IBM WebSphere Business Integration Adapters

Adapter for COM User Guide

Version 1.2.x

IBM WebSphere Business Integration Adapters

Adapter for COM User Guide

Version 1.2.x

Note!
Before using this information and the product it supports, read the information in Notices

25June2004

This edition of this document applies to IBM WebSphere Business Integration adapter for COM, version 1.2.x.

To send us your comments about IBM WebSphere Business Integration documentation, email doc-comments@us.ibm.com. We look forward to hearing from you.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

About this document

The IBM(R) WebSphere(R) Business Integration Adapter portfolio supplies integration connectivity for leading e-business technologies, enterprise applications, legacy, and mainframe systems. The product set includes tools and templates for customizing, creating, and managing components for business process integration.

This document describes the installation, configuration, business object development, and troubleshooting for the IBM WebSphere Business Integration adapter for COM.

Audience

This document is for consultants, developers, and system administrators who support and manage the WebSphere business integration system at customer sites.

Prerequisites for this document

Users of this document should be familiar with the WebSphere business integration system, with business object and collaboration development, and with the COM technology.

Related documents

The complete set of documentation available with this product describes the features and components common to all WebSphere Business Integration Adapters installations, and includes reference material on specific components.

You can install related documentation from the following sites:

- For general adapter information; for using adapters with WebSphere message brokers (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, WebSphere Business Integration Message Broker); and for using adapters with WebSphere Application Server:
  - http://www.ibm.com/websphere/integration/wbiadapters/infocenter
- For using adapters with InterChange Server:
  - http://www.ibm.com/websphere/integration/wicserver/infocenter
  - http://www.ibm.com/websphere/integration/wbicollaborations/infocenter
- For more information about message brokers (WebSphere MQ Integrator Broker, WebSphere MQ Integrator, and WebSphere Business Integration Message Broker):
  - http://www.ibm.com/software/integration/mqfamily/library/manualsa/
- For more information about WebSphere Application Server:
  - http://www.ibm.com/software/webservers/appserv/library.html

These sites contain simple directions for downloading, installing, and viewing the documentation.

Note: Important information about this product may be available in Technical Support Technotes and Flashes issued after this document was published. These can be found on the WebSphere Business Integration Support Web

site, http://www.ibm.com/software/integration/websphere/support/. Select the component area of interest and browse the Technotes and Flashes sections. Additional information might also be available in IBM Redbooks at http://www.redbooks.ibm.com/.

Typographic conventions

This document uses the following conventions:

`courier font`	Indicates a literal value, such as a command name, filename, information that you type, or information that the system prints on the screen.
bold	Indicates a new term the first time that it appears.
italic, *`italic`*	Indicates a variable name or a cross-reference.
blue outline	A blue outline, which is visible only when you view the manual online, indicates a cross-reference hyperlink. Click inside the outline to jump to the object of the reference.
{ }	In a syntax line, curly braces surround a set of options from which you must choose one and only one.
`[ ]`	In a syntax line, square brackets surround an optional parameter.
`...`	In a syntax line, ellipses indicate a repetition of the previous parameter. For example, `option[,...]` means that you can enter multiple, comma-separated options.
`< >`	In a naming convention, angle brackets surround individual elements of a name to distinguish them from each other, as in `<server_name><`*`connector_name`*`>tmp.log`.
\	In this document, backslashes (\) are used as the convention for directory paths. All product pathnames are relative to the directory where the product is installed on your system.
`%text%`	Text within percent (%) signs indicates the value of the Windows™ `text` system variable or user variable.
ProductDir	Represents the directory where the product is installed.

New in this release

New in Release 1.2.x

Updated in June 2004. For version 1.2.x of the adapter for COM, the following items are new in this release:

- The manual has been updated with editorial clarifications.

New in Release 1.1.x

Updated in December 2003. For version 1.1.x of the adapter for COM, the following items are new in this release:

- Adapter installation information has been moved from this guide. See "Install the adapter for COM and related files" on page 13 for the new location of that information.
- Beginning with the 1.1.x version, the adapter for COM is no longer supported on Microsoft Windows NT.

Chapter 1. Overview

The connector for COM is a run time component of the WebSphere Business Integration adapter for COM. The COM Adapter includes a connector, message files, configuration tools, and an Object Discovery Agent (ODA). The connector allows the WebSphere integration broker to exchange business objects with applications, or Common Object Model (COM) components, running on a COM server.

Connectors consist of two components: the connector framework and the application-specific component. The connector framework, whose code is common to all connectors, acts as an intermediary between the integration broker and the application-specific component. The application-specific component contains code tailored to a particular technology (in this case, COM) or application. The connector framework provides the following services between the integration broker and the application-specific component:

- Receives and sends business objects
- Manages the exchange of startup and administrative messages

This document contains information about both the connector framework and the application-specific component. It refers to both of these components as the connector.

Note: The term *connector component* refers to a part of a connector, and should not be confused with the term *COM component*, which refers to the binary software components used in the COM software architecture.

Adapter environment

Before installing, configuring, and using the adapter, you must understand its environment requirements.

Broker compatibility

The adapter framework that an adapter uses must be compatible with the version of the integration broker (or brokers) with which the adapter is communicating. Version 1.2.x of the adapter for COM is supported on the following versions of the adapter framework and with the following integration brokers:

- Adapter framework:
  - WebSphere Business Integration Adapter Framework, version 2.10, 2.2.0, 2.3.0, 2.3.1, 2.4.0
- Integration brokers:
  - WebSphere InterChange Server, version 4.1.1, 4.2.0, 4.2.1, 4.2.2
  - WebSphere MQ Integrator, version 2.1.0
  - WebSphere MQ Integrator Broker, version 2.1.0
  - WebSphere Business Integration Message Broker 5.0
  - WebSphere Application Server Enterprise, version 5.0.2, with WebSphere Studio Application Developer Integration Edition, version 5.0.1

See *Release Notes* for any exceptions.

Note: For instructions on installing the integration broker and its prerequisites, see the following documentation.

- For WebSphere InterChange Server (ICS), see the *System Installation Guide for UNIX* or *for Windows*.
- For message brokers (WebSphere MQ Integrator Broker, WebSphere MQ Integrator, and WebSphere Business Integration Message Broker), see Implementing Adapters with WebSphere Message Brokers, and the installation documentation for the message broker. Some of this can be found at the following Web site: http://www.ibm.com/software/integration/mqfamily/library/manualsa/
- For WebSphere Application Server, see *Implementing Adapters with WebSphere Application Server* and the documentation at http://www.ibm.com/software/webservers/appserv/library.html

Adapter standards

The connector is written to the COM 2.0 specification and as such is compatible with COM applications designed to this standard.

For information about COM, see http://www.microsoft.com/com/tech/com.asp

Adapter platforms

The connector runs on the Windows 2000 platform.

Adapter dependencies

The connector has the following dependencies.

JDK software

The Java Development Kit (JDK), Version 1.3.x, is a prerequisite of installing the adapter for COM. If you do not already have this version of the JDK installed, the WebSphere Business Integration Adapter Framework, Version 2.4 software package (Windows version only) provides a separate installation for the IBM JDK, Version 1.3.1. Note that the IBM JDK, Version 1.3.1 is not installed as part of the WebSphere Business Integration Adapter Framework installation. You must run a separate installation to install the JDK. For details about how to install the JDK from WebSphere Business Integration Adapter Framework, refer to that software package.

COMProxy

The connector uses COMProxy, an interface tool that allows Java programs to communicate with ActiveX objects. This tool generates the Java proxy objects that the connector requires to invoke COM components. The properties, structures, and methods of a COM component are typically defined in a type library file (`.tlb`, `.dll`, `.ole`, `.olb`, or `.exe`). Using the Java Native Interface and COM technology, COMProxy allows a COM component to be treated just like a Java object.

- The connector's startup script ensures that the following COMProxy classes in the package `com.ibm.adapters.utils.comproxy` are included in the classpath at run time:
  - `ActiveXCanvas.class`
  - `COMconstants.class`
  - `ComException.class`
  - `Dispatch.class`
  - `JVariant.class`
  - `OleEnvironment.class`
- The COMProxy C++ run time library (`BIA_COMProxy.dll`) is included in the adapter for interfacing with the COM server. The connector's startup script ensures that this DLL is in the connector's `java.library.path`.
- The connector supports COM objects that implement the Dispatch (OLE Automation) interface type.

Terminology

The following terms are used in this guide:

- **ASI (Application-Specific Information)** Metadata tailored to a particular application or technology. ASI exists at both the attribute and business object level of a business object. See also **Verb ASI**.
- **BO (Business Object)** A set of attributes that represent a business entity (such as Employee) and an action on the data (such as a create or update operation). Components of the WebSphere business integration system use business objects to exchange information and trigger actions.
- **BO (Business Object) handler** A connector component that contains methods that interact with an application and that transforms request business objects into application operations.
- **COM component** The connector interacts with a COM server by processing between a business object and a COM component object. During connector processing, a COM component, which is part of a COM application, is represented in the connector by a proxy object. A *proxy* is a Java class that represents a COM component.
- **COMProxy** The interface tool that allows Java programs to communicate with ActiveX objects. This tool generates the Java proxy objects that the connector requires to invoke COM components. The properties, structures, and methods of a COM component are typically defined in a type library file (`.tlb`, `.dll`, `.ole`, `.olb`, or `.exe`). Using the Java Native Interface and COM technology, COMProxy allows a COM component to be treated just like a Java object.
- **Connection factory** A special kind of proxy object that refers to an application. If the appropriate connector properties are set, the factory object, which is persistent for the life of the connector, can create connections that are placed in the connection pool. The number of connections created depends on the value specified in the `PoolSize` property.

- **Connection object** A special kind of proxy object that is an instance of the connection class. A connection is a reference to an application that can contain state information. For every instance of a connection on the adapter side, there is a corresponding object on the COM side. Connections can be instantiated in batches, retrieved at will, sent back to the connection pool, and be re-used by another thread.
- **Connection pool** A repository used to store and retrieve connection objects.
- **Foreign key** A simple attribute whose value uniquely identifies a child business object. Typically, this attribute identifies a child business object to its parent by containing the child's primary key value. The connector for COM uses the foreign key to specify poolable connection objects.
- **ODA (Object Discovery Agent)** A tool that automatically generates a business object definition by examining specified entities within the application and "discovering" the elements of these entities that correspond to business object attributes. When you install the adapter, the ODA is automatically installed. Business Object Designer provides a graphical user interface to access the ODA and to work with it interactively.
- **Per-call object pool** A programmatic entity for storing objects that need to pass from one method to the next during a single `doVerbFor` method call. Stored objects may be proxy objects or simple attributes.
- **Proxy class** A Java class that represents a COM component class in the connector. The connector creates a proxy object instance of the proxy class name specified in the business object's ASI.
- **Verb ASI (application-specific information)** For a given verb, the verb ASI specifies how the connector should process the business object when that verb is active. It can contain the name of the method to call to process the current request business object.

Architecture of the COM connector

To illustrate the architecture of the connector, this section describes request processing at a high-level, as illustrated in Figure 1 on page 5, and then the details of how the connector works, as illustrated in Figure 2 on page 6.

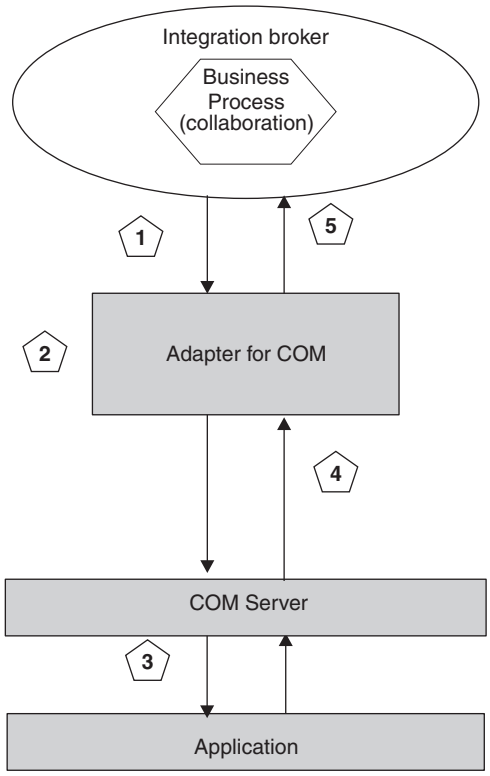


Figure 1. Request processing in the connector for COM

1. The connector receives a business object request from the integration broker.
2. The connector creates a proxy object instance of the business object. The proxy object instance acts as a representation of the COM object to which the connector is sending the request. For details about how the connector creates and processes the proxy object, see "How the connector works."
3. The connector processes the proxy object by using it to invoke the corresponding COM object running on COM server and write data to the COM application.
4. The connector updates the proxy object by reading, or getting, data from COM server object.
5. The adapter returns a message to the integration broker indicating that the original object request was either successful or unsuccessful (a FAIL status). If the request was successful, the connector also returns the updated business object to the broker.

How the connector works

This section describes how the different parts of the connector process a business object, as illustrated in Figure 2 on page 6.

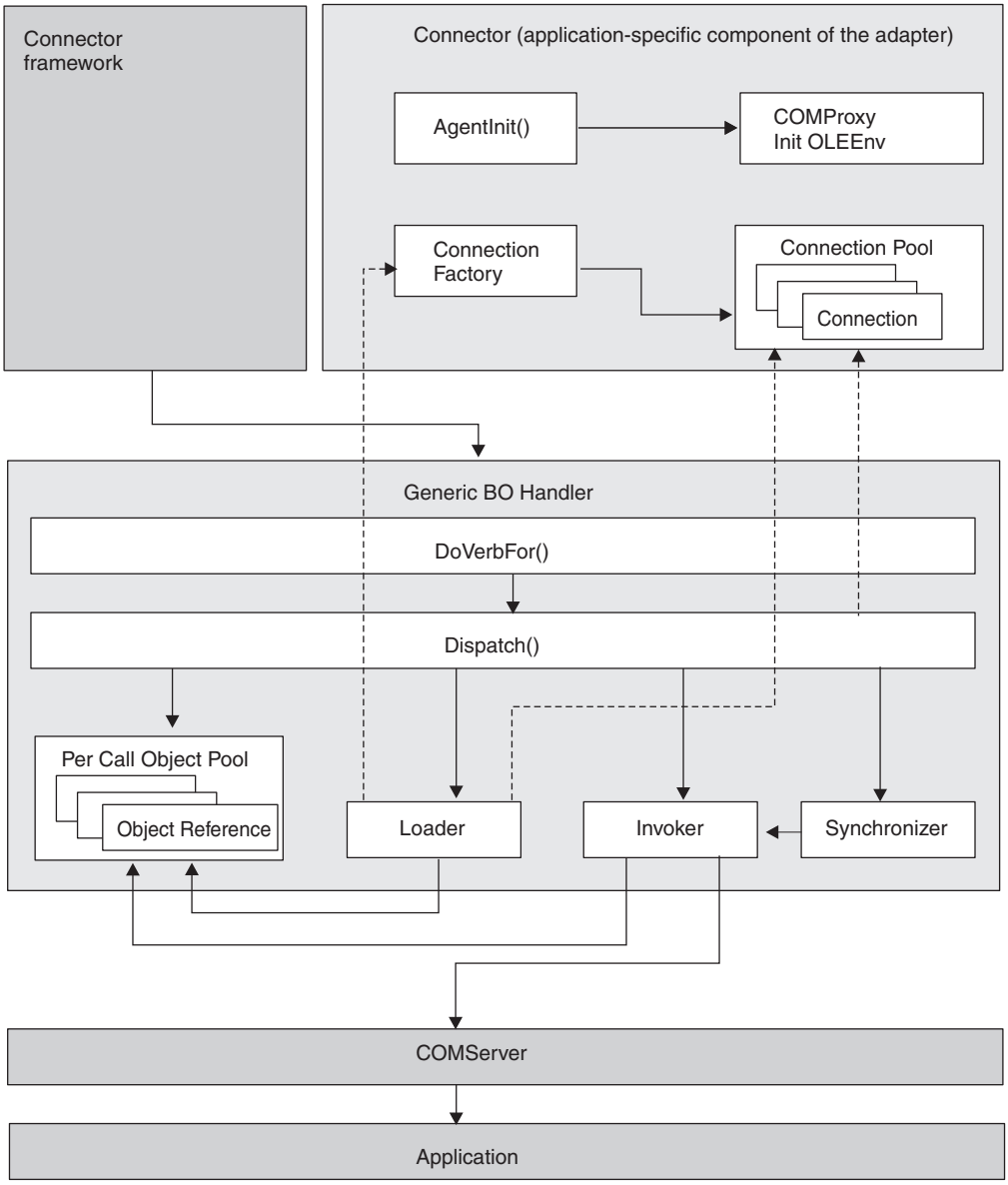


Figure 2. The connector for COM

1. When you first start up the connector, the connector's Agent class performs the following initialization processes:
   - Instantiates the OLE environment.
   - Does one of the following, depending on how the connector properties have been set. For details about the connector properties and how they affect each of the following scenarios, see "Connector-specific properties" on page 16.
     - **Scenario 1**: Creates a connection factory object instance, which is an object that refers to an application. The factory object is persistent for the life of the connector and creates connections that are placed in the connection pool. The number of connections created depends on the value specified in the connector `PoolSize` property.

