composite component method `debit` calls the composition's method `iCompositeAccount.debit`).

You can also specify any attributes you want to add to the business object implementation (in addition to the attributes you already specified in the business object interface).
11. Click **Next**. The Methods page opens.
12. Several private methods related to composition are automatically included:

- A method of the form `loc_<instance name>` is included for each component instance of the composition being used (e.g. `loc_SavingsAccount1` and `loc_CheckingAccount1`). These methods are called during activation to locate or create the managed objects that are

used to initialize the composition helper object when it is created. The implementation of these methods is automatically generated by Object Builder using the information provided on the Location page (see below).

- A method of the form *get\_<instance name>\_<key attribute name>* is included for every key attribute of each component instance of the composition being used (for example, *get\_SavingsAccount1\_accountNo* and *get\_CheckingAccount1\_accountNo*). These methods are called by the *loc\_* methods to get the values used to initialize the primary key attributes of the component instances. These methods will be automatically generated by Object Builder if simple key attribute mappings were supplied for the composite key.

You can also specify any methods you want to add to the business object implementation (in addition to the methods you already specified in the business object interface).

13. Click **Next**. The Key and Copy Helper page opens. Select a key and, optionally, a copy helper that you have created for this business object (for example, *AllAccountsKey* and *AllAccountsCopy*).

14. Click **Next**. The Handle Selection page opens.

You can select a handle for the business object implementation. If you select a handle, then the framework method *getHandleString* is implemented, which overrides the *getHandleString* method of *IManagedClient::IManageable*. The method provides a way to encapsulate the business object implementation, by returning a string that represents a reference to the business object. The handle you select determines the pattern used to form the string (that is, to turn the reference into a string, or to swizzle the pointer).

15. Click **Next**. The Location page opens.

On this page, you set the composite business object's relationship to the managed objects being combined in the composition.

16. For each managed object in the composition, provide the following information:

- a. Indicate whether it should be destroyed when the composition is destroyed, or have its destruction managed independently.
- b. Indicate the expected state of the managed object:
  - Click **Find or create** if the managed object might or might not already exist.
  - Click **Find** if the managed object must already exist (and should not be created if it does not).
  - Click **Create** if the managed object must **not** already exist (and should not be returned if it does).

- c. Indicate whether the managed object should be created using a copy helper instead of its primary key. When creating a component using a copy helper, the attributes that are also key attributes will be initialized as usual (by calling `get_` methods). The other attributes on the copy helper will be set to the initial values specified in the Interface wizard of the component being created.  
**Note:** Components using PAA Services (that is, CICS components) can only be created using a copy helper.
  - d. Select the way the managed object should be located.
  - e. Provide the information necessary to implement the selected location pattern.
17. Click **Next**. If the business object implementation has parent classes with overrideable attributes, then the Attributes to Override page opens.  
You can use this page to select which of the parent class's attributes you want to override.
  18. Click **Next**. If the business object implementation has parent classes with overrideable methods, then the Methods to Override page opens.  
You can use this page to select which of the parent class's methods you want to override.
  19. Click **Next**. If the business object interface defines one-to-many relationships, then the Object Relationships page opens.  
You can use this page to set the way that the object relationship will be implemented.
  20. Click **Next**. The Data Object Interface page opens.  
**Note:** This page does not open if, on the first page, you chose not to create a new data object.  
Appropriate data object names are filled in for you (the business object file name and interface name plus DO: for example, `AAFile::AllAccounts` gets the data object interface `AAFileDO::AllAccountsDO`). You can accept these defaults or replace them with your own names.  
If you implemented a one-to-many relationship as a **Local persistent reference**, then an attribute representing it appears here, so you can select to preserve it in the data object.
  21. Select the attributes you want preserved in the data object. Because this component is a composite one, the state for all of the composite attributes is already preserved in the referenced components. In other words, composite attributes are already preserved in the data objects of their originating components. You only need to select the key attributes here, and any non-composite (not derived from the composition) attributes you defined for the business object.
  22. Click **Next**. The Data Object Methods page opens. (This page does not open if, on the first page, you chose not to create a new data object.)

23. Select which business object methods you want to push down to the data object (that is, call equivalent methods to be defined in the data object).
24. Click **Next**. The Summary of Framework Methods page opens.

Based on your selections on the previous pages of the wizard, this page displays the methods that your object implements. For example, if you selected a caching pattern to handle the essential state of your business object (on the first page), this list includes the `synchToDataObject` method required to keep the two sets of attributes synchronized.

Because this business object is a composite, this list also includes two composition methods, `initializeComposition` and `uninitComposition`. These two methods are also automatically generated by Object Builder.

You can review the framework methods before closing the wizard.
25. Click **Finish**. The business object implementation and data object interface appear in the User-Defined Business Objects folder, under your business object interface. The data object interface also appears in the User-Defined Data Objects folder.

Now that the business object implementation is defined, you can enter the implementation code for any new methods you defined.

#### **RELATED CONCEPTS**

“Composite business object” on page 264  
Business object (*Programming Guide*)  
Data object (*Programming Guide*)

#### **RELATED TASKS**

“Creating a composite component” on page 261  
“Working with composite business objects” on page 891  
“Implementing methods” on page 752  
“Adding a data object implementation” on page 807  
“Defining a 1-n relationship” on page 281

#### **RELATED REFERENCES**

“Internationalization of data” on page 132  
“Naming objects” on page 128

## **Editing a composite business object interface**

Composite business object interfaces are defined in the User-Defined Business Objects folder, where they are shown under the file (and module, if any) in which they are defined.

Composite business objects are based on compositions, from which they derive attributes and methods. These derived methods are not editable in the business object interface. You can edit them (for example, change their names



or delegating behavior) in the composition where they are defined. Changes to the composition are applied to the composite business object automatically.

You can edit the file, module, and the non-composite attributes and methods of the business object interface as follows:

1. From the pop-up menu of the file, module, or interface, click **Properties** to display the appropriate wizard.
2. Click the the page title to select a page to turn to.
3. Change your selections as necessary.

If you want to specify a parent for the interface after you have defined the implementation for the business object, follow these steps:

- a. Add the parent to the **Parents** folder on the Interface Inheritance page of the Business Object Interface wizard
  - b. Open the Business Object Implementation wizard, and on the Name and Data Access Pattern page specify the pattern for handling state data as **Same as parent's**.
  - c. Click **Next**.
  - d. Add the implementation parent on the Implementation Inheritance page.
4. Click **Finish** to apply your changes.

#### **RELATED CONCEPTS**

"Composite business object" on page 264

#### **RELATED TASKS**

"Working with composite business objects" on page 891

#### **RELATED REFERENCES**

"Internationalization of data" on page 132

"Naming objects" on page 128

## **Editing a composite business object implementation**

Composite business object implementations are defined in the User-Defined Business Objects folder, where they are shown under the composite business object interface they were added to. You can edit a composite business object implementation by following these steps:

1. From the pop-up menu of the business object implementation, click **Properties**. The Business Object Implementation wizard opens to the Name and Data Access Pattern page.
2. Click the page title to select another page to turn to.
3. Change your selections as necessary.
4. Click **Finish** to apply your changes.