- **Scenario 2**: Creates connection objects only that are placed in the connection pool. The number of connections depends on the value specified in the `PoolSize` property. No factory object is created in this scenario.
- **Scenario 3**: Creates a factory proxy object against which the business object will call methods (the factory class matches the proxy class ASI of the BO). In this scenario, no connections are created.

2. The integration broker sends a request, in the form of a business object, to the connector.
3. The connector's BO handler receives the object.
4. The `doVerbFor()` method of the BO handler calls the `Dispatch()` method, which reads the BO ASI to obtain the proxy class name. The `Dispatch()` method gets the proxy class name and sends it to the Loader.
5. The Loader uses the proxy class name to load the proxy class (qualified using valid Java class notation, ie. `Mypackage.myclass`) and create a proxy object instance, loading it in the per-call object pool. The Loader checks to see if the object is one of the following:
   - Is it a connection? If so, retrieve it as a connection object from the connection pool.
   - Is it a factory object? If so, retrieve it as a static object from the connection factory.
6. Dispatch then reads through the BO's verb ASI and builds a list of methods. The verb ASI is an ordered list of attribute names. Each attribute represents a method on the proxy object. In other words, the verb ASI is not a list of methods, but a list of attributes, each one having a value that represents a proxy object method.
7. For each method on the verb ASI list, the `InvokeMethods()` method of the BO handler calls `InvokeMethod()` to do one of the following:
   - Call Invoker, if the method is a regular method. If the argument is marked as a foreign key, store it in the per-call object pool. If the attribute is not populated, check the attribute ASI for `use_attribute_value`. If the `use_attribute_value` ASI is present, attempt to pull the object from the per-call object pool.
   - Call the Load (LoadFromProxy function) and Store (WriteToProxy function) operations of Synchronizer (the BO handler's object synchronization process) against all attributes on the proxy object. The operation called depends on what is in the verb ASI. LoadFromProxy (Load) and WriteToProxy (Store) are pre-defined functions that you can include in the verb ASI. Their purpose is to synchronize a business object's simple attributes to a COM component's public properties.
   - Call Load or Store operations against a single, specific attribute (LoadFromProxy gets the proxy property and sets the BO property to that value; WriteToProxy sets the proxy property with values from the BO).

   Note: If the verb ASI is empty, the BO handler will search for a method on the BO with populated parameters and call that. Only one method can have populated parameters. Otherwise, if multiple methods are populated and the verb ASI is empty, then the connector logs an error and returns a FAIL code.
8. For each method of the proxy object, Invoker constructs the parameters and arguments of the method by doing the following:

- If it encounters a BO type (rather than a simple data type, such as a String) in the attribute, Invoker recursively calls the `Dispatch()` method on the active BO handler.
- `Dispatch()` returns a proxy object that the parent method can use to invoke its method call.
- The BO handler's synchronization process, called Synchronizer, invokes WriteToProxy to store (set) a value in each property of the COM component (proxy object), thus updating data on the COM server. The value stored is from the corresponding attribute on the business object that the COM component corresponds to.

9. When values are returned from the COM server, the LoadFromProxy function calls the "getters" of the proxy object and loads the data returned from the proxy object onto the BO.
10. The connector returns the business object to the integration broker.

Business object requests

Business object requests are processed when the integration broker sends a business object to the connector. The only requirement of the business object is that it must map to the corresponding COM component object that the proxy object will represent. The proxy class is a Java class that represents a COM component in the connector. The connector creates a proxy object instance of the proxy class name specified in the business object's ASI.

Verb processing

The connector processes business objects passed to it by a broker based on the verb for each business object.

When the connector framework receives a request from the broker, it calls the `doVerbFor()` method of the business-object-handler class associated with the business object definition of the request business object. The role of the `doVerbFor()` method is to determine the verb processing to perform, based on the active verb of the request business object. It obtains information from the request business object to build and send requests for operations to the application.

When the connector framework passes the request business object to `doVerbFor()`, this method retrieves the business object ASI and invokes the BO handler, which in turn reads the verb ASI and translates it into a series of callable functions. The verb ASI is an ordered list of the methods that need to be called for that verb. The order in which the calls are made is critical to the successful processing of the object.

The connector supports processing multiple components in a single `doVerbFor()`call.

For examples of the sequence of calls that the connector makes during verb processing, see "Sample business object properties" on page 31 and Figure 16 on page 44.

If the verb ASI is blank, the BO handler searches for a method with populated parameters and calls that. Only one method can be populated; otherwise, if multiple methods are populated yet the verb ASI is blank, the connector logs an error and returns a FAIL code. For details about error processing, see "Error handling" on page 49.

The connector does not support any specific verbs, but using the ODA, the user can configure custom verbs.The standard, pre-existing verbs are Create, Retrieve, Update, and Delete. These can be given whatever semantic meaning you provide through the Object Discovery Agent (ODA) running in Business Object Designer. For details about using the ODA to assign a method call sequence to a verb, see Chapter 5, "Creating and modifying business objects," on page 37.

Note: You can specify two pre-defined functions in the verb ASI: `LoadFromProxy` and `WriteToProxy`. Their purpose is to synchronize a business object's simple attributes to a COM component's public properties.

Custom business object handlers

When you create a business object, you can override the default BO handler by specifying the `CBOH` keyword in the BO verb ASI. At connector run time, the `doVerbFor()` method retrieves the business object ASI. If it detects the `CBOH` keyword, `doVerbFor()` invokes the custom BO handler.

A custom BO handler can access proxy classes directly. Therefore, it bypasses the Loader, the Invoker, and the process of retrieving connections from the connection pool. These processes are described in Step 5 on page 7 and Step 7 on page 7.

The connector supports custom BO handlers on parent-level business objects only. For details about creating a custom BO handler, see the *Connector Development Guide*.

Custom BO handler example

The following example of a custom BO handler assumes that a business object has been defined with a verb ASI of `CBOH=comadaptertest.Lotus123BOHandler`. The business object has an attribute ASI of `CellAddress=A1`, where `A1` is the address in which the attribute will appear in the worksheet. The custom BO handler puts the attribute on the worksheet, using the cell address specified by the attribute ASI.

```
package comadaptertest;

import com.crossworlds.cwconnectorapi.*;
import com.crossworlds.cwconnectorapi.exceptions.*;
import java.util.*;
import com.ibm.adapters.utils.comproxy.*;
import lotus123.*;

public class Lotus123BOHandler implements
CWCustomBOHandlerInterface {

        public int doVerbForCustom(CWConnectorBusObj bo) throws
VerbProcessingFailedException {

                Application currentApplication;
                Document currentDocument;

                CWConnectorUtil.traceWrite(CWConnectorUtil.LEVEL4,
"Entering AdapterBOHandler.doVerbFor()");

                try {
                        currentDocument = new Document();

                        // Get the application object.
                        currentApplication = new Application(currentDocument.
get_Parent());

                        // Make the application visible.
```

```
                        currentApplication.set_Visible(new Boolean("true"));
                        currentDocument = new Document(currentApplication.
NewDocument());

                } catch (ComException e) {
                        CWConnectorUtil.generateAndLogMsg(91000,
CWConnectorUtil.XRD_ERROR, CWConnectorUtil.
CONNECTOR_MESSAGE_FILE,
e.getMessage());
                        CWConnectorExceptionObject vSub = new
CWConnectorExceptionObject();
                        vSub.setMsg(e.getMessage());
                        vSub.setStatus(CWConnectorConstant.
APPRESPONSETIMEOUT);
                        throw new VerbProcessingFailedException(vSub);

                }

                //do verb processing on this business object
                dispatch(bo, currentDocument);

                CWConnectorUtil.traceWrite(CWConnectorUtil.
LEVEL4, "Leaving AdapterBOHandler.doVerbFor()");

                return CWConnectorConstant.SUCCEED;
        } //doVerbFor

        private void dispatch(CWConnectorBusObj bo,
Document currentDocument) throws VerbProcessingFailedException {

                CWConnectorUtil.traceWrite(CWConnectorUtil.
LEVEL4, "Entering dispatch");

                CWConnectorUtil.traceWrite(CWConnectorUtil.
LEVEL3, "Processing business object" + bo.getName());

                try {
                        //put this object out onto the spreadsheet.
                        //Follow ASI for Cell addresses
                        businessObjectToWorksheet(bo, currentDocument);
                } catch (ComException e) {
                        CWConnectorUtil.generateAndLogMsg(90001,
CWConnectorUtil.XRD_ERROR, CWConnectorUtil.
CONNECTOR_MESSAGE_FILE, e.getMessage());
                        CWConnectorExceptionObject vSub = new
CWConnectorExceptionObject();
                        vSub.setMsg(e.getMessage());
                        vSub.setStatus(CWConnectorConstant.
APPRESPONSETIMEOUT);
                        throw new VerbProcessingFailedException(vSub);

                } catch (CWException e) {
                        CWConnectorExceptionObject vSub = new
CWConnectorExceptionObject();
                        vSub.setMsg(e.getMessage());
                        vSub.setStatus(CWConnectorConstant.FAIL);
                        throw new VerbProcessingFailedException(vSub);
                }
        }

        public static void businessObjectToWorksheet(CWConnectorBusObj
bo, Document currentDocument) throws CWException {
                String incomingAttribute = "";
                int attrCount = bo.getAttrCount() - 1; //ignore objeventID
                Ranges ranges;
                Range currentRange;
```

```
                ranges = new Ranges(currentDocument.get_Ranges());

                for (int i = 0; i < attrCount; i++) {
                        try {
                                if ((!bo.isObjectType(i)) && (!bo.isIgnore(i))) {
                                        if (bo.isBlank(i))
                                                incomingAttribute = "";
                                        else
                                                incomingAttribute =
bo.getStringValue(i);

                                    String CellAddress = getCellAddress(bo, i);

                                   currentRange = new Range(ranges.Item(new
String(CellAddress)));

                                   currentRange.set_Contents(incomingAttribute);

                                  if (CWConnectorUtil.isTraceEnabled
(CWConnectorUtil.LEVEL5)) {
                                                CWConnectorUtil.traceWrite
(CWConnectorUtil.LEVEL5,
"Application datum from BO to application " + CellAddress + "=" +
incomingAttribute);
                                        }
                                }
                        } catch (AttributeNotFoundException e) {
                                CWConnectorUtil.generateAndLogMsg(91012,
CWConnectorUtil.XRD_ERROR,
 CWConnectorUtil.CONNECTOR_MESSAGE_FILE,
bo.getAttrName(i), bo.getName());
                                throw e;
                        } catch (WrongAttributeException e) {
                                CWConnectorUtil.generateAndLogMsg(91013,
CWConnectorUtil.XRD_ERROR,
 CWConnectorUtil.CONNECTOR_MESSAGE_FILE,
bo.getAttrName(i), bo.getName());
                                throw e;
                        }
                }
        }
        public static String getCellAddress(CWConnectorBusObj bo, int i)
throws CWException {
                String columnName = null;
                try {
                        columnName = getNameFromASI
(bo.getAttrASIHashtable(i, ":"), "CellAddress");
                } catch (WrongASIFormatException e) {
                        CWConnectorUtil.generateAndLogMsg(91014,
CWConnectorUtil.XRD_ERROR,
CWConnectorUtil.CONNECTOR_MESSAGE_FILE, bo.getAttrName(i),
"ColumnName");
                        throw e;
                }

                return columnName;
        }
        private static String getNameFromASI(Hashtable asi, String fieldName)
throws CWException,
WrongASIFormatException {
                String resultName = (String) asi.get(fieldName);

                if (resultName == null || resultName.equals(""))
                        throw new WrongASIFormatException();

                resultName = resultName.toUpperCase();
```

```
                CWConnectorUtil.traceWrite(CWConnectorUtil.LEVEL4, "Found " +
fieldName + " = " + resultName);
                return resultName;
        }

}
```

DCOM support

The Distributed Component Object Model (DCOM) is a protocol that enables software components to communicate directly over a network. It extends COM to support communication among objects on different computers—on a LAN, a WAN, or even the Internet. With DCOM, an application can be distributed across various locations.

The connector provides DCOM support by allowing you to specify the remote server name when you create an object at another location. The remote server name can be specified as a fixed configuration in the system registry of the machine where the connector is running or in the DCOM Class Store. By registering the remote server, the connector has transparent access to the DCOM components.

Processing locale-dependent data

The connector has been internationalized so that it can support delivery of double-byte character sets going into a COM interface that also supports double-byte character sets, and deliver message text in the specified language. When the connector transfers data from a location that uses one character code to a location that uses a different code set, it performs character conversion to preserve the meaning of the data.

The Java run time environment within the Java Virtual Machine (JVM) represents data in the Unicode character code set. Unicode contains encodings for characters in most known character code sets (both single-byte and multibyte). Most components in the WebSphere business integration system are written in Java. Therefore, when data is transferred between most integration components, there is no need for character conversion.

Chapter 2. Installing the adapter

This chapter describes how to install the connector.

Overview of installation tasks

To install the connector for COM, you must perform the following tasks:

Confirm adapter prerequisites

Before you install the adapter, confirm that all the environment prerequisites for installing and running the adapter are on your system. For details, see "Adapter environment" on page 1.

Install the integration broker

Installing the integration broker, a task that includes installing the WebSphere business integration system and starting the broker, is described in the documentation for your broker. For details about the brokers that the connector for COM supports, see "Broker compatibility" on page 1.

For details about installing the broker, see the appropriate implementation documentation of the broker you are using.

Install the adapter for COM and related files

For information on installing WebSphere Business Integration adapter products, refer to the *Installation Guide for WebSphere Business Integration Adapters*, located in the WebSphere Business Integration Adapters Infocenter at the following site:

http://www.ibm.com/websphere/integration/wbiadapters/infocenter

Connector file structure

The Installer copies the standard files associated with the connector into your system.

The utility installs the connector into the *ProductDir*\connectors\COM directory, and adds a shortcut for the connector to the Start menu. Note that *ProductDir* represents the directory where the connector is installed.

Table 1 describes the file structure used by the connector, and shows the files that are automatically installed when you choose to install the connector through the Installer.

Table 1. File structure for the connector

Subdirectory of *ProductDir*	Description
\connectors\COM\BIA_COM.jar	Contains classes used by the COM connector only
\connectors\COM\start_COM.bat	The startup script for the generic connector
\connectors\COM\ext\	A directory where the ODA-generated .jar files can be saved. If you save to this directory, specify the directory in the startup script (start_COM.bat).

Table 1. File structure for the connector (continued)

Subdirectory of *ProductDir*	Description
\connectors\COM\BIA_COMProxy.dll	The COMProxy C++ run time library that defines the properties, structures, and methods of the Java proxies used to invoke COM components. Enables the connector to interface with a COM Server.
\connectors\messages\BIA_COMConnector.txt	Message file for the connector
\ODA\COM\BIA_COMODA.jar	The COM ODA
\ODA\COM\start_COMODA.bat	The ODA startup file
\ODA\COM\BIA_COMProxyGen.exe	Generates the proxy classes that will be used by the connector to create proxy object instances of those classes
\ODA\messages\BIA_COMODAAgent_de_DE.txt	Message file for the ODA (German text strings)
\ODA\messages\BIA_COMODAAgent_en_US.txt	Message file for the ODA (US English text strings)
\ODA\messages\BIA_COMODAAgent_es_ES.txt	Message file for the ODA (Spanish text strings)
\ODA\messages\BIA_COMODAAgent_fr_FR.txt	Message file for the ODA (French text strings)
\ODA\messages\BIA_COMODAAgent_it_IT.txt	Message file for the ODA (Italian text strings)
\ODA\messages\BIA_COMODAAgent_ja_JP.txt	Message file for the ODA (Japanese text strings)
\ODA\messages\BIA_COMODAAgent_ko_KR.txt	Message file for the ODA (Korean text strings)
\ODA\messages\BIA_COMODAAgent_pt_BR.txt	Message file for the ODA (Portuguese - Brazil -text strings)
\ODA\messages\BIA_COMODAAgent_zh_CN.txt	Message file for the ODA (Simplified Chinese text strings)
\ODA\messages\BIA_COMODAAgent_zh_TW.txt	Message file for the ODA (Traditional Chinese text strings)
\repository\COM\BIA_CN_COM.txt	Repository definition for the connector. The default name is BIA_CN_COM.txt.

Note: All product pathnames are relative to the directory where the product is installed on your system.

Post-installation tasks

After you install the adapter, you must configure it before you can run it. For details, see Chapter 3, “Configuring the adapter,” on page 15.

Chapter 3. Configuring the adapter

Overview of configuration tasks

After installation and before startup, you must configure components as described in this section.

Configure the connector

Configuring the connector includes setting up and configuring the connector. For details, see "Configuring the connector."

Configure the business objects

You configure business objects through an ODA (Object Discovery Agent). The ODA enables you to generate business object definitions. A business object definition is a template for a business object. The ODA examines specified application objects, "discovers" the elements of those objects that correspond to business object attributes, and generates business object definitions to represent the information. Business Object Designer provides a graphical interface to access the Object Discovery Agent and to work with it interactively.

For details about using the ODA, see Chapter 5, "Creating and modifying business objects," on page 37.

Configuring the connector

Connectors have two types of configuration properties: standard configuration properties and adapter-specific configuration properties. You must set the values of these properties using Connector Configurator before running the adapter. For further information, see Appendix B, "Connector Configurator," on page 71.

A connector obtains its configuration values at startup. During a run time session, you may want to change the values of one or more connector properties. Changes to some connector configuration properties, such as AgentTraceLevel, take effect immediately. Changes to other connector properties require connector component restart or system restart after a change. To determine whether a property is dynamic (taking effect immediately) or static (requiring either connector component restart or system restart), refer to the Update Method column in the Connector Properties window of the System Manager.

Standard connector properties

Standard connector configuration properties provide information that all adapters use. See Appendix A, "Standard configuration properties for connectors," on page 53 for documentation of these properties.

The following table provides information specific to this connector about standard configuration properties listed in the appendix.

Property	Description
DuplicateEvent Elimination	The connector does not use this property.
Locale	Because this connector has been internationalized, you can change the value of this property.
PollEndTime	The connector does not use this property.
PollFrequency	The connector does not use this property.
PollStartTime	The connector does not use this property.

You must provide a value for the `ApplicationName` configuration property before running the connector.

Connector-specific properties

Connector-specific configuration properties provide information needed by the connector at run time. These properties also provide a way for you to change static information or logic within the connector without having to recode and rebuild it.

To configure connector-specific properties, use Connector Configurator. Click the **Application Config Properties** tab to add or modify configuration properties. For more information, see Appendix B, "Connector Configurator," on page 71.

Note that all the connector-specific properties are optional in that you can choose to set them based on your specific connector configuration requirements: Do you want the connector to create both factory objects and connections, only a factory object, or only connections?

Table 2 lists the connector-specific configuration properties for the connector, along with their descriptions and possible values. The + character indicates the entry's position in the property hierarchy. See the sections that follow for details about the properties, including a representation of the hierarchical relationship of the properties in Figure 3 on page 17.

Table 2. Connector-specific configuration properties

Name	Possible values	Default value
+ Factory	None	None
+ + FactoryClass	*The class name*	None
+ + FactoryInitializer	*Method name of the initializer*	None
+ + + Arguments	*Any encrypted or non-encrypted strings*	None
+ + FactoryMethod	*Method name*	None
+ + + Arguments	*Any encrypted or non-encrypted strings*	None
+ ConnectionPool	None	None
+ + ConnectionClass	*Class name*	None
+ + ConnectionInitializer	*Method name of the initializer*	None
+ + + Arguments	*Any encrypted or non-encrypted strings*	None
+ + PoolSize	*Any integer*	0
+ ThreadingModel	`Apartment`, `Free`	`Free`

Figure 3 illustrates the hierarchical relationship of the connector-specific properties.

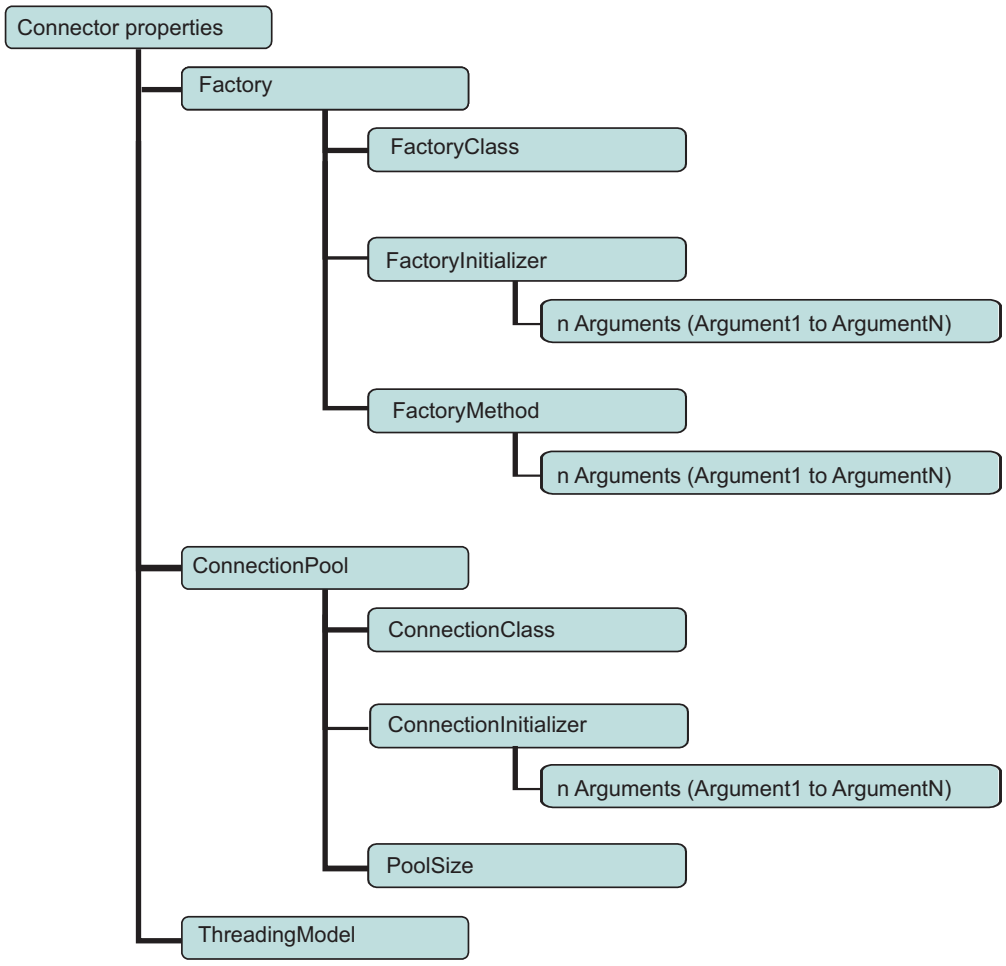


Figure 3. Hierarchy of connector-specific properties

Factory

A hierarchical property that represents the Factory class information.

FactoryClass

The name of the factory class.

- If you specify a `FactoryClass`, a `ConnectionClass`, and a `FactoryMethod`, the connector instantiates a factory proxy object and connections (**Scenario 1** in step 1 on page 6, of How the connector works).
- If you do not specify a `FactoryClass`, then specify a `ConnectionClass`, in which case a connection pool of the specified connection class and size is created when you initialize the connector (**Scenario 2** in step 1 on page 6).
- If you specify a `FactoryClass` only, the connector will instantiate a factory proxy object (**Scenario 3** in step 1 on page 6) and will not use connections.

FactoryInitializer
The method name of the initializer for the `FactoryClass`. This method never acts as a constructor.

Arguments
String values representing the parameters of the FactoryInitializer method. These can be any encrypted or non-encrypted strings.

FactoryMethod
The method name of the FactoryMethod on the FactoryClass. If you specify a FactoryMethod, the connection pool obtains connections using the FactoryMethod. `ConnectionInitializer` is called after a connection object is created. The connection may or may not come from the FactoryMethod.

Arguments
The parameters of the FactoryMethod must be arguments (Argument1, Argument2, and so on) on the Factory, listed in proper sequential order. The property names are `Argument1`, `Argument2`, and so on, for as many parameters as the method takes. The value of each argument is any encrypted or non-encrypted string.

ConnectionPool
The property for the Connection class information.

ConnectionClass
The name of the poolable connection class.

- If you specify a `ConnectionClass` *and* a `FactoryClass`, the connector instantiates a factory proxy object and a connection pool instance is created for storing connections. The connections are created by the Factory. (**Scenario 1** in step 1 on page 6).
- If you specify a `ConnectionClass`, but not a `FactoryClass`, then, when the connector is initialized, a connection pool instance is created for storing connections (**Scenario 2** in step 1 on page 6). In this scenario, the connections are not created by the Factory.

The size of the pool (number of connections) is based on the value you specify in the PoolSize property.

Since a connection is a reference to an application with some kind of state information, note that if you use connection pooling on a single-use server, multiple instances of the application referenced by the connection are created. Each instance is called on a single BO handler thread.

Likewise, if you use connection pooling on a multi-use server (one instance of a server object can be re-used to create a connection), then you have to set up a factory and factory method call to create the connection pool. In this case, each BO handler thread pulls a discrete connection from the pool to be used during processing.

ConnectionInitializer
The name of the poolable ConnectionClass initializer method. This method never acts as a constructor.

`ConnectionInitializer` is called after the connection object is created, whether or not the connection comes from the Factory.

Arguments
String values representing parameters for the initializer. The values may be encrypted or non-encrypted strings.

PoolSize
Determines the size of the connection pool. This property is required if you specify a ConnectionClass.

ThreadingModel
Indicates whether the connector uses the Multi-Threaded Apartment (MTA) or the Single Threaded Apartment (STA) threading model. If this property is set to `Apartment`, the connector runs in STA mode (not threadsafe). STA mode causes the connector to be single-threaded.

If the property is set to `Free`, the connector runs in MTA mode. In MTA mode (threadsafe), multi-threaded clients can make direct calls to the object. The default value is `Free` (threadsafe, Multi-Threaded Apartment mode).

Note: If you are running in Single-Threaded Apartment mode (you have set the property value to `Apartment`), the connection pool and factory are disabled.

Creating multiple connector instances

Creating multiple instances of a connector is in many ways the same as creating a custom connector. You can set your system up to create and run multiple instances of a connector by following the steps below. You must:

- Create a new directory for the connector instance
- Make sure you have the requisite business object definitions
- Create a new connector definition file
- Create a new start-up script

Create a new directory

You must create a connector directory for each connector instance. This connector directory should be named:

```
ProductDir\connectors\connectorInstance
```

where `connectorInstance` uniquely identifies the connector instance.

If the connector has any connector-specific meta-objects, you must create a meta-object for the connector instance. If you save the meta-object as a file, create this directory and store the file here:

```
ProductDir\repository\connectorInstance
```

Create business object definitions

If the business object definitions for each connector instance do not already exist within the project, you must create them.

1. If you need to modify business object definitions that are associated with the initial connector, copy the appropriate files and use Business Object Designer to import them. You can copy any of the files for the initial connector. Just rename them if you make changes to them.
2. Files for the initial connector should reside in the following directory:

   ```
   ProductDir\repository\initialConnectorInstance
   ```

Any additional files you create should be in the appropriate `connectorInstance` subdirectory of `ProductDir\repository`.

Create a connector definition

You create a configuration file (connector definition) for the connector instance in Connector Configurator. To do so:

1. Copy the initial connector's configuration file (connector definition) and rename it.
2. Make sure each connector instance correctly lists its supported business objects (and any associated meta-objects).
3. Customize any connector properties as appropriate.

Create a start-up script

To create a startup script:

1. Copy the initial connector's startup script and name it to include the name of the connector directory:

   dirname
2. Put this startup script in the connector directory you created in "Create a new directory" on page 19.
3. Create a startup script shortcut (Windows only).
4. Copy the initial connector's shortcut text and change the name of the initial connector (in the command line) to match the name of the new connector instance.

You can now run both instances of the connector on your integration server at the same time.

For more information on creating custom connectors, refer to the *Connector Development Guide for C++ or for Java*.

Configuring the startup file

Before you start the connector for COM, you must configure the startup file.

To complete the configuration of the connector for Windows platforms, you must modify the `start_COM.bat` file:

1. Open the `start_COM.bat` file.
2. Scroll to the section beginning with "`SET JCLASSES...`"
3. Edit the `JCLASSES` variable to point to the .jar file created by the ODA. For example, if the .jar file created by the ODA is `c:\WebSphereAdapters\connectors\COM\SampleLotus123.jar`, then set the `JCLASSES` variable to `JCLASSES= c:\WebSphereAdapters\connectors\COM\SampleLotus123.jar;%JCLASSES%`

Starting the connector

A connector must be explicitly started using its **connector start-up script**. The startup script should reside in the connector's runtime directory:

ProductDir`\connectors\`*connName*

where *connName* identifies the connector. The name of the startup script depends on the operating-system platform, as Table 3 shows.