**Note:** The **Same as parent's** option is selected by default if the interface for this business object inherits from another business object interface. However, you still have to indicate the implementation parent on the Implementation Inheritance page of this wizard.

**RELATED CONCEPTS**

"Composite business object" on page 264

**RELATED TASKS**

"Working with composite business objects" on page 891

"Setting platform constraints" on page 421

**RELATED REFERENCES**

"Internationalization of data" on page 132

"Naming objects" on page 128

---

## Working with composite keys

A composite key object defines which attributes are to be used to find a particular instance of the composite component on the server. The key consists of one or more of the business object attributes, which must contain enough information to uniquely identify an instance.

The following tasks deal with composite keys:

- "Adding a composite key" on page 901
- "Editing a composite key" on page 903

**RELATED CONCEPTS**

"Composite key" on page 265

"Composite component" on page 262

**RELATED TASKS**

"Creating a composite component" on page 261

"Working with components" on page 697

"Working with composite business objects" on page 891

**RELATED REFERENCES**

"Naming objects" on page 128

"Internationalization of data" on page 132

## Adding a composite key

Each composite component must have a primary key class that contains enough information to uniquely identify the component. The key is used when new instances of the component are created or when existing instances need to be found.

Once you have created a composite business object interface, you can define its composite key.

To add a composite key, follow these steps:

1. From the User-Defined Business Objects folder, select your composite business object interface.
2. From the object's pop-up menu, select **Add Key**. The Key wizard opens to the Name and Key Attributes page.
3. Appropriate key names are filled in for you (the business object file name and interface name plus Key: for example, AAFile::AllAccounts gets a key named AAFileKey::AllAccountsKey). You can accept these defaults or replace them with your own names.
4. Select the composite business object attributes that make up the primary key.

If possible, you will want to select attributes that were part of the keys for the original combined components.

For example, given the following situation:

- A composite component AllAccounts is based on a composition of two other components, SavingsAccount and CheckingAccount.
- The primary keys of both SavingsAccount and CheckingAccount contain a single attribute accountNo (the account number).
- The two account numbers are exposed in the composite business object as attributes savingsAccountNo and checkingAccountNo.

If you select the attributes savingsAccountNo and checkingAccountNo for the primary key of AllAccounts, the composite key then includes all the information needed not only to uniquely identify the AllAccounts component, but to identify the SavingsAccount and CheckingAccount components as well. This eliminates the need to maintain persistent references from the composite component to the original combined components.

If the composite business object has a parent business object (specified on the Interface Inheritance page of its wizard), you can also select from the parent interface's attributes (you should **not** select attributes of the parent interface if you are planning to inherit from the parent interface's key).

5. Click **Next**. The Composite Key page opens. Here you are given the opportunity to provide mappings between the composite key attributes and the attributes of keys for the grouped components.
6. For each key attribute you selected that corresponds directly to an attribute of a component key, describe the mapping:
  - a. Select an attribute in the Composite Key list (for example, `checkingAccountNo`).
  - b. Select an attribute of a key in the Composite Key Elements list (for example, the `accountNo` attribute of `CheckingAccountKey`).
  - c. Click **Add**.
7. Click **Next**. The Implementation Inheritance page opens.  
On this page, you can specify the type of key (primary or unique), and inherit from the appropriate parent class (`IPrimaryKey` or `IUniqueKey`).
8. Verify that the primary key type is selected.  
If the key has a parent, you can specify it here.  
**Note:** You should not inherit from a parent key if you also selected inherited attributes on the previous page.
9. Click **Next**. The Summary of Framework Methods page opens. This page summarizes the framework methods this object implements. No action is needed.
10. Click **Next**. The Optional Framework Methods page opens. Select any additional framework methods you want to implement. Object Builder will add signatures for the methods you select, but you must provide your own implementation code. The methods you implement will override the equivalent framework methods of the parent class.  
**Note:** The Source pane will not allow you to edit these methods until you set them as editable in the Method Implementation wizard. To set a method as editable, follow these steps:
  - a. In the Methods pane, select the framework method.
  - b. From its pop-up menu, click **Properties**.
  - c. In the Method Implementation wizard, specify that you want to use the implementation defined in the Source pane. This lets you use the Source pane editor to edit the method implementation.
11. Click **Finish**. The key appears in the User-Defined Business Objects folder, under your composite business object interface.

In the Methods pane, you should see some items listed in the Framework Methods folder. Default implementation code is provided for these methods, which you can view in the edit pane by selecting a method. Normally, you will not want to edit this code (except for the code for the optional framework methods, as noted above). The code for framework methods is read-only by default.

**RELATED CONCEPTS**

“Composite key” on page 265  
Key (*Programming Guide*)

**RELATED TASKS**

“Creating a composite component” on page 261  
“Working with composite keys” on page 900  
“Adding a composite business object implementation and data object interface” on page 894

**RELATED REFERENCES**

“Internationalization of data” on page 132  
“Naming objects” on page 128

## Editing a composite key

Composite keys are defined in the User-Defined Business Objects folder, where they are shown under the composite business object interface they were added to. You can edit a key by following these steps:

1. From the pop-up menu of the key, click **Properties**. The Key wizard opens to the Name and Key Attributes page.
2. Click the page title to select a page to turn to.
3. Change your selections as necessary.
4. Click **Finish** to apply your changes.

**RELATED CONCEPTS**

“Composite key” on page 265

**RELATED TASKS**

“Working with composite keys” on page 900

**RELATED REFERENCES**

“Internationalization of data” on page 132  
“Naming objects” on page 128



---

## Chapter 16. Troubleshooting

---

### Troubleshooting

►CHANGED

If you encounter problems when you build your code into DLLs, the following tips may help you.

#### Memory problems

If you are working with a large project (more than 30 components), you may need to increase the maximum heap size of the Java virtual machine. You can do so by editing the ob.bat file:

1. Make sure Object Builder is closed.
2. Edit \Cbroker\bin\ob.bat
3. Change the parameter -mx255m, increasing the number by five for each additional component in your project (this number is approximate, and assumes components of average complexity).

For example, if your project contains one hundred components then change the parameter to -mx605m (seventy additional components multiplied by 5m each, plus the original 255m).

4. Start Object Builder. The new parameter is used, and the maximum size of the Java virtual machine is increased.

#### Generally odd behavior

Not all exceptions are displayed in the user interface. After major actions such as saving a project, check Object Builder's command window for any exceptions. The command window is the window from which you started Object Builder, or the window that appeared in the background if you started Object Builder from the **Start** menu.

#### Cannot start Object Builder on AIX

If you receive the operating system message "Killed" when you try to start Object Builder, you need to increase the amount of paging space on your machine. Object Builder requires a minimum of 200MB of paging space in order to run on AIX.