Table 3. Startup scripts for a connector

Operating system	Startup script
UNIX-based systems	`connector_manager_connName`
Windows	`start_connName.bat`

You can invoke the connector startup script in any of the following ways:

- On Windows systems, from the **Start** menu

  Select **Programs>IBM WebSphere Business Integration Adapters>Adapters>Connectors**. By default, the program name is "IBM WebSphere Business Integration Adapters". However, it can be customized. Alternatively, you can create a desktop shortcut to your connector.
- From the command line
  - On Windows systems:

    `start_connName connName brokerName [-cconfigFile ]`
  - On UNIX-based systems:

    `connector_manager_connName -start`

  where *connName* is the name of the connector and *brokerName* identifies your integration broker, as follows:
  - For WebSphere InterChange Server, specify for *brokerName* the name of the ICS instance.
  - For WebSphere message brokers (WebSphere MQ Integrator, WebSphere MQ Integrator Broker, or WebSphere Business Integration Message Broker) or WebSphere Application Server, specify for *brokerName* a string that identifies the broker.

  Note: For a WebSphere message broker or WebSphere Application Server on a Windows system, you *must* include the `-c` option followed by the name of the connector configuration file. For ICS, the `-c` is optional.
- From Adapter Monitor (WebSphere Business Integration Adapters product only), which is launched when you start System Manager

  You can load, activate, deactivate, pause, shutdown or delete a connector using this tool.
- From System Monitor (WebSphere InterChange Server product only)

  You can load, activate, deactivate, pause, shutdown or delete a connector using this tool.
- On Windows systems, you can configure the connector to start as a Windows service. In this case, the connector starts when the Windows system boots (for an Auto service) or when you start the service through the Windows Services window (for a Manual service).

For more information on how to start a connector, including the command-line startup options, refer to one of the following documents:

- For WebSphere InterChange Server, refer to the *System Administration Guide*.
- For WebSphere message brokers, refer to *Implementing Adapters with WebSphere Message Brokers*.
- For WebSphere Application Server, refer to *Implementing Adapters with WebSphere Application Server*.

Stopping the connector

The way to stop a connector depends on the way that the connector was started, as follows:

- If you started the connector from the command line, with its connector startup script:
  - On Windows systems, invoking the startup script creates a separate "console" window for the connector. In this window, type "Q" and press Enter to stop the connector.
  - On UNIX-based systems, connectors run in the background so they have no separate window. Instead, run the following command to stop the connector:

    `connector_manager_`*`connName`* `-stop`

    where *connName* is the name of the connector.
- From Adapter Monitor (WebSphere Business Integration Adapters product only), which is launched when you start System Manager

  You can load, activate, deactivate, pause, shutdown or delete a connector using this tool.
- From System Monitor (WebSphere InterChange Server product only)

  You can load, activate, deactivate, pause, shutdown or delete a connector using this tool.
- On Windows systems, you can configure the connector to start as a Windows service. In this case, the connector stops when the Windows system shuts down.

Using log and trace files

The adapter components provide several levels of message logging and tracing. The connector uses the adapter framework to log error, informational, and trace messages. Error and informational messages are recorded in the log file, and trace messages and their corresponding trace levels (0 to 5) are recorded in a trace file. For details about logging and trace levels, see Chapter 6, "Troubleshooting and error handling," on page 49.

You configure both the log and trace file names, as well as the trace level, in Connector Configurator. For details about this tool, see Appendix B, "Connector Configurator," on page 71.

Note that the ODA has no logging capability. Error messages are sent directly to the user interface. Trace files and the trace level are configured in Business Object Designer. The process is described in "Configure the agent" on page 38. The ODA trace levels are the same as the connector trace levels, defined in "Tracing" on page 50.

Chapter 4. Understanding business objects

This chapter describes the structure of business objects, how the adapter processes the business objects, and the assumptions the adapter makes about them.

The chapter contains the following sections:

Defining metadata

The connector for COM is metadata-driven. In the WebSphere business integration system, metadata is defined as application-specific information that describes a COM application object's data structures. The metadata is used to construct business object definitions, which the connector uses at run time to build business objects.

After installing the connector, but before you can run it, you must create the business objects definitions. The business objects that the connector processes can have any name allowed by the integration broker. For information about naming conventions, see *Naming Components Guide*.

A metadata-driven connector handles each business object that it supports according to the metadata encoded in the business object definition. This enables the connector to handle new or modified business object definitions without requiring modifications to the code. New objects can be created through the Object Discovery Agent (ODA) in Business Object Designer. To modify an existing object, use Business Object Designer directly (without going through the ODA).

Application-specific metadata includes the structure of the business object and the settings of its attribute properties. Actual data values for each business object are conveyed in message objects at run time.

The connector makes assumptions about the structure of its supported business objects, the relationships between parent and child business objects, and the format of the data. Therefore, it is important that the structure of the business object exactly match the structure defined for the corresponding COM object or the adapter will not be able to process business objects correctly.

If you need to make changes to the business object structure, make them to the corresponding object in COM and then export the changes to the type library for input into the ODA.

For more information on modifying business object definitions, see *WebSphere Business Integration Adapters Business Object Development Guide.*

Connector business object structure

The connector processes business objects used by COM components. This section describes the key concepts related to the structure of business objects processed by the COM connector.

Attributes

For each attribute present in a COM component defined in a type library file (.tlb, .dll, .olb, .ole, or .exe), a corresponding business object attribute is generated by the ODA. The type library file contains interfaces, each with methods and properties. It is used by the ODA to compile proxy object definitions.

If an attribute in the COM class is not a simple attribute, and instead is a component, then the BO attribute maps to a child object whose definition matches the corresponding component in the COM object.

Business objects can be flat or hierarchical. A flat business object only contains simple attributes, that is, attributes that represent a single value (such as a string) and do not point to child business objects. A hierarchical business object contains both simple attributes and child business objects or arrays of child business objects that contain attribute values.

A cardinality 1 container object, or single-cardinality relationship, occurs when an attribute in a parent business object contains a single child business object. In this case, the child business object represents a collection that can contain only one record. The attribute type is the child business object.

A cardinality n container object, or multiple-cardinality relationship, occurs when an attribute in the parent business object contains an array of child business objects. In this case, the child business object represents a collection that can contain multiple records. The attribute type is the same as that of the array of child business objects.

Methods

For each method defined in the COM type library file, an attribute is created in the business object. The attribute type is a child BO containing attributes that represent method parameters. The attributes of the child BO appear in the exact same order as the parameters of the COM method. The child BO also has a `Return_Value` attribute, appearing last in the order of arguments, that represents the result of the COM method call. These attributes (of the child BO) can be simple type or object type (complex), depending on the type of the method parameter or return value. The return value is always last in the order of arguments.

Note that the only non-alphanumeric character used in a COM identifier name is the underscore (_), which is resolved to an underscore in the corresponding WebSphere Business Integration business object. However, if the underscore appears at the start of the COM identifier name, which WebSphere Business Integration format does not allow, the ODA resolves the underscore to the string `BBB_`.

Application-specific information

Application-specific information provides the connector with application-dependent instructions on how to process business objects. If you extend or modify

a business object definition, you must make sure that the application-specific information in the definition matches the syntax that the connector expects.

Application-specific information is represented as a name-value pair and can be specified for the overall business object, for each business object attribute, and for each verb.

Business object-level ASI

Object-level ASI provides fundamental information about the nature of a business object and the objects it contains. Table 4 describes the business object-level ASI of business objects that represent proxy objects.

Note: ASI names are not recognized for business objects that represent methods, method parameters, and method return values. For details about business object attributes created for methods of COM objects, see "Methods" on page 24.

Table 4. Object-level ASI

Object-level ASI	Description
`proxy_class=<nameOfProxyClass>`	The name of the proxy class that the business object represents. Use this ASI to map a proxy class to a business object. You must specify this using valid Java Package notation (for example, `java.lang.Vector`).
`auto_load_or_write=true`	Indicates that a business object represents a record structure that is used both as an argument and as a return value. This ASI tells the adapter to write to the proxy object (WriteToProxy) before the function call on arguments for child object arguments, and then read the proxy object (LoadFromProxy) after the function call returns a value.

Verb ASI

Every business object contains a verb. The verb describes how the data in the business object should be handled by the receiving application.

The verb ASI contains a sequence of attribute names, each of which contains a method for the generic business object handler to call. Typically, the method to be invoked belongs to the object itself (versus belonging to a parent of the business object), in which case you specify the method in the object's verb ASI. For example, a component that has the method `IncrementCounter` would require that you specify that method in the corresponding business object's verb ASI.

If the method to be invoked belongs to a parent in the business object hierarchy, then that parent can be referenced by prefixing the method name with the `PARENT` tag.

For example, Figure 4 on page 26 illustrates a business object hierarchy whereby ContactDetails is a child object of Contact, which itself is a child of

PSRCustomerAccount.

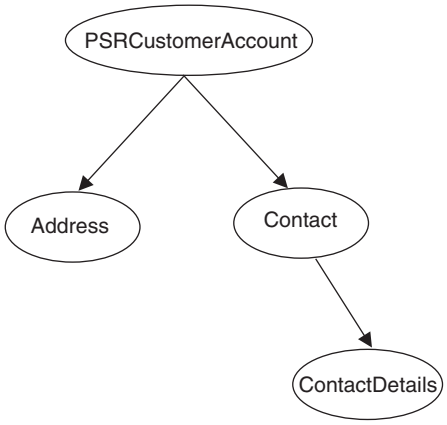


Figure 4. Business object hierarchy and verb ASI

If a method that belongs to PSRCustomerAccount is called on the ContactDetails business object, then the verb ASI for ContactDetails represents the business object hierarchy as follows:

`PARENT.PARENT.<methodName>`

If the method belongs instead to the Contact business object, then the verb ASI for ContactDetails must be set as:

`PARENT.<methodName>`

Note that only methods that belong to parent objects within the hierarchy can be called. Furthermore, a parent business object cannot invoke a child's method.

The connector developer determines the COM operations assigned to the verb. Supported verbs include:

- Create
- Delete
- Retrieve
- Update

The following keywords can be used in the verb ASI sequence of attribute names:

Table 5. Keywords allowed in verb ASI

Keyword	Description
`LoadFromProxy= <attributeName>`	Calls the getter method for the specified proxy object attribute, thus loading the attribute from the proxy to the business object.
`WriteToProxy = <attributeName>`	Calls the setter method for the specified proxy object attribute, thus writing the attribute value from the business object to the corresponding proxy object.
`LoadFromProxy` (no attribute name)	Calls all getter methods for non-method attributes on the current BO from the proxy object.
`WriteToProxy` (no attribute name)	Calls all setter methods for non-method attributes on the current BO to the proxy object.

Table 5. Keywords allowed in verb ASI (continued)

Keyword	Description
CBOH=<*custom BO handler className*>	The class name of a custom BO handler, in cases where the generic BO handler is not used. For information about custom BO handlers, see "Custom business object handlers" on page 9.

For a given object, you can specify the four supported verbs (Create, Retrieve, Delete, and Update) and assign as actions of each verb `n` plus two methods, where `n` equals the number of methods in the corresponding COM component. The two additional methods are those supported by the connector (`LoadFromProxy` and `WriteToProxy`), defined in Table 5 on page 26.

Attribute-level ASI

The attribute-level ASI of a business object can be for complex attributes, which contain child objects, and simple attributes. For a complex attribute, the ASI varies, depending on whether the contained child is a property or a method of an object. The mapping of all the attribute-types in the original COM type library to the ODA-generated business object is defined in Table 9 on page 29.

Table 6 describes the ASI for simple attributes. A simple attribute is always a non-child, for example a boolean, string, or integer value.

Table 6. Attribute-level ASI for simple attributes

Attribute	Description
`Name`	Specifies the business object field name.
`Type`	Specifies the business object field type. See Table 9 on page 29 for details about mapping COM component types to Java proxy types and business object attribute types.
`MaxLength`	Not used.
`IsKey`	Each business object must have at least one key attribute, which you specify by setting the key property to `true` for an attribute. Note that this attribute is used by Business Object Designer, rather than by the connector.
`IsForeignKey`	Specifies that the connector should check whether or not the object must be stored in the per call object pool (see Step 7 on page 7).
`IsRequired`	Not used.
`AppSpecInfo`	Holds the original Java type. This attribute is formatted as follows: `property=<propertyName>, type=<typeName>` `property` is the name of the COM object property. Use this name-value pair to capture the original COM object property name. `type` is the Java type of a simple attribute. See Table 9 on page 29 for details about mapping COM component types to Java proxy types and business object attribute types. This attribute should be set to `proxy` if the attribute is a business object. If it does not map to a business object and is not intended to be de-referenced, as in the case of a simple attribute, you can specify `type=PlaceholderOnly`. This tells the BO handler to not de-reference and not populate the attribute. The attribute can thus continue to be used as part of a multi-call flow if it is marked as a foreign key (`IsForeignKey` is checked), or if `use_attribute_value` is set to a compatible value.

Table 6. Attribute-level ASI for simple attributes (continued)

Attribute	Description
DefaultValue	Not used.

Table 7 describes the ASI for complex attributes containing child objects that are not methods.

Table 7. Attribute-level ASI for attributes containing non-method child objects

Attribute	Description
type	The type of the contained object. Set to proxy if the type is a business object.
ContainedObjectVersion	Not used.
Relationship	Specifies that the child is a container attribute. Set to Containment.
IsKey	Not used
IsForeignKey	Not used
Is Required	Not used
AppSpecificInfo	Holds the original COM application field Name. This attribute is formatted as follows: property=*propertyName*, use_attribute_value=<*(optional)BOName.AttributeName*>, type=<typeName> property is the name of the COM object property. Use this name-value pair to capture the original COM object property name. In the case of an attribute that holds an argument to a method, do not set a value for property, as the argument does not have a name and is simply an argument of any standard type. use_attribute_value is the business object name formatted as *BOName.AttributeName*. Setting this ASI causes the adapter to access the attribute from the per call object pool. Note that this value is not set in the ODA when you create the business object, but rather via Business Object Designer. type is the Java type of a property. This should be set to proxy if the attribute is non-simple, in other words, if it holds a business object (the corresponding COM type is IDispatch*. See Table 9 on page 29 for the mapping of types across COM, Java, and business objects.)
Cardinality	Set to 1.

Table 8 describes the ASI of complex attributes containing child objects that are methods.

Table 8. Attribute-level ASI for attributes containing method child objects

Attribute	Description
Name	The business object field name
type	The business object.
Relationship	Set to Containment, indicating that this is a child object.
IsKey	Set to true if the attribute name equals UniqueName, otherwise it is set to false.
IsForeignKey	Set to false.

Table 8. Attribute-level ASI for attributes containing method child objects (continued)

Attribute	Description
`Is Required`	Set to `false`.
`AppSpecificInfo`	Holds the original COM application field name, which represents the name of the method call placed to the external COM server. This attribute is formatted as: `method_name=<nameOfMethod>`
`Cardinality`	Set to 1.

Note that methods have arguments and return values. Arguments and return values can be complex (containing child objects) or simple.

Mapping attributes: COM, Java, and business object

This section provides a list of the COM types defined in a type library and their corresponding Java constructs and business object attributes. For all business object attributes that are not child business objects, the data type is `String`. In a business object, the ASI holds the actual data type of the attribute and is used when invoking methods against the Java proxy object.

For details about business object ASI, see "Application-specific information" on page 24.

Note: If a COM type is not supported by the COMProxy interface tool, then it is not supported by the connector.

Table 9. Object mapping: COM, JAVA, and business object

COM type	Java primitive	Java boxed	COMProxy internal	Business object	Attribute ASI type=
Float	float	Float	VT_R4	Float	`float`
Float*	float[]	Float[]	VT_R4 | VT_BYREF	Float	`float_refererence`
BSTR	No primitive type exists	String	VT_BSTR	String	`String`
BSTR*	No primitive type exists	String[]	VT_BSTR | VT_BYREF	String	`String_reference`
Int	int	Integer	VT_I4	Integer	`int`
int*	int[]	Int[]	VT_I4 | VT_BYREF	Integer	`int_reference`
IDispatch*	No primitive type exists	Object	VT_DISPATCH	Business object	`proxy`
IDispatch**	No primitive type exists	Object[]	VT_DISPATCH | VT_ARRAY	Business object	`ArrayOf_proxy`
Short	short	Short	VT_I2	Integer	`short`
Short*	short[]	Short[]	VT_I2 | VT_BYREF	Integer	`short_reference`
VARIANT	No primitive type exists	Object	VT_VARIANT	String	`variant`
VARIANT_BOOL	boolean	Boolean	VT_BOOL	Boolean	`boolean`
VARIANT_BOOL*	boolean[]	Boolean[]	VT_BOOL | VT_BYREF	Boolean	`boolean_reference`

Table 9. Object mapping: COM, JAVA, and business object (continued)

COM type	Java primitive	Java boxed	COMProxy internal	Business object	Attribute ASI type=
Long	int	Integer	VT_I4	Integer	`int`
Long*	int[]	Integer	VT_I4 | VT_BYREF	Integer	`int_reference`
CURRENCY	long	Long	VT_CY	Integer	`long`
CURRENCY*	long[]	Long[]	VT_CY | VT_BYREF	Integer	`long_reference`
DATE	No primitive type exists	java.util.Date	VT_DATE	Date	`Date`
DATE*	No primitive type exists	Date[]	VT_DATE | VT_BYREF	Date	`Date_reference`
double	double	Double	VT_R8	Double	`double`
double*	double[]	Double[]	VT_R8 | VT_BYREF	Double	`double_reference`
unsigned char	byte	Byte	VT_UI1	Integer	byte
unsigned char*	byte[]	Byte[]	VT_UI1 | VT_BYREF	Integer	byte_reference
Decimal	No primitive type exists	Not supported	Not supported	Not supported	Not supported
Decimal*	No primitive type exists	Not supported	Not supported	Not supported	Not supported
hyper	No primitive type exists	Not supported	Not supported	Not supported	Not supported
hyper*	No primitive type exists	Not supported	Not supported	Not supported	Not supported
Small	No primitive type exists	Not supported	Not supported	Not supported	Not supported
Small*	No primitive type exists	Not supported	Not supported	Not supported	Not supported
SAFEARRAY(type)	*type[]*	*Type[]*	VT_ARRAY	Cardinality n business object child with single attribute	`ArrayOf_`*`type`*
Enum	int	Integer	VT_INT	Integer	`int`

Note: In cases where the attribute is not intended to be de-referenced, the ASI `type=PlaceholderOnly` should be used. This tells the adapter to not populate this attribute. The attribute may still be used as part of a multi-call flow if it is either marked as a foreign key (`IsForeignKey` is set to `true`), or has the ASI `use_attribute_value` pointing to a compatible attribute.

Array types

Note the following about `array` types:

- To use an `array` type, specify an ASI of `type=ArrayOf_<`*`value`*`>`, where *value* is one of the attribute ASI values listed in Table 9 on page 29. For example, `type=ArrayOf_int` specifies an array of `int` variables. These are mapped to a cardinality n business object that contains the element.

- An Object array (`Object[]`) in Java has a corresponding ASI type of `ArrayOf_proxy`. The processing of proxy objects is done against every element of the array. If the proxy array is an argument to a function, verb processing will occur on every object in the array *before* executing the method. If the array is a return value, verb processing will occur on every object in the array *after* executing the method.
- A sized array may be used as input but not as output.
- A SafeArray is supported as both an input and a return value.

Sample business object properties

This section provides the following examples that illustrate how the adapter processes a business object:

- "Connector call sequence sample"
- "Business object sample"

Connector call sequence sample

The following sample code illustrates connector code that writes a simple message ("Hello World") to WebSphere MQ.

```
/*******************************************
*Create the MQSession object and access the MQQueueManager and (local) MQQueue
*******************************************/
MQSession MQSess = new MQSession();
iDispatch = MQSess.AccessQueueManager("COMTest");
MQQueueManager QMgr = new MQQueueManager(iDispatch);

QMgr.Connect();
MQQueue MyQueue=new MQQueue(QMgr.AccessQueue_4("SYSTEM.DEFAULT.LOCAL.QUEUE",17));

MyQueue.Open();
MQMessage PutMsg = new MQMessage(MQSess.AccessMessage());

//write a string to the message
PutMsg.WriteString("Hello World";);

//put the message on the queue
MyQueue.Put(PutMsg);
MyQueue.Close();
QMgr.Disconnect();
```

Business object sample

The following sample screens illustrate the business object structure and application specific information required for the code in the "Connector call sequence sample" to function properly. The business object illustrated in these sample screens uses the `Create` verb.

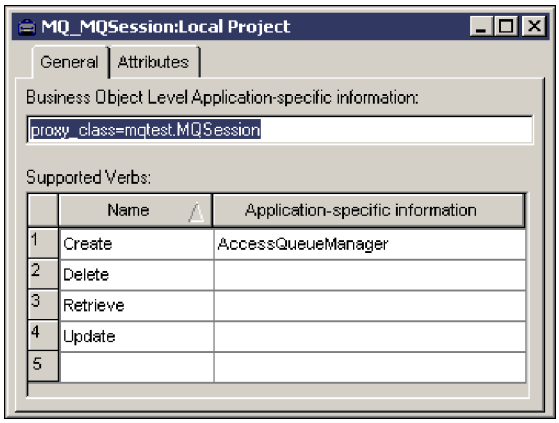


Figure 5. Business object level ASI and supported verbs

Figure 5 illustrates the top-level business object, `MQ_MQSession`, which corresponds to the first object created in the method sequence. The `Create` verb ASI contains `AccessQueueManager`, which is the function that will allow access to an `MQQueueManager` object.

`AccessQueueManager` returns an object of type `Integer`. This object can be passed into the constructor of `MQQueueManager` to create an instance of the `MQQueueManager` proxy.

Note that the business object level ASI contains the string `proxy_class=mqtest.MQSession`, which is the proxy object that represents the COM component for this business object. For a description of the `proxy_class` ASI, see Table 4 on page 25.

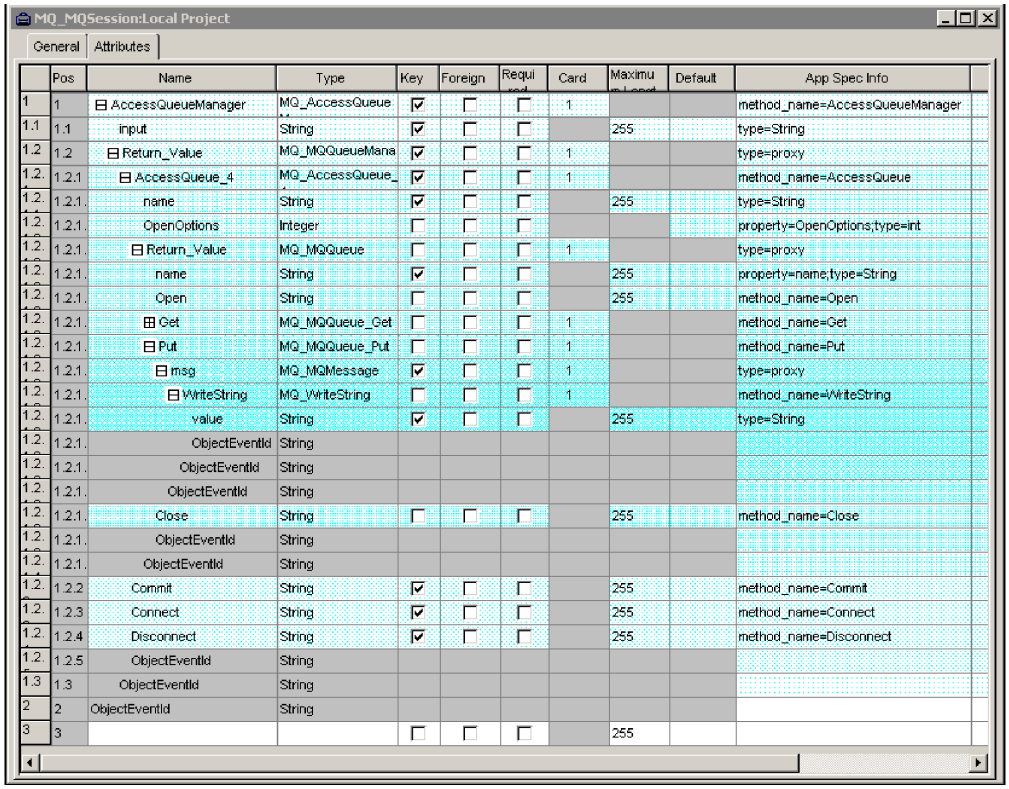


Figure 6. Business object hierarchy for the MQSession object

Figure 6 illustrates the business object hierarchy, from the parent (`MQSession`) all the way down to the lowest-level child.

`AccessQueueManager` is a method that returns an `iDispatch` pointer, which is mapped to `MQQueueManager` (designated as the `Return_Value` attribute in the business object). `MQQueueManager` is a proxy object, represented by the child object `MQ_MQQueueManager` in the business object structure shown in Figure 6.

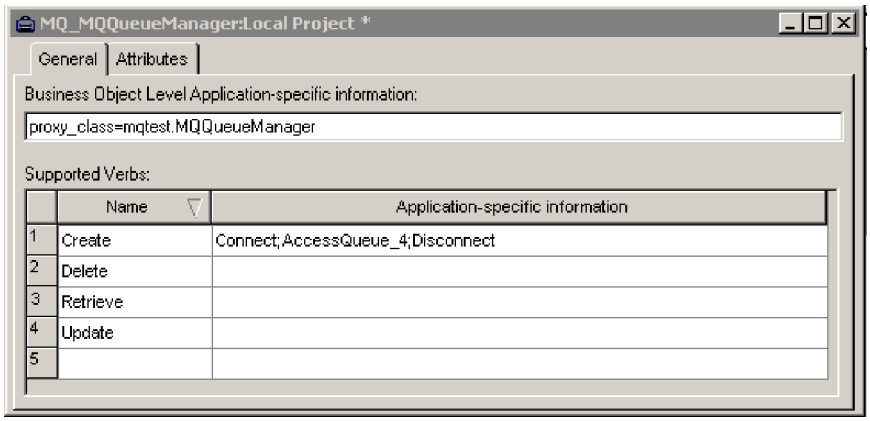


Figure 7. Method call sequence for MQ_MQQueueManager

Figure 7 illustrates the method sequence on the verb `Create` of the business object `MQ_MQQueueManager`.

The sequence is made up of the following three methods:

1. `Connect`
2. `AccessQueue_4`
3. `Disconnect`

Note that in the BO structure illustrated in Figure 6 on page 33, `AccessQueue_4` returns a proxy object, called MQ_MQQueue. Therefore, the connector will process this returned proxy object (which is the child of `AccessQueue_4`) *after* executing `AccessQueue4` and *before* executing `Disconnect`, the third and last method in the call sequence of `MQ_MQQueueManager`.

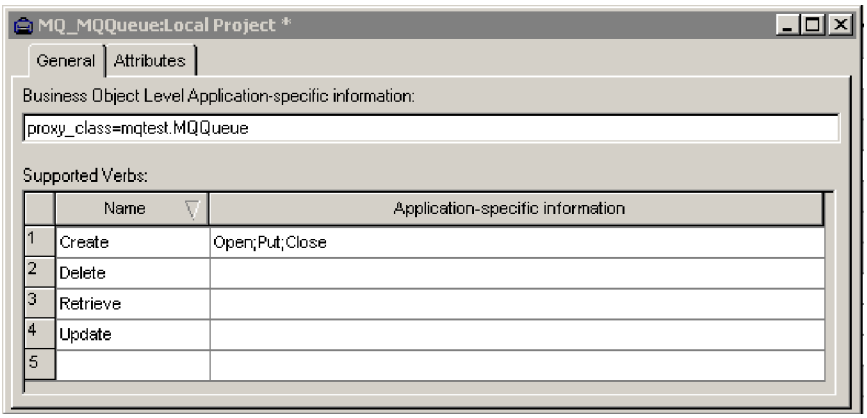


Figure 8. Method call sequence for MQ_MQQueue

Figure 8 illustrates the method sequence for `MQ_MQQueue`, the child of `AccessQueue_4` (Figure 6 on page 33). The call sequence is made up of the following three methods:

1. `Open`
2. `Put`
3. `Close`

The `Put` method takes `MQMessage` as an argument, so the connector must create the `MQMessage` object (and execute methods on it) *before* executing the `Put` method.

Note that the recursive nature of the connector's processing behavior means that `Put` is executed *before* the `Disconnect` method of `MQ_MQQueueManager` (Figure 7 on page 33), the parent of `AccessQueue_4` (Figure 6 on page 33).

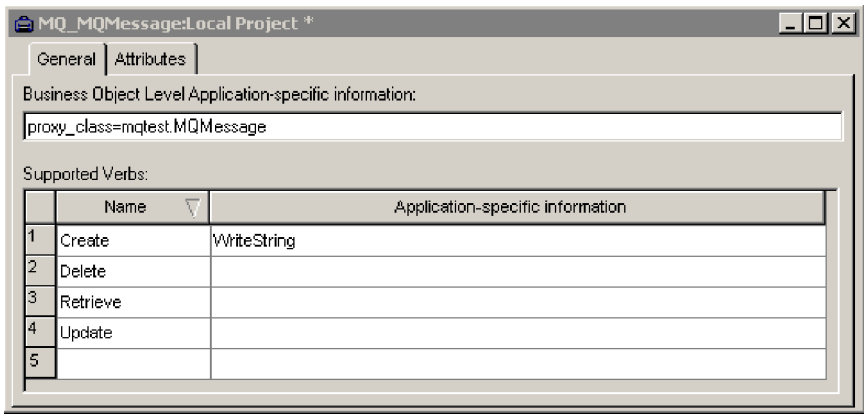


Figure 9. Method call sequence for MQMessage

Figure 9 illustrates the method sequence for `MQMessage`. This sequence calls only the `WriteString` method, which takes a simple string as an argument. In this example, the argument is the "Hello World" message. The message is written to WebSphere MQ and then the connector continues processing methods in the recursive sequence described in this section: after calling `WriteString`, the connector "backtracks" up the hierarchy and executes the `Close` method of `MQ_MQQueue` (Figure 8 on page 34), followed by the `Disconnect` method of `MQ_MQQueueManager` (Figure 7 on page 33).

Generating business objects

Each time an event occurs during run time, a COM application sends a message object containing object-level data and information about the type of transaction. The connector maps this data to the corresponding business object definition, to create an application-specific business object. The connector sends these business objects on to the integration broker for processing. It also receives business objects back from the integration broker, which it passes back to the COM application.

Note: If the object model in the COM application is changed, use the ODA to create a new definition. If the business object definitions in the integration broker repository do not match exactly the data that the COM application sends, the connector is not able to create a business object and the transaction will fail.