#### BAD\_OPERATION exception with composite components

If the client program receives a BAD\_OPERATION exception while using a composite component, the most probable cause is inaccurate location information in the business object implementation's properties. Look in the activity log of the server to determine the cause of the exception. If the

problem is a failure to locate one of the member components in the composition, check the Location page of the composite component's Business Object Implementation wizard.

### **Java server fails to run with composite components**

If the Java client and Java server are installed on the same machine, make sure the CLASSPATH has the files ibmcbjs.zip and somshor.zip listed before somojor.zip. Otherwise the Java business object will attempt to use the ORB interfaces in somojor.zip, instead of the server-side ORB interfaces it requires in ibmcbjs.zip and somshor.zip.

### **Error on running Object Builder: "The input line is too long"**

If the CLASSPATH environment variable is too long (approximately 1700 characters or more), then you cannot run Object Builder. You must shorten the classpath by removing directories and .jar or .zip files (besides those added by the Component Broker install, and besides any PA beans and their dependencies being used in your projects), before running Object Builder.

### **Unsatisfactory build performance**

The performance of a build (nmake -f all.mak), particularly if you are using Java-based compilers such as javac, rmic, and so on can suffer if Object Builder, or any other large Java application is up and running at the time of the build. To improve the performance of large builds, close Object Builder after generating either from within Object Builder or by using obgen, and run nmake from the command line.

### **Compilation fails because of an incorrect user ID and password error**

If your compile command fails due to an incorrect DB2 user ID and password error, run the following command before you run the make (AIX) or nmake (NT) command:

```
export IVB_DB2AUTH="USER test USING password"
set IVB_DB2AUTH=USER test USING password
```



---

## Appendix. IR Browser

---

### Starting the IR Browser

The IR Browser enables you to examine and modify the contents of the Component Broker Interface Repository.

You can start the IR Browser in one of the following ways:

- From the command line, type `irbrowser`.
- From the Windows NT desktop, select **Start > Programs > IBM Component Broker > Interface Repository (IR) Browser**.

To exit the IR Browser, select **Repository > Quit**.

#### **RELATED CONCEPTS**

Interface Repository (*Advanced Programming Guide*)

---

### Viewing objects in the repository

#### **Viewing the definition of an object**

To view the contents of an object, double-click on the object in the Containmentview. A textual representation (such as the IDL definition or IR dump output) appears in the IDL view.

#### **Viewing relationships between objects**

To view the ancestors or children of an interface, double-click the object in the Containment or Inheritance view. A graphical representation appears in the Inheritance view, showing the following relationships to the highlighted object:

- Direct base (parent) interfaces
- Direct derived (child) interfaces

The flow of the Inheritance view is from left to right. The base interfaces are to the left of the selected interface, and the derived interfaces are to the right.

To view the siblings of an interface, double-click the direct base interface in the Containment view. The tree expands to show a hierarchical representation. A container such as a module can be expanded to show sibling interfaces for the interface.

#### **RELATED TASKS**

“Finding an object” on page 908

## Viewing the operations of an interface

To view the operations of an interface, double-click the object in the Containment or Inheritance view. The hierarchical representation for the object, which shows the container relationships, appears in the left pane of the window.

---

## Searching the repository

### Finding an object

To find an object in the Interface Repository, follow these steps:

1. Select **Search > Find**. The Find window opens.

The screenshot shows a 'Search' dialog box with the following fields and options:

- Search By:** Name (dropdown menu)
- Name:** (text input field)
- Case Sensitive:**
- Search In:** (dropdown menu)
- Object Types:**
  - Attribute
  - Constant
  - Exception
  - Interface
  - Module
  - Operation
  - Type
- Used By:**
  - Attribute
  - Constant
  - Exception
  - Interface
  - Module
  - Operation
  - Type

Buttons: Find, Cancel

2. Click the **Search By** field, and select either **Object name** or **Repository ID**. Both methods accept wildcard characters as input.
3. To specify your search criteria, select or clear the **Object Types** and **Used By** check boxes.
4. Click **Find**. All objects that match the search criteria are listed in the **Result** box.
5. Double-click on an object in the list. It will appear in the Containment, IDL, and Inheritance views.
6. Close the window: click the Close button (X) in the right hand, top corner of the window.

**Note:** Due to the size and complexity of the Interface Repository, some searches might take several minutes. You can click **Cancel** to stop the search, and narrow your search criteria.

**RELATED TASKS**

“Finding an interface’s referencing operations”

“Searching with wildcards”

“Searching by object type”

## Searching with wildcards

The Find window uses a string-matching facility to find object types either within the selected containment or in the entire repository. The following wildcards are allowed:

**Asterisk (\*)**

Matches any number of characters.

**Question mark (?)**

Matches one character.

**RELATED TASKS**

“Finding an object” on page 908

## Finding an interface’s referencing operations

To find the operations that reference a particular interface, follow these steps:

1. In the IR Browser, select **Search > Find**.
2. Type the name of the interface in the **Name** field.
3. Under **Object Types**, select **Interface**.
4. Under **Used By**, select the listed operations that you are interested in.
5. Click **Find**.

**RELATED TASKS**

“Searching with wildcards”

“Searching by object type”

## Searching by object type

To find an attribute, constant, exception, interface, module, operation or type, follow these steps:

1. Select **Search > Find**. The Find window opens.
2. In the **Object Name** field, type the name of the attribute, constant, exception, interface, module, operation, or type.  
**Restriction:**You can only use single-byte alphanumeric characters to search for objects.

- To narrow the scope of the search, select the appropriate **Object Type**.  
**Note:** Use the **Used By** check boxes to restrict the search to a list of objects that reference the input objects.
- Click **Find**.

All objects that match the search criteria are listed in the **Result** list box. When you select an item from the list, that object is highlighted in the Containment view and displayed in the IDL and Inheritance views.

#### RELATED TASKS

“Finding an interface’s referencing operations” on page 909

“Searching with wildcards” on page 909

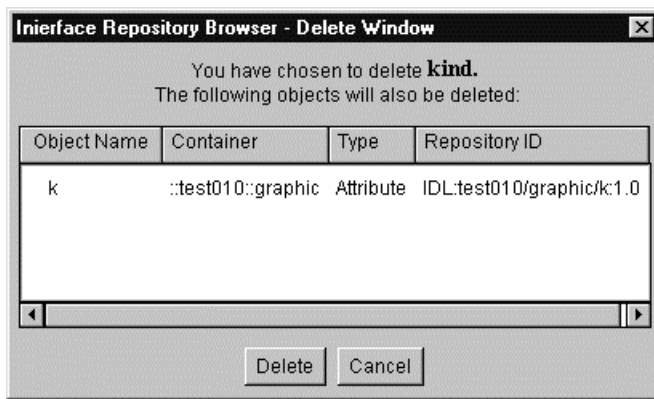
---

## Deleting objects from the repository

**Note:** Objects that you delete from the Interface Repository cannot be restored.

To permanently delete an object, follow these steps:

- Select **Options > Allow Updating Interface Repository**.
- In either the Containment or the Inheritance view, select the object that you want to delete.
- Select **Edit > Delete**. The IR Browser returns a dialog box listing any objects that will also be deleted as a result of your action.