Business Object Designer provides a graphical interface that enables you to create and modify business object definitions for use at run time. For details, see Chapter 5, "Creating and modifying business objects," on page 37.

Chapter 5. Creating and modifying business objects

Overview of the ODA for COM

An ODA (Object Discovery Agent) enables you to generate business object definitions. A business object definition is a template for a business object. The ODA examines specified application objects, "discovers" the elements of those objects that correspond to business object attributes, and generates business object definitions to represent the information. Business Object Designer provides a graphical interface to access the Object Discovery Agent and to work with it interactively.

The Object Discovery Agent (ODA) for COM generates business object definitions from metadata contained in COM type library files. The Business Object Designer wizard automates the process of creating these definitions. You use the ODA to create business objects and Connector Configurator to configure the connector to support them. For information about Connector Configurator, see Appendix B, "Connector Configurator," on page 71.

Generating business object definitions

This section describes how to use the COM ODA in Business Object Designer to generate business object definitions. For information on launching and using Business Object Designer, see *IBM WebSphere Business Integration Adapters Business Object Development Guide*.

Starting the ODA

The ODA can be run from any machine that can mount the file system on which the metadata repository (the type library files) resides, using the start_COMODA.bat start file. This file contains start parameters, including the paths to certain required COM and connector .jar files. These .jar files must also be accessible from the machine on which you are running the ODA.

The ODA for COM has a default name of COMODA. The name can be changed by changing the value of the `AGENTNAME` variable in the start script.

To start the ODA, run this command:

```
start_COMODA
```

Note that this startup file requires that the directory of the Java compiler (`javac.exe`), be included in the PATH environment variable. For example, if `javac.exe` is in the directory `c:\jdk131_02\bin`, then include the following line in `start_COMODA.bat`:

```
set PATH=c:\jdk131_02\bin;%PATH%
```

Or, add the string `c:\jdk131_02\bin` in the System PATH variable.

Running Business Object Designer

Business Object Designer provides a wizard that guides you through the steps to generate a business object definition using the ODA.

At any point while using the wizard screens, click **Back** to go to the previous screen, **Next** to go to the next screen, or **Cancel** to cancel the current screen and exit out of the wizard.

The wizard steps are as follows:

Select the agent

To select the agent, follow these steps.

1. Start Business Object Designer.
2. Click **File > New Using ODA**. The *Business Object Wizard - Step 1 of 6 - Select Agent* screen appears.
3. Select the AGENTNAME (specified in the start_COMODA script) appended with the [hostname:port] from the **Located agents** list and click **Next**. (You may have to click **Find Agents** if the desired agent is not listed.)

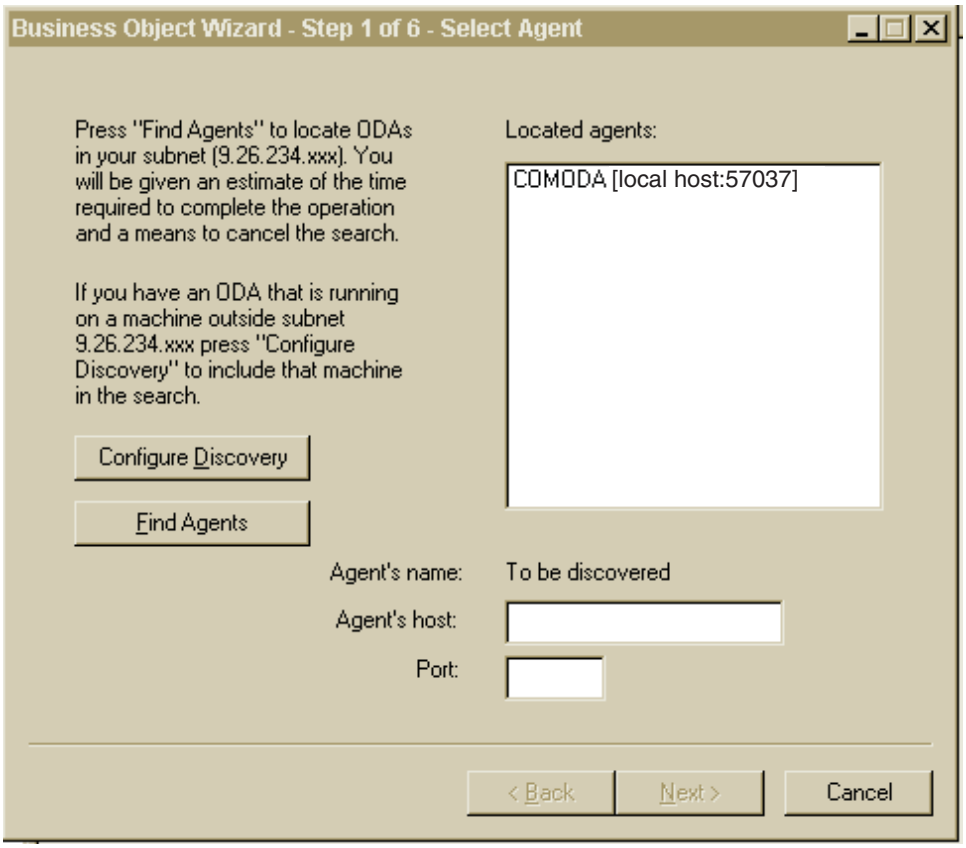


Figure 10. Select Agent Screen

Configure the agent

After you click **Next** on the Select Agent screen, the *Business Object Wizard - Step 2 of 6 - Configure Agent* screen appears. Figure 11 on page 39 illustrates this screen with sample values.

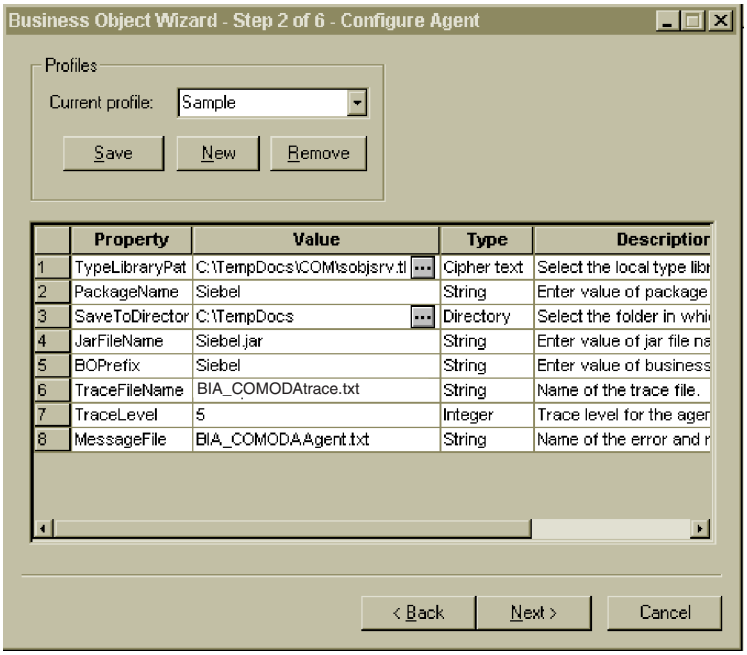


Figure 11. Configure Agent screen

The properties you set on this screen are described in Table 10.

Table 10. Configure Agent properties

Property name	Default value	Type	Description
`TypeLibraryPath`	None	String	(required) The path to the local type library file (.tlb, .dll, .ole, .olb, or .exe) that defines the COM interface. Browse the list of file paths and select the pathname.
`PackageName`	None	String	(required) The package in which all the proxy files generated by COMProxy are stored. COMProxy generates the Java proxy objects that the connector requires to invoke COM components.
`SaveToDirectory`	`None`	String	(required) The directory in which the package specified in `PackageName` is stored. Browse the list of file paths and select the pathname.
`JarFileName`	None	String	(required) The .jar file in which the proxy classes generated by the ODA will be stored. Note that the ODA determines the directory location where this file is placed.
`BOPrefix`	None	String	The prefix that the ODA will add to the names of the business objects it generates.
`TraceFileName`	None	String	The name of the trace message file; the default value is `BIA_COMODAtrace.txt`.
`TraceLevel`	5	Integer	(required) The tracing level (from 0 to 5) for the Agent. For details about tracing levels, see "Tracing" on page 50.

Table 10. Configure Agent properties (continued)

Property name	Default value	Type	Description
`MessageFile`	None	String	(required) The name of the message file that contains all the messages displayed by the ODA. For COM, the name of this file is `BIA_COMODAAgent.txt`. If you do not correctly specify the name of the message file, the ODA will generate the error "Cannot find or read message file."

You can save all the values you enter on the Configure Agent screen (Figure 11 on page 39) to a profile. Instead of retyping the property data next time you run the ODA, you simply select a profile from the drop-down menu and re-use the saved values. You can save multiple profiles, each with a different set of specified values.

1. On the Configure Agent screen (Figure 11 on page 39), click the **New** button in the Profiles group box to create a new profile.
2. Type the value of each property, as defined in Table 10 on page 39.

   Note: If you use an existing profile, the property values are filled in for you, though you can modify the values as needed.
3. Click the **Save** button in the Profiles group box to save the new or updated values.

Select a business object

The *Business Object Wizard - Step 3 of 6 - Select Source* screen appears, as illustrated in Figure 12. The screen lists the components that have been defined in the COM type library file. Use this screen to select any number of COM components for which the ODA will generate business object definitions.

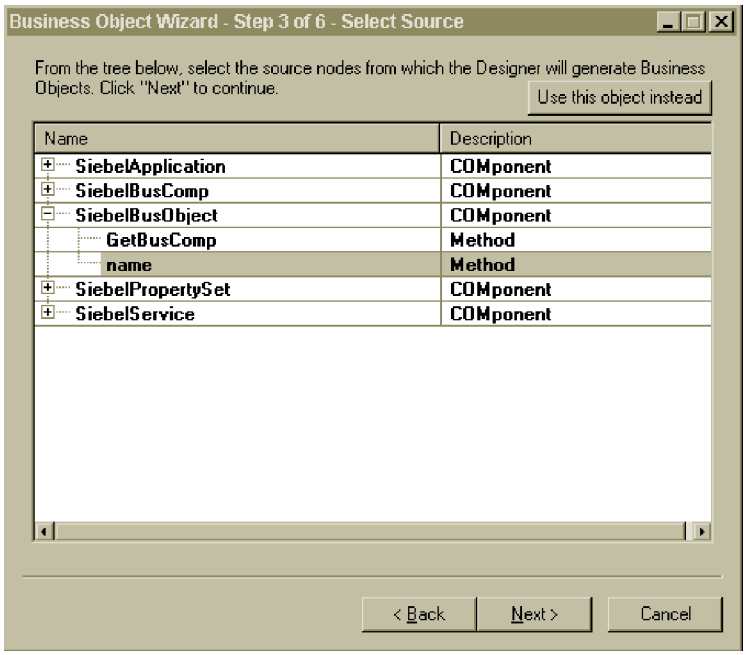


Figure 12. Select Source screen

1. If necessary, expand a COM component to see a list of methods of the component.
2. Select the COM object(s) you want to use. In Figure 12 on page 40, the `name` method is selected.
3. Click **Next** to continue to the next screen in the wizard.

Confirm the object selection

The *Business Object Wizard - Step 4 of 6 - Confirm source nodes for business object definitions* screen appears. It shows the object(s) you selected.

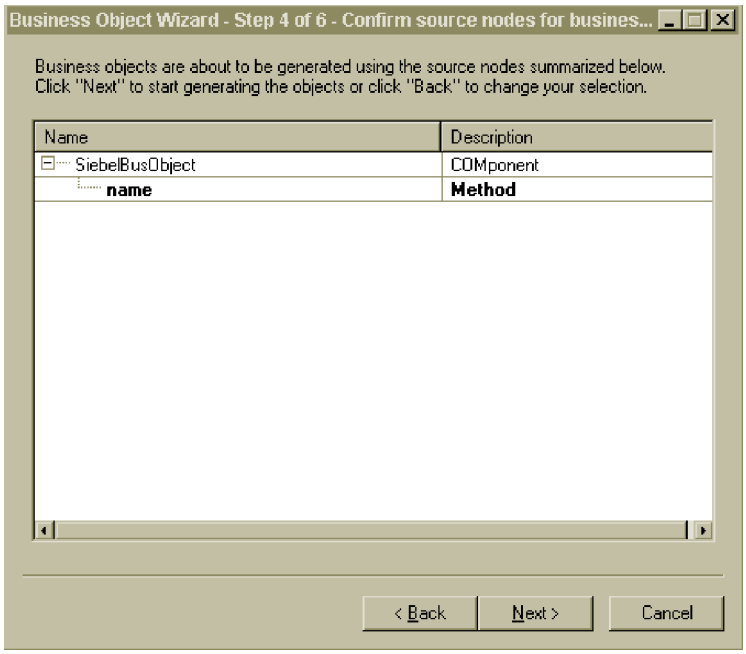


Figure 13. Confirm source node screen

Click **Back** to make changes or **Next** to confirm that the list is correct.

The *Business Object Wizard - Step 5 of 6 - Generating business objects...* screen appears with a message stating that the wizard is generating the business objects.

Note that if you selected a method (see "Select a business object" on page 40), that has a parameter or return value with a Java type of `Object` or `Object[]`, the ODA displays the BO Properties screen, illustrated in Figure 14 on page 42. Use this screen to map an object of such a type to either a COM component or `String`. The drop-down menu in the **Value** column lists only components from the current type library. For a complete list of the mapping of COM and Java types to business object ASI, see Table 9 on page 29.

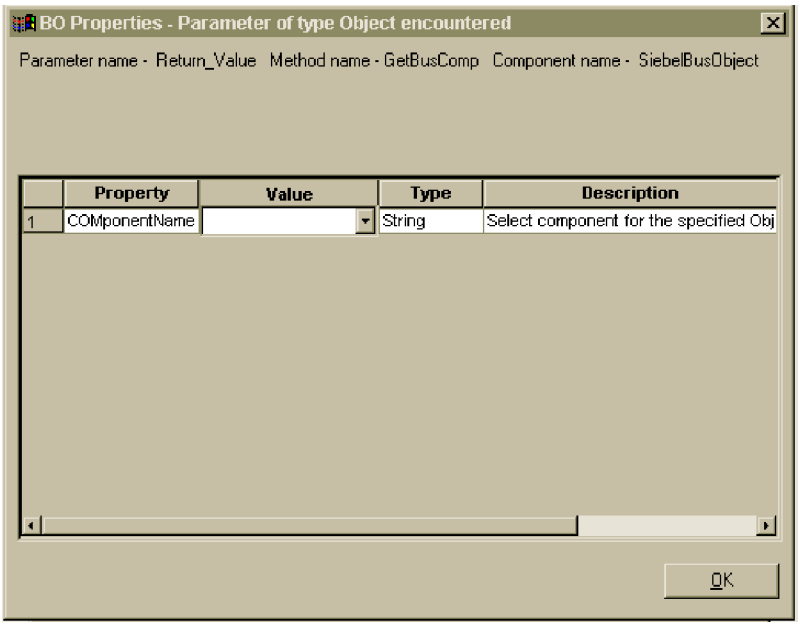


Figure 14. BO Properties screen

The ODA assigns a name to this screen that provides details about the component name, the method name of the component that the ODA is currently processing, and the name of the method parameter whose data type is `Object` or `Object[]`.

Specifying business object information

You can specify the verbs that are valid for the object and the method sequence of a given verb on the object. This section describes how to specify this information, using the ODA with Business Object Designer. For a detailed description of these categories of information and what they mean for business object structure in the COM connector, see Chapter 4, “Understanding business objects,” on page 23.