- Verify that you want to delete all of these objects by clicking **Delete**. Otherwise, click **Cancel**.

---

## Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing  
IBM Corporation  
North Castle Drive  
Armonk, NY 10504-1785  
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation Licensing  
2-31 Roppongi 3-chome, Minato-ku  
Tokyo 106, Japan

**The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:**

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS DOCUMENT "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will

be incorporated in new editions of the document. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

**For Component Broker:**

IBM Corporation  
Department LZKS  
11400 Burnet Road  
Austin, TX 78758  
U.S.A.

**For TXSeries:**

IBM Corporation  
ATTN: Software Licensing  
11 Stanwix Street  
Pittsburgh, PA 15222  
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

---

## Trademarks and service marks

The following terms are trademarks or registered trademarks of the IBM Corporation in the United States, other countries, or both:

AFS	IMS
AIX	MQSeries
AS/400	MVS/ESA
CICS	OS/2
CICS OS/2	OS/390
CICS/400	OS/400
CICS/6000	PowerPC
CICS/ESA	RISC System/6000
CICS/MVS	RS/6000
CICS/VSE	S/390
CICSplex	Transarc
DB2	TXSeries
DCE Encina Lightweight Client	VSE/ESA
DFS	VTAM
Encina	VisualAge
IBM	WebSphere

Microsoft, Windows, Windows NT, and the Windows 95 logo are trademarks of Microsoft Corporation in the United States and/or other countries.

Oracle and Oracle8 are registered trademarks of the Oracle Corporation in the United States and/or other countries.

UNIX is a registered trademark of The Open Group in the United States and/or other countries licensed exclusively through X/Open Company Limited.

OSF and Open Software Foundation are registered trademarks of the Open Software Foundation, Inc.

\* HP-UX is a Hewlett-Packard branded product. HP, Hewlett-Packard, and HP-UX are registered trademarks of Hewlett-Packard Company.

Orbix is a registered trademark and OrbixWeb is a trademark of IONA Technologies Ltd.

Sun, SunLink, Solaris, SunOS, Java, all Java-based trademarks and logos, NFS, and Sun Microsystems are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

Some of this documentation is based on material from Object Management Group bearing the following copyright notices:

Copyright 1995, 1996 AT&T/NCR  
Copyright 1995, 1996 BNR Europe Ltd.  
Copyright 1991, 1992, 1995, 1996 by Digital Equipment Corporation  
Copyright 1996 Gradient Technologies, Inc.  
Copyright 1995, 1996 Groupe Bull  
Copyright 1995, 1996 Expersoft Corporation  
Copyright 1996 FUJITSU LIMITED  
Copyright 1996 Genesis Development Corporation  
Copyright 1989, 1990, 1991, 1992, 1995, 1996 by Hewlett-Packard Company  
Copyright 1991, 1992, 1995, 1996 by HyperDesk Corporation  
Copyright 1995, 1996 IBM Corporation  
Copyright 1995, 1996 ICL, plc  
Copyright 1995, 1996 Ing. C. Olivetti &C.Sp  
Copyright 1997 International Computers Limited  
Copyright 1995, 1996 IONA Technologies, Ltd.  
Copyright 1995, 1996 Itasca Systems, Inc.  
Copyright 1991, 1992, 1995, 1996 by NCR Corporation  
Copyright 1997 Netscape Communications Corporation  
Copyright 1997 Northern Telecom Limited  
Copyright 1995, 1996 Novell USG  
Copyright 1995, 1996 02 Technolgies  
Copyright 1991, 1992, 1995, 1996 by Object Design, Inc.  
Copyright 1991, 1992, 1995, 1996 Object Management Group, Inc.  
Copyright 1995, 1996 Objectivity, Inc.  
Copyright 1995, 1996 Oracle Corporation  
Copyright 1995, 1996 Persistence Software



Copyright 1995, 1996 Servio, Corp.  
Copyright 1996 Siemens Nixdorf Informationssysteme AG  
Copyright 1991, 1992, 1995, 1996 by Sun Microsystems, Inc.  
Copyright 1995, 1996 SunSoft, Inc.  
Copyright 1996 Sybase, Inc.  
Copyright 1996 Taligent, Inc.  
Copyright 1995, 1996 Tandem Computers, Inc.  
Copyright 1995, 1996 Teknekron Software Systems, Inc.  
Copyright 1995, 1996 Tivoli Systems, Inc.  
Copyright 1995, 1996 Transarc Corporation  
Copyright 1995, 1996 Versant Object Technology Corporation  
Copyright 1997 Visigenic Software, Inc.  
Copyright 1996 Visual Edge Software, Ltd.

Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE OBJECT MANAGEMENT GROUP, AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND WITH REGARDS TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. The Object Management Group and the companies listed above shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.



This software contains RSA encryption code.

Other company, product, and service names may be trademarks or service marks of others.



---

# Index

## A

- abstract base class
  - inheritance 303
- abstract classes
  - querying 882
- ActiveX clients, building for 562
- adornments
  - around methods 239, 242
  - in files 239, 240
- application
  - DDL files
    - generating 581
- application adaptor 578
- application DDL files
  - defined 602
- application family
  - creating 575
  - DDL files 602
  - install path 604
  - objects, definition 604
  - trace-enabled version 602
- application libraries 602
- applications
  - deploying 602
  - documenting 595
  - existing
    - extending 215
  - packaging 574
  - team environment, packaging
    - in 489
  - testing
    - with QuickTest 611
- attribute
  - foreign key 296
- attribute identity in XML 520
- attribute mapping properties 716
  - data object attributes 716
  - DB schema columns 728
  - key attributes 717
  - mapping helper class 719
  - mapping patterns 722
  - patterned attribute mapping
    - selection 724
  - persistent object attributes 726
- attribute properties
  - Rational Rose, in 113
- attributes 698
  - adding 699
  - deleting 701

- attributes 698 (*continued*)
  - editing 700
  - foreign key 284
  - IDL 698
  - implementation-only 698
  - in implementation,
    - overriding 301
  - private 698
  - protected 698
  - public 698
- automated build process
  - setting up 465

## B

- bean-managed persistence (BMP) 399
- behavior
  - business object 244
  - data object 249
  - in implementation,
    - overriding 301
- binary data encoding scheme 138
- bind steps 138
- BMP (bean-managed persistence) 399
- BMP entity beans 670
  - settings 399
  - workload management 399
- bridging
  - between Rose and Object Builder 70
- build configuration behavior
  - build options 568
  - build targets 567
  - default configuration 566
- build configuration options 688
- build order
  - specifying 558
- build process
  - automated
    - generating code 465
    - setting up 465
- building DLLs 549, 558
  - in a team environment 487
- builds
  - configuring 549
  - specifying location 550
  - specifying order 558
- business object
  - behavior 244

- business object (*continued*)
  - composite 264
  - data object, adding from 784
  - OOSQL implementation methods
    - customizing 765
    - working with 774
- business object application adaptor 249
- business object file
  - creating 775
  - editing 789
- business object implementation 239
  - composite
    - editing 899
    - with data object interface,
      - adding 894
    - deleting 794
    - editing 792
    - methods 751
    - resource methods
      - checkpointResource() 248
      - endResource() 248
      - resetResource() 248
    - with data object implementation
      - adding 780
- business object implementation file
  - editing 793
- business object interface 698
  - adding 777
  - composite
    - adding 892
    - editing 898
  - creating
    - by importing an IDL file 780
  - deleting 794
  - editing 791
  - methods 751
- business object module
  - adding 777
- business object reference attributes 744

## C

- C++ 1
- CB Server service 32
- cbejb command
  - options 670, 673
- change control
  - defined 463

- change control (*continued*)
  - managing a team
    - environment 463
  - managing information 463
  - process
    - setting up 464
  - system 463
  - XML-based 469
- change control system 469
- child component
  - building 380
  - copy helper, creating 300
  - creating 306
  - key, creating 300
  - packaging 381
  - with attributes duplication
    - defining 309
  - with key duplication
    - defining 325
  - with single datastore
    - defining 342
  - with views
    - defining 360
- class properties
  - Rational Rose, in 109
- class relationships
  - Rational Rose, in 119
- classes
  - C++, Java
    - importing 386
- client DLL
  - defining 552
- CMP (container-managed persistence) 396
- CMP entity beans
  - settings 396
    - backend storage type 396
    - bottom-up PAA 396
    - home type 396
    - use wstring in data object 396
  - workload management 396
- CMVC (Change Management Version Control)
  - XML-based change control 471
- code
  - generating 383, 551
    - from command line 684
- collections 291, 581
- color assignments 28
- command line
  - migrating projects 35
- command-line interfaces
  - cbejb 673
- command-line interfaces (*continued*)
  - importidl
    - syntax 667
  - importimpl
    - syntax 683
  - make options 688
  - obcheck
    - syntax 694
  - obexport
    - syntax 661
  - obgen
    - syntax 685
  - obimport
    - syntax 664
  - obmigrate
    - syntax 659
- Compare and Merge Tool for XML 497
  - comparing files with 497
  - merging files with 498
- complex attributes
  - defined 745
  - mapping
    - using the Explode pattern 748
    - using the Primitive pattern 746
  - mapping patterns
    - Explode 745
    - Primitive 745
  - with persistent objects, associating 745
- Component Broker 602
  - development environment 1
  - distributed environment 757
  - frameworks
    - importing 68
    - services 757
- Component Broker frameworks 64
  - QuickTest 611
- Component Broker Server SDK 1
- component layers 458
- components
  - creating 127
    - for existing DB data 139
    - for new DB data 136
    - for PA data 157
    - for transient data 135
  - working with 697
- composers
  - in VisualAge for Java 394
- composite business objects
  - attributes 264
  - defined 264
  - helper objects 264
- composite business objects (*continued*)
  - key 264
  - methods 264
  - working with 891
- composite components
  - creating 262
    - overview 261
  - defined 262
  - kinds of
    - conjunction 262
    - disjunction 262
  - objects, composed of 262
  - packaging 263
- composite keys
  - adding 901
  - defined 265
  - editing 903
  - location of composition
    - components, using for 265
  - working with 900
- composition file
  - creating 885
- compositions
  - adding 886
  - class source files 263
  - composite business objects, creating from 263
  - defined 263
  - editing 889
  - helper objects 263
  - modules
    - adding 886
  - objects, composed of 263
  - restrictions 16
  - working with 884
- configuration directories 602
- Configuration Management Version Control (CMVC) 471
- conjunction composite 262
- connection 601
- consistency checker 487, 565
- constructs
  - constant 770
  - defined 770
  - deleting 774
  - editing 773
  - enumeration 770
  - exception 770
  - Rational Rose, in 101
  - structure 770
  - typedef 770
  - union 770
  - with file scope
    - defining 770

- constructs (*continued*)
  - with interface scope
    - defining 772
  - with module scope
    - defining 771
  - working with 769
- container behavior 595
  - connection 601
  - container policies 597
  - container service 596
  - deployment platforms 423
  - methods, called outside session
    - behavior 600
  - methods, called outside transaction
    - behavior 599
- container instances
  - creating 578
  - deleting 884
  - editing 883
  - working with 883
- container-managed persistence (CMP) 396
- container policies 597
- container service 596
- containers 578, 602
- converters
  - in VisualAge for Java 394
- copy helpers 239, 698
  - adding 830
  - deleting 832
  - editing 831
  - for child component, creating 300
- CORBA
  - data types
    - complex declarators 745
    - constructed types 745
    - template types 745
  - programming model 1
- CORBA IDL 99
- CORBA types, supported 140
- cross-platform development 426
- cross-project applications
  - testing
    - with QuickTest 489
- cross-project dependencies 458, 462
- D**
- data
  - internationalization 132
  - persistent 757
- data access pattern 254
- data definition 137
  - statements 137
- data encoding schemes 152
  - (*continued*)
    - binary 138, 152
    - double-byte character set (DBCS) 138
    - multi-byte character set (MBCS) 152
    - single-byte character set (SBCS) 152
- data inheritance 299
- data object
  - adding
    - from a DB persistent object 814
    - from a PA persistent object 815
  - behavior 249
    - data access pattern 254
    - deployment platforms 423
    - environment 249
    - handle for storing pointers 255
    - type of persistence 251
  - business object, adding
    - from 813
  - with persistent object, associating 811
- data object behavior 249
- data object file
  - creating 797
  - editing 798
- data object implementation 239
  - adding 807
  - deleting 824
  - editing 760, 819, 822
  - methods 751
- data object implementation file
  - editing 823
- data object interface 247, 698
  - adding or selecting later 247
  - creating 801
    - by importing an IDL file 804
  - deleting 824
  - editing 817
  - methods 751
  - with business object, creating 247
- data object module
  - adding 800
- data storage formats
  - binary 138
  - DBCS 138
  - multi-byte character set (MBCS) 138
- data storage formats (*continued*)
  - single-byte character set (SBCS) 138
- data type mappings
  - DB2 142
  - DBCS 149
  - Informix 148
  - Java 159
  - Java to Object Builder 394
  - Oracle 146
- data types
  - tracking, in models 235
  - usage, in model 238
- database data 138
- database metadata 138
- database queries 138, 291
- databases
  - DB2 844
  - Informix 844
  - Oracle 844
- DB (database) persistent objects
  - deleting 839
- DB (database) schemas
  - deleting 859
- DB persistent objects
  - editing 838
  - using push-down methods
    - with 761
- DB schema group
  - deleting 843
  - editing 841
- DB schemas
  - creating
    - by importing an SQL file 844
  - editing 855
- DB2 844
- DB2 precompiler 138
- DBCS data types 735
- DBCS encoding scheme 138
- DDL
  - files
    - objects 604
    - structure 604
  - SQL 137
  - system management 137
- DDL files 603
  - applications 602
  - creating 603
  - generating 593
  - structure 604
- dependencies
  - cross-project 458, 462
  - managing 499
  - IDL files, in 292
- deployment descriptor 402