Selecting verbs

After the BO Properties screen, illustrated in Figure 14, the BO Properties – Select Verbs for component screen appears. Figure 15 on page 43 illustrates this screen for the `name` business object created in Figure 12 on page 40.

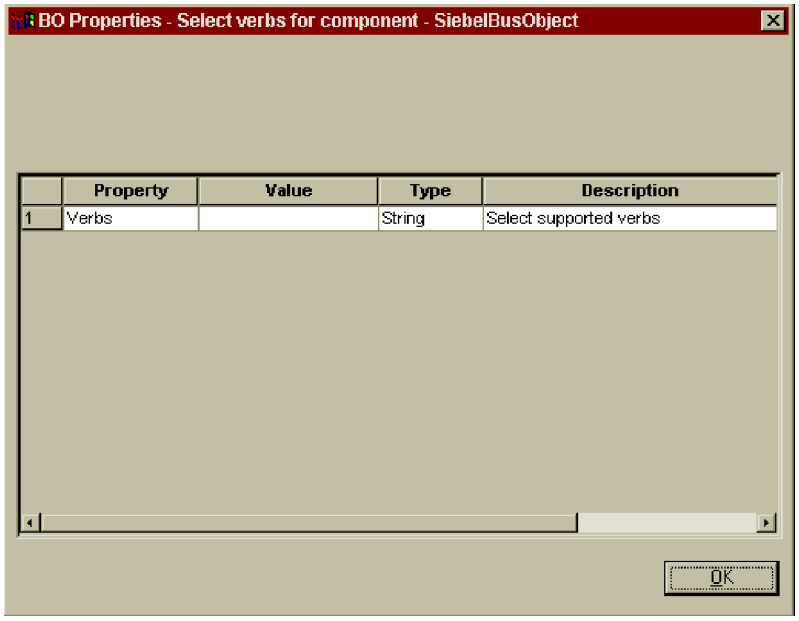
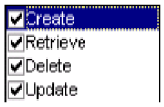


Figure 15. Select verb for component screen

On this screen you specify the verbs that the business objects supports. The ODA allows you to specify the four supported verbs (Create, Retrieve, Delete, and Update). To specify additional verbs beyond the supported four, or to edit verb information after you create a business object, use Business Object Designer.

For details about business object verbs for the COM connector, see "Verb ASI" on page 25

1. In the **Value** list for the Verbs property, select the verbs that you want the business object to support. You can select one or more verbs. You can also deselect a verb at any time.



2. Click **OK**.

Specifying the verb ASI

The ODA allows assigning as actions of each verb `n` plus two methods, where `n` equals the number of methods in the corresponding COM component. The two additional methods are those supported by the connector (`LoadFromProxy` and `WriteToProxy`). For each verb selected in Step 1 of "Selecting verbs" on page 42, a separate window appears where you specify the method sequence that must be executed for the verb.

Figure 16 on page 44 illustrates this screen for the `Create` verb of the `name` business object created in Figure 12 on page 40 and Figure 13 on page 41.

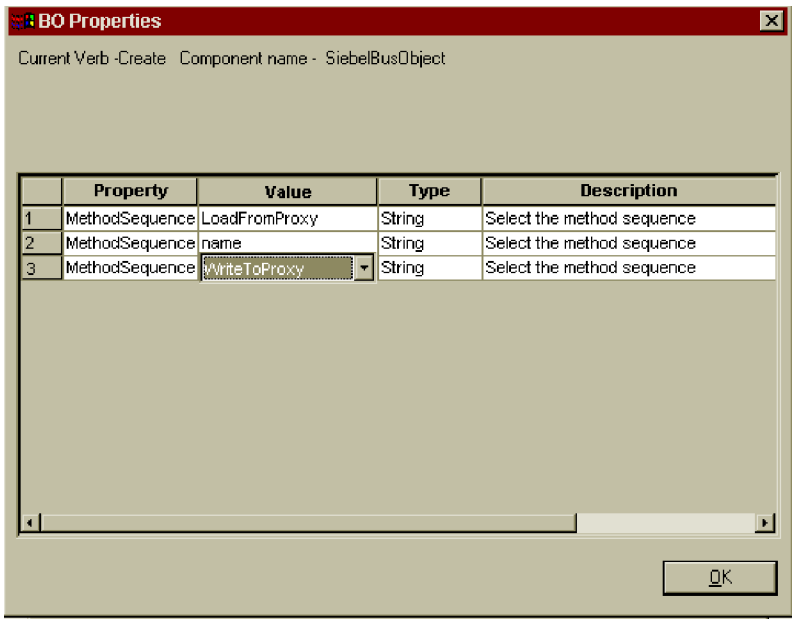


Figure 16. Setting the verb method sequence

1. In the **Value** list for the MethodSequence property, click the method that you want the business object to execute first for the verb. In Figure 16, the method sequence is as follows:
   - The first method that will be executed in the sequence of methods for the `Create` verb is `LoadFromProxy`.
   - The second method in the sequence is `name`.
   - The third method in the sequence is `WriteToProxy`.

   The `name` method is provided by the Siebel business object component (defined in the type library file). The `LoadFromProxy` and `WriteToProxy` methods are provided by the ODA.

   By specifying a method sequence for the verb, you are creating the verb ASI that is associated with that verb. If necessary, this verb ASI can be modified later using Business Object Designer.
2. Click **OK**.

For a list of the keywords supported by the COM verb ASI, see Table 5 on page 26.

Open the business object in a separate window

The *Business Object Wizard - Step 6 of 6 - Save business objects* screen appears.

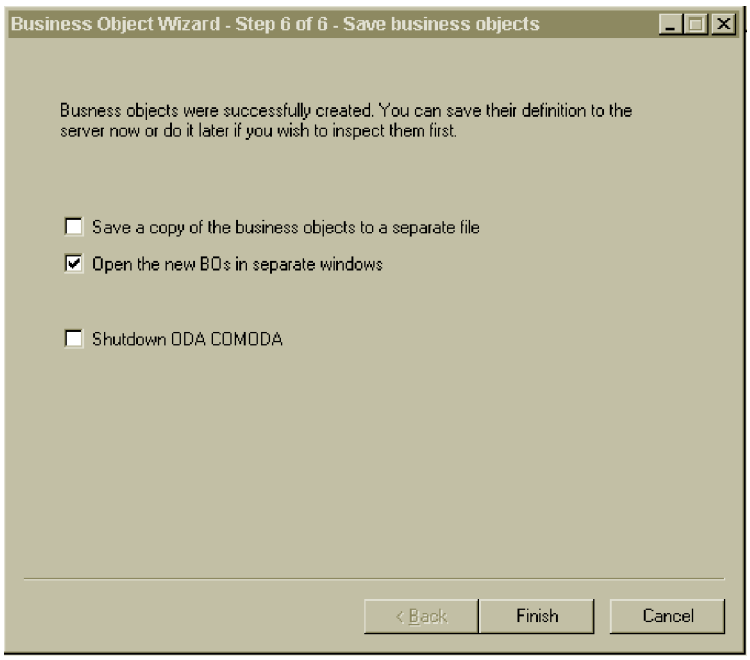


Figure 17. Save business objects screen

You can optionally open the new business objects in separate windows within Business Object Designer, or (after specifying a key for the top-level business object) you can save the generated business object definitions to a file.

To open the business objects in separate windows

1. Select **Open the new BOs in separate windows**.
2. Click **Finish**. Each business object appears in a separate window where you can view and set the ASI information for the business objects and business object verbs you just created. For details, see "Selecting verbs" on page 42 and "Specifying the verb ASI" on page 43.

To save the business objects to a file (only after you specify a key for the parent-level business object, as illustrated in Figure 18 on page 46):

1. Select **Save a copy of the business objects to a separate file**. A dialog box appears.
2. Type or select the location in which you want the copy of the new business object definitions to be saved.

Business Object Designer saves the files to the specified location.

If you have finished working with the ODA, you can shut it down by checking "Shutdown ODA COM ODA" before clicking **Finish**.

Specifying the attribute-level ASI

Business Object Designer displays the attributes for the business object, as illustrated in Figure 18 on page 46. For details about the attribute-level ASI in the COM connector, see "Attribute-level ASI" on page 27.

The attributes are listed on the **Attributes** tab in the order in which they appear in the business object structure, as defined by the numeric value in the **Pos** column.

Simple COM object attributes are represented as simple attributes and their ASI contains the original COM attribute name and type.

For each attribute, the screen provides the name of the attribute, its type, and the ASI information. For example, the `name` attribute of the business object has an ASI that maps the attribute to the original COM component method. In this example, the original method is indicated under the **App Spec Info** column, by the `method_name=name` ASI.

In addition, `name` (a child business object) has the following child object attributes:

- `errCode`, which is a parameter of the original method in the COM type library. This attribute has an ASI of `type`, which in the COM type library file is set to `short_reference`. In the business object, the type is mapped to `Integer`.
- `return_value`, which is used to capture the return value of the `name` method. In the COM type library file, the method is defined as having a return value of type `BSTR`, and in the business object ASI, the type is set to `String`. Note that if a method in the COM type library does not return a value, the `return_value` attribute is not included in the list of business object attributes.

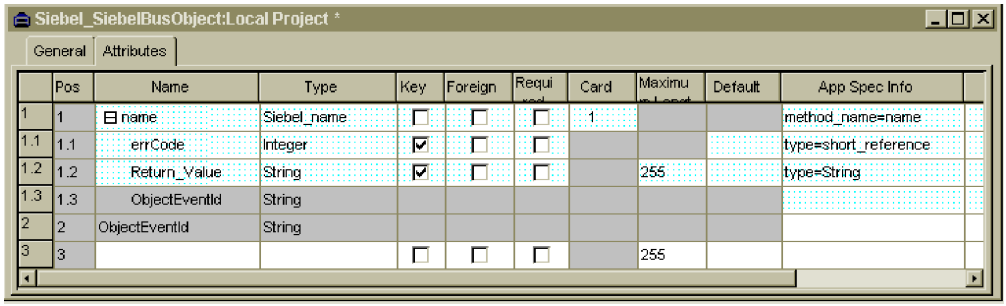


Figure 18. Setting the attribute ASI

On this screen, you should specify whether or not a parent-level object is a key (which is required by the ODA for saving the business objects to a separate file). You can also use this screen to set child object keys as needed and to specify the following information:

- Is the attribute required for the connector to process the business object? If so, click the **Required** check box.
- Is the maximum length of the attribute different from the value that appears in the **Maximum Length** column.
- Does the attribute have a default value? If so, type the value in the **Default** column.

Note: While you can create a business object through the ODA (running in Business Object Designer) and set the parent level key(s), do not configure the foreign key in this manner. The foreign key is non-ASI meta data and therefore must always be configured without the ODA (in Business Object Designer, click **File** > **New** to create a new business object without using the ODA).

Specifying the business object-level ASI

You can view and modify the business object-level ASI. For details about business object-level ASI, see "Business object-level ASI" on page 25.

The business object-level ASI is listed on the **General** tab. The ASI value that appears in the field **Business Object Level Application-specific information** contains the name of the proxy class that represents this business object. The connector uses this information to map a proxy class to a business object.

This screen also lists all the verbs that are supported by the business object and provides the ASI for each verb, as it was defined in "Specifying the verb ASI" on page 43. If a verb ASI is blank, then a method sequence will not be executed for that verb.

Figure 19 illustrates the business object-level ASI for the `name` business object. The only verb that will execute a method sequence for this business object is Create, which has a verb ASI with the method sequence illustrated here (it was originally set in Figure 16 on page 44).

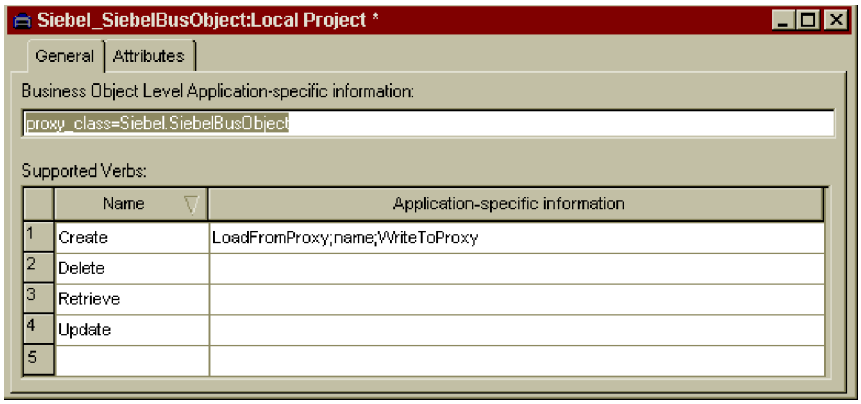


Figure 19. Setting the business object level ASI

On this screen you can modify the ASI of the top-level business object and its supporting verbs. Note that if you open a business object that contains attributes of the child object type, the **General** tab presents the ASI of the top-level business object, which you can modify. If you open a child business object in a separate window, the **General** tab presents the ASI as that of a top-level object.

Uploading business object files

The newly created business object definition files must be uploaded to the integration broker once they have been created. The process depends on whether you are running WebSphere InterChange Server, WebSphere MQ Integrator Broker, or WebSphere Application Server.

- **WebSphere InterChange Server**: If you have saved your business object definition files to a local machine and must upload them to the repository on the server, refer to the InterChange Server implementation documentation.
- **WebSphere MQ Integrator Broker**: You must export the business object definitions out of Business Object Designer and into the integration broker. For details, refer to the implementation documentation of WebSphere MQ Integrator Broker.
- **WebSphere Application Server**: For details, see the implementation documentation of WebSphere Application Server.

Chapter 6. Troubleshooting and error handling

This chapter describes how the adapter for COM handles errors. The adapter generates logging and tracing messages. This chapter describes these messages and provides troubleshooting tips. The chapter contains the following sections:

Error handling

All messages generated by the connector are stored in a message file named `BIA_COMConnector.txt`. (The name of the file is determined by the `LogFileName` standard connector configuration property.) Each message has a message number followed by the message:

```
Message number
Message text
```

The connector handles specific errors as described in the following sections.

COM exception generated by COMProxy

The COMProxy interface tool can generate a variety of errors. For example, if the COM application is down, or the COM call returns a failure, the COMProxy tool throws an exception.

The connector handles such COMProxy exceptions by logging and returning a FAIL code. The HRESULT of the COM call is contained in the COM exception. To aid in debugging, the connector logs the HRESULT, and returns it in the message field of the VerbProcessingFailed exception. The exception also contains information about which call in the sequence failed.

ClassNotFound for proxy

When the Loader receives the proxy class name and tries to create a proxy object of that class, an exception is raised if it cannot find the class. The connector logs the error, which includes the name of the class not found, and returns a FAIL code.

InstantiationException in Loader

When the Loader receives the proxy class name and tries to create a proxy object of that class, an exception is raised if it cannot create the object instance. The connector logs the error, which includes the class name of the object that cannot be instantiated, and returns a FAIL code.

InstantiationException or ClassNotFound during setup of factory or connection pool

A fatal exception is raised if one of the following occurs:

- The Agent Init() method cannot find the Factory class or Connection class specified in the connector's configuration properties.
- The Agent Init() method cannot instantiate a Factory or Connection object of the specified class.

The connector logs the error and returns an APP_RESPONSE_TIMEOUT code.

Illegal AccessException in Loader or Invoker

The connector raises an exception due to invalid code or improper access (public or private) on a method by the COMProxy tool.