- deployment library files 402
  - deployment platforms 419, 423, 602
  - design patterns
    - defined 291
    - iterators 291
  - designs
    - Rose, exporting from 80
  - development
    - cross-platform 426
    - Java
      - requirements for 32
    - multi-platform
      - code generation 419
      - constraints 419
      - method implementation 419
      - views 419
  - dialog box
    - Type Usage 238
  - disjunction composite 262
  - distributed query 280, 296
  - DLLs
    - client
      - configuration 565
      - rebuilding 565
  - document type definition (DTD) 492
  - DTD (document type definition) 492
- E**
- EAB (Enterprise Access Builder) 759
  - edited source files
    - importing 385
    - from command line 682
  - EJB class
    - deleting 417
    - editing 416
  - EJB Deployment Tool, the 402
    - cbejb command 670
    - syntax 673
  - EJB JAR file 670
    - deleting 415
    - editing 415
    - introspecting 402
  - EJB JAR files
    - deployed
      - creating 414
      - working with 413
  - embedded SQL 138
  - Enterprise Access Builder (EAB) 159
    - defined 158
    - system
      - managing a team
        - environment 158
  - Enterprise Access Builder (EAB) 159 (*continued*)
    - system (*continued*)
      - managing information 158
  - enterprise beans
    - command line, importing
      - from 670
    - deployed
      - creating 416
      - working with 416
    - deploying 408
      - into polymorphic homes 412
      - using Object Builder 410
      - using the EJB Deployment Tool 411
    - entity
      - CMP (container-managed persistence) 578
    - managing 402
    - Object Builder, importing
      - into 392
      - using Object Builder 401
      - using the EJB Deployment Tool 670
      - using VisualAge for Java 405
      - working with 391
    - environment 249
    - environment variables 32
      - Object Builder's 24
    - existing objects
      - reusing 215
    - Explode mapping pattern 745
    - exported design
      - working with 88
- F**
- factories 291, 581
  - factory name 296
  - file adornments 239
    - adding 240
  - files
    - .cat 79, 98
    - .dll 604
    - .java 618
    - .sqx 138
    - .xmi 69, 79, 89
    - application DDL 604
    - back-up 65
    - batch 541
    - checking in 469, 478, 482
    - checking out 469, 478, 483
    - DDL 604
    - EJB JAR files 413
    - extracting 478, 484
    - frameset 541
  - files (*continued*)
    - generating 602
    - HTML 538
    - HTML (Hypertext Markup Language) 541
    - JAR 167, 177
    - template 383
    - XML 79, 123, 402, 538, 541
      - exported 541
    - XML interchange 492, 493
  - filters
    - available 536
    - creating
      - for viewing Tasks and Objects pane 537
      - creating new 536
  - folders
    - DBA-Defined Schemas 849
    - in Tasks and Objects pane 1
    - non-IDL type object 1, 661
    - User-Defined Compositions 263
  - foreign key
    - attributes 284, 296
    - patterns 284
    - reference 284, 296
    - relationships 284
  - Foreign Key Assistant 296
  - foreign key mapping 735
  - foreign key pattern
    - defining 285
  - framework methods
    - calling 757
    - defined 757
    - editing 757
    - special 757
  - frameworks
    - importing 68
- G**
- generating code 685
    - performance 684, 685
  - get and set methods
    - business object
      - implementation 755
    - copy helpers 755
    - data object implementation 755
    - defined 755
    - editing 756
    - key 755
    - persistent object 755
- H**
- handles
    - for storing pointers 255
  - home 602

- home 602 (*continued*)
  - defined 581
  - instance 581
  - PAA 866
  - polymorphic 581
  - specialized 581
- home to query 296
- homes
  - polymorphic 396
  - queryable 396
  - regular 396
- HTML 538
- I**
- IBM's alphaWorks site 538
- IDL 1, 64, 99
  - importing
    - from command line 666
- IDL (Interface Definition Language) files
  - dependencies within 292
- IDL attributes 698
- IDL file dependencies
  - constructs 292
  - interface 292
  - module 292
- importidl 667
- importimpl 683
- importing
  - non-IDL types 386
- inbound MQSeries data
  - component, creating for 187
- Informix 844
  - data type mappings 148
- inheritance
  - abstract base class 303
  - behavior 299
  - business object
    - implementation 248
  - component objects, recommended
    - for 299
  - data 299
  - data object implementation 257
  - defined 299
  - implementation 299
  - interface 299
  - with attributes duplication 307
  - with key duplication 322
  - with single datastore 341
  - with views 357
- inheritance and overriding
  - in business objects 301
  - in data objects 303
- inheritance patterns
  - attributes duplication 303, 304
  - for persistence 304
- inheritance patterns (*continued*)
  - key duplication 303, 304
  - single datastore with views 304
- initializer method
  - adding 753
- install objects 602
- integration project
  - team environment, adding
    - to 460
- Interface Repository Browser 907
- internationalization
  - data 132
- inverse object reference 284, 296
- IR file name
  - migrating 33
- J**
- J2SDK (the Java 2 SDK) 32
- JAR files
  - building 561
- Java 1
- Java business objects 32
- Java clients, building for 563
- Javadoc comments 239, 242
- K**
- key 239, 698
  - adding 826
  - deleting 829
  - editing 828
  - for child component,
    - creating 300
- key and copy helper
  - inheritance 300
  - overriding 300
- keywords
  - query support 154
- L**
- lazy evaluation
  - using 246
- libraries
  - application 602
- list()
  - OOSQL customization 764
- local-only object
  - adding 219
  - creating
    - by importing an IDL file 669
- local-only object file
  - creating 217
- local-only object module
  - adding 218
- local-only objects 216
- local-only objects, creating 216
- Logical View
  - Class Diagram 98
- M**
- macros 383
- makefiles
  - generating 556
- managed object configuration
  - deleting 875
  - editing 874
- managed object configuration behavior 584
  - home name 584
  - home options 587
- managed object file
  - editing 873
- Managed Object Framework (MOFW)
  - MOFW (Managed Object Framework) 216
- managed objects 578, 581, 698
  - adding 871
  - configuring 588
  - deleting 874
  - editing 872
  - services
    - Session 870
    - Transaction 870
  - working with 869
- mapping
  - attributes
    - using a key 732, 744
    - using a mapping helper 738
    - using the Primitive pattern 731
  - business object
    - to data object 787
  - complex attributes
    - using the Explode pattern 748
    - using the Primitive pattern 746
  - data object
    - to child's persistent object 712
    - to DB persistent object 703
    - to parent's persistent object 711
    - to persistent object 703
  - data object to persistent object 307
  - data objects, to persistent objects 322, 357, 670
  - data objects, to shared persistent object 341

- mapping (*continued*)
    - using foreign key 735
  - mapping helper
    - class 735
    - file
      - default 735
    - methods 735
    - Object Builder, provided by 735
    - providing your own 735
    - sample 149
    - using 735
  - mapping rules
    - Object Builder to Rose 123
    - Rational Rose to Object Builder 97
    - non-project packages 99
  - MBCS encoding scheme 138
  - metadata
    - migrating 33
  - method adornments 239, 242
  - method bodies
    - editing 383
    - external files for 383
  - method mapping properties 767
    - method reordering 769
    - special framework methods 768
    - user-defined methods 768
  - method properties
    - Rational Rose, in 116
  - methods
    - changes, importing 385
    - deleting 767
    - for public attributes 755
    - get 755
    - implementing 752
    - push-down
      - in Enterprise Access Builder 759
      - in Object Builder 759
      - using 759
      - using ECI 759
      - using HOD 759
    - relationship 764
    - reordering 769
    - set 755
    - user-defined
      - defining 751
      - method bodies, providing 751
      - working with 750
  - methods, called outside session
    - behavior 600
  - methods, called outside transaction
    - behavior 599
  - Methods pane 419
  - migrating
    - IR file name 33
    - metadata 33
  - migrating projects
    - from the command line 35
  - Model Consistency Checker 487, 565
    - command line 694
  - model information
    - between projects, exchanging 492
  - model objects 602
    - changing 602
  - modeling tools
    - object-oriented analysis, design Rational Rose 64
  - models
    - consistency, checking for 31, 694
    - secondary
      - propagating changes to 485
      - tracking data types in 235
  - multi-platform development 419
- N**
- naming objects 128
  - non-IDL type object 128, 493
    - folders 1
  - non-IDL types 3
    - folders 536, 661
    - importing 386
  - non-project packages 99
  - null value tolerance 155, 701
    - for foreign keys 155, 298
- O**
- obexport 661
  - obimport 664
  - Object Builder 469, 757
    - command-line usage 659
    - components 1
    - customizing 503
    - defined 1
    - developing in 19
    - environment variables 24
    - panes 1
    - preferences
      - setting 27
    - restrictions 3
    - starting 1
    - tutorials 39
  - Object Builder and Rose
    - bridging guidelines 70
  - Object Builder projects 97
  - Object Builder Tools
    - setting up 24
  - Object Editor 604
  - object references 246
    - foreign key 284
    - foreign key, resolved by 296
    - inverse 284, 296
    - storing 288
  - object relationships 602, 604
  - objects 128
    - forward declaration 604
    - naming 128
    - System Management 128
  - objects to source files mapping 258
  - obmigrate 659
  - OOSQL clauses
    - WHERE 866
  - OOSQL keywords 154
  - operating system platforms
    - for deployment 419
  - Oracle 844
  - OS/390 602
    - environment variables 571
  - outbound MQSeries data
    - component, creating for 201
- P**
- PA (procedural adaptor) bean 159
  - PA bean 158, 698
    - push-down methods
      - exceptions, handling 163
    - query methods
      - customizing 160
  - PA persistent objects 158
    - associating with data object implementation 822
    - deleting 861
    - editing 861
    - push-down methods 301
    - using push-down methods with 762
    - working with 860
  - PA schema 158
    - creating
      - by importing a PA bean 862
  - PA schemas
    - deleting 869
    - editing 868
    - working with 862
  - PAA 768
  - PAA (procedural application adaptor) 759
  - package file
    - creating 138
    - naming 138
    - using 138
  - package properties
    - Rational Rose, in 116



- packages 458
  - Rational Rose, in 98, 99
- panes, Object Builder
  - Methods pane 419
  - Source pane 263, 383, 419
  - Tasks and Objects 263, 536
- pass ticket
  - compositions 571
  - for OS/390 571
  - in RACF 571
  - using 571
- patterns, state data handling
  - Caching 245, 246
  - Delegating 245
  - Same as parent's 245
- performance
  - code compilation using
    - obgen 684, 685
- persistence
  - inheritance patterns 304
  - single datastore, provided
    - by 357
    - type 870
- persistent data 757
- persistent object 239, 698, 745, 811
  - DB schema, adding from 837
  - ESQL framework methods
    - customizing 766
  - PA schema, adding from 860
- persistent object and schema,
  - adding 833
- persistent objects 602
- platform constraints
  - object-specific 419
  - setting 421
- platform differences 425
- platform-specific information 20
- polymorphic homes 581
  - deploying enterprise beans
    - into 412
  - specialized
    - creating 880
- polymorphism 588
- Primitive mapping pattern 745
- problem diagnosis
  - run-time 611
- procedural adaptor (PA)
  - bean
    - importing 159
    - persistent object 159
    - schema 159
- procedural application adaptor
  - (PAA) 759
- processing
  - visual display 611

- profile
  - file 571
    - for remote OS/390 build 571
- project divisions 458
  - changing 496
  - in a team environment 458
- project owner 458
- projects 469, 602
  - creating
    - in a team environment 481
  - deleting
    - in a team environment 486
  - directories 17
  - documenting 501
  - editing
    - in a team environment 485
  - files 17
  - migrating, old 33
  - model name
    - using 17
  - moving 491
  - multiple
    - building 462
  - Object Builder 97
  - organization 17
  - Rational Rose, in 98
  - Rose, importing into 92
  - splitting
    - for team development 459
  - starting 26
- propagating changes
  - to\_secondary models 485

## Q

- queries 138, 284
  - abstract classes 882
  - distributed 280, 296
  - SQL 284
- query method parameters
  - PA bean attributes, mapping
    - to 161
- query methods 866
- query support for keywords 154
- QuickScript 611, 616, 617, 622, 634, 647
  - description 616
  - recording 617
  - running 618
- QuickScript file
  - compiling 618
- QuickTest
  - description 611
  - features 611
  - files
    - generated 647

- QuickTest (*continued*)
  - Framework 621
  - Java and JFC 634
  - programming model, Component
    - Broker 622
  - QuickScript 616
  - tutorial 647, 648
- QuickTest, building for 564
- QuickTest client applications 616
  - generating 614

## R

- RACF (Resource Access Control Facility) 571
- Rational Rose 1, 99, 469
  - attribute properties 113
  - bridging guidelines 70
  - class properties
    - exporting 109
  - class relationships
    - associations and
      - aggregations 119
    - exporting 119
    - inheritance 119
  - classes 64
    - Object Builder classes,
      - mapping to 109
  - constructs 101
  - defined 64
  - editions 64
  - element identifier 89
  - elements
    - attribute 89
    - class 89
    - package 89
    - role of association 89
  - IDL name scoping in 99
  - Logical View 79
  - method properties 116
  - modules 99
  - package properties 116
  - packages 99
  - properties
    - changing 70
  - relationships 64
  - Rose, exporting from 80
  - using 64
  - versions 64
- Rational Rose, using with Object
  - Builder 63
- Rational Rose, working with
  - in team environment 444
- Rational Rose 2000
  - setting up 66

- Rational Rose 98
  - setting up 65
- Rational Rose 98i
  - setting up 66
- Rational Rose design
  - rules 97
  - team environment, exporting to 445
  - team environment, importing from 447
- referential integrity
  - customizing 714
- regression
  - testing 611
- relationship methods
  - add() 764
  - list() 764
  - remove() 764
  - using 764
- relationships
  - 1-n (one-to-many) 284
  - circular
    - defining 283
  - foreign key 284
  - in implementation, overriding 301
  - one-to-many
    - defining 281
  - one-to-one
    - defining 279
- remote build 570
  - launching 572
- Resource Access Control Facility (RACF) 571
- resource methods
  - sessional business object, adding to 164
- restrictions
  - compositions 16
  - Object Builder 3
- Rose Bridge, the
  - design, exporting 79
  - design, re-exporting 69
  - loading Component Broker frameworks 69
  - Object Builder project, exporting design to 69
  - Rose, importing design into 69
- Rose design
  - Object Builder, exporting to 69
  - Object Builder, importing from 69
- Rose model
  - re-exporting 79
- Rose properties
  - changing 70
- S**
- samples
  - change control 473
  - mapping helper 149
  - XSL 541
- SBCS encoding scheme 138
- schema 698, 745
  - column properties 855
  - identification 855
  - structure
    - editing 855
- schema columns 698
- schema groups
  - creating 840
- sentinel values for null 155, 701, 719
- server application
  - adding 576
- server code
  - testing 611
- server DLL 263
  - defining 554
- services
  - Component Broker 757
- session beans 670
  - settings 400
    - state management 400
    - timeout value 400
    - workload management 400
- Session service 248
- sessional business object
  - resource methods, adding to 164
- shared library file 263
- SmartGuide Customizer for XML
  - attribute properties 531
  - constraints 535
  - element properties 530
  - macro setting 530
  - starting 506
- SmartGuides
  - constraints 535
- Source pane 263, 383, 419
  - editing 29
- special framework methods 769
  - del() 758
  - editing 758
  - insert() 758
  - retrieve() 758
  - setConnection() 758
  - update() 758
- specialized homes 581
  - creating 876
    - example 878
  - deleting 879
  - editing 879
  - working with 875
- specialized polymorphic homes
  - creating 880
- SQL clauses
  - using complex relationships in 289
- SQL DDL 137
- SQL files
  - generated
    - editing 857
    - re-importing 847
- SQL queries 284
- SQL View Editor 849
  - notebook pages
    - View Properties 849
    - View Summary 849
    - View Work Area 849
  - working with 850
- state data
  - pattern for handling 245
- structures
  - members 745
  - nested 745
- style sheets
  - XSL 541
- subdirectories
  - Export 89
  - Import 69, 79
  - Model 69, 79, 469
  - XMI 69, 79, 89, 469
- substitution string 866
- syntax
  - WHERE clause 866
- System Management 602
- system management DDL 137
- system management objects 128
- System Manager 602
  - user interface 602
- System Manager User Interface 32
- T**
- tables
  - columns
    - persistent object, mapping to 357
    - parent and child data, identifying in 357
  - rows
    - persistent object, mapping to 357
- Tasks and Objects pane 263, 536

- Tasks and Objects pane 263, 536  
(*continued*)
  - filtering 537
  - searching 30
- team development 444
  - defined 443
  - environment
    - Object Builder project, working with 443
    - Rose package, working with 443
- team environment 469
  - applications, packaging 489
  - building DLLs in 487
  - designs, exporting to 445
  - editing projects 485
  - integration project
    - adding 460
  - maintaining 490
  - migrating 36
  - projects, deleting in 486
  - projects, importing from 447
  - Rational Rose, working with 444
  - setting up 457
  - working in 480
- template design
  - XML 505
- template files 383
- template interpreter format 478
- testing
  - regression 611
  - server code 611
- the Java 2 SDK (J2SDK) 32
- troubleshooting
  - class path environment variable too long 905
  - exception with composite components 905
  - inability to start Object Builder on AIX 905
  - Java Server failure with composite components 905
  - memory problems 905
  - odd behavior 905
- tutorials
  - additional 62
  - component for new DB data, creating 50
  - component for PA data, creating 167, 177
  - component with transient data, creating 39
  - composite component, creating 267

- tutorials (*continued*)
  - inbound message application, creating 188
  - inheritance with attributes
    - duplication 310
  - inheritance with key
    - duplication 327
  - inheritance with single
    - datastore 344
  - inheritance with views 362
  - local-only objects, creating 220
  - multi-platform applications, developing 429
  - Object Builder, for 39
  - outbound message application, creating 203
  - procedural adaptors, unit test for 166
  - QuickTest, running 647
  - remote OS/390 build, launching 573
  - Rose, exporting from 81
  - Rose, importing into 94
  - team development with Rose 449
- type of persistence 251

## U

- Universally Unique Identifier (UUID) 89
- user-defined methods 768
  - code, adding 752
  - editing 754
- user interface
  - System Manager 602
- UUID (Universally Unique Identifier) 89

## V

- view
  - creating
    - with SQL View Editor 850
  - editing 853
  - with SQL View Editor 851
- VisualAge for Java 394

## W

- Web browsers 538
- WHERE clause
  - syntax 866
- wide character set data 735
- with data object, associating 811
- wizards
  - XML 469
    - properties 533, 535

- workload management 396, 399, 400

## X

- XML
  - attribute identity 520
  - export levels 492
  - exporting 387, 492
    - from command line 660
  - ID attributes 521
  - importing 389
    - from command line 663
  - model interchange with 492
  - references 522
- XML-based change control
  - customizing 475
- XML-based change control process
  - XML formats
    - macro format (script format) 469
    - Object Builder's 469
    - template format 469
    - template interpreter format 469
- XML browsing
  - with XSL 538
- XML files
  - browsing 539, 541
- XML interchange files 492, 493
- XML references 522
  - with customized targets 525
- XML template design 505
- XML wizards 469
  - constraining values in 515
  - constraints 535
  - creating 504
  - deriving values in 511
  - distributing 519
  - editing 519
  - in SmartGuide Customizer for XML
    - using 503
  - layout
    - defining 516
  - macros
    - defining 507
  - propagating values in 513
  - properties
    - attribute identity 533
    - constraints 535
  - running 518
  - testing 517
  - value lists
    - customizing 510
- XSL
  - setting up 540

- XSL-based document
  - sample, viewing 543
- XSL processor 538
- XSL sample 541
- XSL style sheets 541
  - applying 547
  - capabilities 538
  - creating your own 546
  - samples
    - applying 544





Printed in the United States of America  
on recycled paper containing 10%  
recovered post-consumer fiber.

SC09-4448-01



Spine information:



WebSphere

Application Development Tools Guide

Version 3.5

SC09-4448-01