The connector logs the error and returns a FAIL code.

NoSuchMethodException in Invoker

The connector raises an exception if a method is specified on the business object that does not exist in the corresponding proxy object.

The connector logs the error and returns a FAIL code.

InvocationTargetException in Invoker

The connector raises an exception when the COM application (with which the connector is exchanging business objects) raises an exception.

The connector logs the error and returns a FAIL code.

Invalid argument (CXIgnore) in a method object in Invoker

The connector raises an exception when a method is included in the business object's verb ASI, but the arguments of that method have not been populated.

The connector logs the error and returns a FAIL code.

Cast failure or wrong attribute type

The connector raises an exception if a proxy object method takes or returns a different data type than what has been specified in the business object.

The connector logs the error and returns a FAIL code.

Invalid verb ASI

The connector raises an exception if the verb ASI of the business object being passed to it is formatted incorrectly or uses improper syntax. Examples of this include a verb ASI that does not contain a proper method sequence, or a child business object that specifies `CBOH` (custom BO handler) for an active verb.

The connector logs the error and returns a FAIL code.

Logging

All errors described in "Error handling" on page 49 must be read from the message file (`BIA_COMConnector.txt`).

Tracing

Tracing is an optional debugging feature you can turn on to closely follow connector behavior. Trace messages, by default, are written to STDOUT. For more on configuring trace messages, see the connector configuration properties in "Configuring the connector" on page 15. For more information on tracing, including how to enable and set it, see the *Connector Development Guide*.

Table 11 lists the recommended content for connector tracing message levels.

Table 11. Tracing messages content

Level	Description
Level 0	Use this level for trace messages that identify the connector version. No other tracing is performed at this level.
Level 1	Use this level for trace messages that: • Provide status information. • Provide key information on each business object processed. • Record each time a polling thread detects a new message in an input queue.
Level 2	Use this level for trace messages that: • Identify the BO handler used for each object that the connector processes. • Log each time a business object is posted to the integration broker. • Indicate each time a request business object is received.
Level 3	Use this level for trace messages that: • Identify the foreign keys being processed, if applicable. These messages appear when the connector has encountered a foreign key in a business object or when the connector sets a foreign key in a business object. • Relate to business object processing. Examples of this include finding a match between business objects, or finding a business object in an array of child business objects.
Level 4	Use this level for trace messages that: • Identify application-specific information. Examples of this include the values returned by the methods that process the application-specific information fields in business objects. • Identify when the connector enters or exits a function. These messages help trace the process flow of the connector. • Record any thread-specific processing. For example, if the connector spawns multiple threads, a message logs the creation of each new thread.
Level 5	Use this level for trace messages that: • Indicate connector initialization. This type of message can include, for example, the value of each connector configurator property that has been retrieved from the broker. • Detail the status of each thread that the connector spawns while it is running. • Represent statements executed in the application. The connector log file contains all statements executed in the target application and the value of any variables that are substituted, where applicable. • Record business object dumps. The connector should output a text representation of a business object before it begins processing (showing the object that the connector receives from the collaboration) as well as after it finishes processing the object (showing the object that the connector returns to the collaboration).

Appendix A. Standard configuration properties for connectors

This appendix describes the standard configuration properties for the connector component of WebSphere Business Integration adapters. The information covers connectors running on the following integration brokers:

- WebSphere InterChange Server (ICS)
- WebSphere MQ Integrator, WebSphere MQ Integrator Broker, and WebSphere Business Integration Message Broker, collectively referred to as the WebSphere Message Brokers (WMQI).
- WebSphere Application Server (WAS)

Not every connector makes use of all these standard properties. When you select an integration broker from Connector Configurator, you will see a list of the standard properties that you need to configure for your adapter running with that broker.

For information about properties specific to the connector, see the relevant adapter user guide.

Note: In this document, backslashes (\) are used as the convention for directory paths. For UNIX installations, substitute slashes (/) for backslashes and follow the conventions for each operating system.

New and deleted properties

These standard properties have been added in this release.

New properties

- XMLNameSpaceFormat

Deleted properties

- RestartCount

Configuring standard connector properties

Adapter connectors have two types of configuration properties:

- Standard configuration properties
- Connector-specific configuration properties

This section describes the standard configuration properties. For information on configuration properties specific to a connector, see its adapter user guide.

Using Connector Configurator

You configure connector properties from Connector Configurator, which you access from System Manager. For more information on using Connector Configurator, refer to the sections on Connector Configurator in this guide.

Note: Connector Configurator and System Manager run only on the Windows system. If you are running the connector on a UNIX system, you must have a Windows machine with these tools installed. To set connector properties

for a connector that runs on UNIX, you must start up System Manager on the Windows machine, connect to the UNIX integration broker, and bring up Connector Configurator for the connector.

Setting and updating property values

The default length of a property field is 255 characters.

The connector uses the following order to determine a property's value (where the highest number overrides other values):

1. Default
2. Repository (only if WebSphere InterChange Server is the integration broker)
3. Local configuration file
4. Command line

A connector obtains its configuration values at startup. If you change the value of one or more connector properties during a run-time session, the property's **Update Method** determines how the change takes effect. There are four different update methods for standard connector properties:

- **Dynamic**
  The change takes effect immediately after it is saved in System Manager. If the connector is working in stand-alone mode (independently of System Manager), for example with one of the WebSphere message brokers, you can only change properties through the configuration file. In this case, a dynamic update is not possible.
- **Agent restart (ICS only)**
  The change takes effect only after you stop and restart the application-specific component.
- **Component restart**
  The change takes effect only after the connector is stopped and then restarted in System Manager. You do not need to stop and restart the application-specific component or the integration broker.
- **Server restart**
  The change takes effect only after you stop and restart the application-specific component and the integration broker.

To determine how a specific property is updated, refer to the **Update Method** column in the Connector Configurator window, or see the Update Method column in Table 12 on page 55 below.

Summary of standard properties

Table 12 on page 55 provides a quick reference to the standard connector configuration properties. Not all the connectors make use of all these properties, and property settings may differ from integration broker to integration broker, as standard property dependencies are based on `RepositoryDirectory`.

You must set the values of some of these properties before running the connector. See the following section for an explanation of each property.

Note: In the "Notes"column in Table 12 on page 55, the phrase "Repository directory is REMOTE" indicates that the broker is the InterChange Server. When the broker is WMQI or WAS, the repository directory is set to LOCAL

Table 12. Summary of standard configuration properties

Property name	Possible values	Default value	Update method	Notes
AdminInQueue	Valid JMS queue name	*CONNECTORNAME* /ADMININQUEUE	Component restart	Delivery Transport is JMS
AdminOutQueue	Valid JMS queue name	*CONNECTORNAME*/ADMINOUTQUEUE	Component restart	Delivery Transport is JMS
AgentConnections	1-4	1	Component restart	Delivery Transport is MQ or IDL: Repository directory is <REMOTE> (broker is ICS)
AgentTraceLevel	0-5	0	Dynamic	
ApplicationName	Application name	Value specified for the connector application name	Component restart	
BrokerType	ICS, WMQI, WAS		Component restart	
CharacterEncoding	ascii7, ascii8, SJIS, Cp949, GBK, Big5, Cp297, Cp273, Cp280, Cp284, Cp037, Cp437 **Note:** This is a subset of supported values.	ascii7	Component restart	
ConcurrentEventTriggeredFlows	1 to 32,767	1	Component restart	Repository directory is <REMOTE> (broker is ICS)
ContainerManagedEvents	No value or JMS	No value	Component restart	Delivery Transport is JMS
ControllerStoreAndForwardMode	true or false	true	Dynamic	Repository directory is <REMOTE> (broker is ICS)
ControllerTraceLevel	0-5	0	Dynamic	Repository directory is <REMOTE> (broker is ICS)
DeliveryQueue		*CONNECTORNAME*/DELIVERYQUEUE	Component restart	JMS transport only
DeliveryTransport	MQ, IDL, or JMS	JMS	Component restart	If Repository directory is local, then value is JMS only

Table 12. Summary of standard configuration properties (continued)

Property name	Possible values	Default value	Update method	Notes
DuplicateEventElimination	`true or false`	`false`	Component restart	JMS transport only: Container Managed Events must be <NONE>
FaultQueue		*CONNECTORNAME*/FAULTQUEUE	Component restart	JMS transport only
jms.FactoryClassName	`CxCommon.Messaging.jms .IBMMQSeriesFactory` or `CxCommon.Messaging .jms.SonicMQFactory` or any Java class name	`CxCommon.Messaging. jms.IBMMQSeriesFactory`	Component restart	JMS transport only
jms.MessageBrokerName	If FactoryClassName is `IBM`, use `crossworlds.queue. manager`. If FactoryClassName is `Sonic`, use `localhost:2506`.	`crossworlds.queue.manager`	Component restart	JMS transport only
jms.NumConcurrentRequests	Positive integer	`10`	Component restart	JMS transport only
jms.Password	Any valid password		Component restart	JMS transport only
jms.UserName	Any valid name		Component restart	JMS transport only
JvmMaxHeapSize	Heap size in megabytes	`128m`	Component restart	Repository directory is <REMOTE> (broker is ICS)
JvmMaxNativeStackSize	Size of stack in kilobytes	`128k`	Component restart	Repository directory is <REMOTE> (broker is ICS)
JvmMinHeapSize	Heap size in megabytes	`1m`	Component restart	Repository directory is <REMOTE> (broker is ICS)
ListenerConcurrency	`1- 100`	`1`	Component restart	Delivery Transport must be `MQ`
Locale	`en_US`, `ja_JP`, `ko_KR`, `zh_CN`, `zh_TW`, `fr_FR`, `de_DE`, `it_IT`, `es_ES`, `pt_BR` **Note:** This is a subset of the supported locales.	`en_US`	Component restart	

Table 12. Summary of standard configuration properties (continued)

Property name	Possible values	Default value	Update method	Notes
LogAtInterchangeEnd	`true` or `false`	`false`	Component restart	Repository Directory must be <REMOTE> (broker is ICS)
MaxEventCapacity	`1-2147483647`	`2147483647`	Dynamic	Repository Directory must be <REMOTE> (broker is ICS)
MessageFileName	Path or filename	`CONNECTORNAMEConnector.txt`	Component restart	
MonitorQueue	Any valid queue name	*`CONNECTORNAME`*`/MONITORQUEUE`	Component restart	JMS transport only: DuplicateEvent Elimination must be true
OADAutoRestartAgent	`true` or `false`	`false`	Dynamic	Repository Directory must be <REMOTE> (broker is ICS)
OADMaxNumRetry	A positive number	`1000`	Dynamic	Repository Directory must be <REMOTE> (broker is ICS)
OADRetryTimeInterval	A positive number in minutes	`10`	Dynamic	Repository Directory must be <REMOTE> (broker is ICS)
PollEndTime	`HH:MM`	`HH:MM`	Component restart	
PollFrequency	A positive integer in milliseconds `no` (to disable polling) `key` (to poll only when the letter `p` is entered in the connector's Command Prompt window)	`10000`	Dynamic	
PollQuantity	`1-500`	`1`	Agent restart	JMS transport only: Container Managed Events is specified
PollStartTime	`HH:MM`(HH is 0-23, MM is 0-59)	`HH:MM`	Component restart	

Table 12. Summary of standard configuration properties (continued)

Property name	Possible values	Default value	Update method	Notes
RepositoryDirectory	Location of metadata repository		Agent restart	For ICS: set to <REMOTE> For WebSphere MQ message brokers and WAS: set to C:\crossworlds\ repository
RequestQueue	Valid JMS queue name	*CONNECTORNAME*/REQUESTQUEUE	Component restart	Delivery Transport is JMS
ResponseQueue	Valid JMS queue name	*CONNECTORNAME*/RESPONSEQUEUE	Component restart	Delivery Transport is JMS: required only if Repository directory is <REMOTE>
RestartRetryCount	0-99	3	Dynamic	
RestartRetryInterval	A sensible positive value in minutes: 1 - 2147483547	1	Dynamic	
RHF2MessageDomain	mrm, xml	mrm	Component restart	Only if Delivery Transport is JMS and WireFormat is CwXML.
SourceQueue	Valid WebSphere MQ name	*CONNECTORNAME*/SOURCEQUEUE	Agent restart	Only if Delivery Transport is JMS and Container Managed Events is specified
SynchronousRequestQueue		*CONNECTORNAME*/ SYNCHRONOUSREQUESTQUEUE	Component restart	Delivery Transport is JMS
SynchronousRequestTimeout	0 - any number (millisecs)	0	Component restart	Delivery Transport is JMS
SynchronousResponseQueue		*CONNECTORNAME*/ SYNCHRONOUSRESPONSEQUEUE	Component restart	Delivery Transport is JMS
WireFormat	CwXML, CwBO	CwXML	Agent restart	CwXML if Repository Directory is not <REMOTE>: CwBO if Repository Directory is <REMOTE>
WsifSynchronousRequestTimeout	0 - any number (millisecs)	0	Component restart	WAS only
XMLNameSpaceFormat	short, long	short	Agent restart	WebSphere MQ message brokers and WAS only

Standard configuration properties

This section lists and defines each of the standard connector configuration properties.

AdminInQueue

The queue that is used by the integration broker to send administrative messages to the connector.

The default value is `CONNECTORNAME/ADMININQUEUE`.

AdminOutQueue

The queue that is used by the connector to send administrative messages to the integration broker.

The default value is `CONNECTORNAME/ADMINOUTQUEUE`.

AgentConnections

Applicable only if `RepositoryDirectory` is <REMOTE>.

The AgentConnections property controls the number of ORB (Object Request Broker) connections opened by `orb.init[]`.

The default value of this property is set to 1. You can change it as required.

AgentTraceLevel

Level of trace messages for the application-specific component. The default is `0`. The connector delivers all trace messages applicable at the tracing level set or lower.

ApplicationName

Name that uniquely identifies the connector's application. This name is used by the system administrator to monitor the WebSphere business integration system environment. This property must have a value before you can run the connector.

BrokerType

Identifies the integration broker type that you are using. The options are ICS, WebSphere message brokers (WMQI, WMQIB or WBIMB) or WAS.

CharacterEncoding

Specifies the character code set used to map from a character (such as a letter of the alphabet, a numeric representation, or a punctuation mark) to a numeric value.

Note: Java-based connectors do not use this property. A C++ connector currently uses the value `ascii7` for this property.

By default, a subset of supported character encodings only is displayed in the drop-down list. To add other supported values to the drop-down list, you must manually modify the `\Data\Std\stdConnProps.xml` file in the product directory. For more information, see the sections on Connector Configurator in this guide.

ConcurrentEventTriggeredFlows

Applicable only if RepositoryDirectory is <REMOTE>.

Determines how many business objects can be concurrently processed by the connector for event delivery. Set the value of this attribute to the number of business objects you want concurrently mapped and delivered. For example, set the value of this property to 5 to cause five business objects to be concurrently processed. The default value is 1.

Setting this property to a value greater than 1 allows a connector for a source application to map multiple event business objects at the same time and deliver them to multiple collaboration instances simultaneously. This speeds delivery of business objects to the integration broker, particularly if the business objects use complex maps. Increasing the arrival rate of business objects to collaborations can improve overall performance in the system.

To implement concurrent processing for an entire flow (from a source application to a destination application), you must:

- Configure the collaboration to use multiple threads by setting its Maximum number of concurrent events property high enough to use multiple threads.
- Ensure that the destination application's application-specific component can process requests concurrently. That is, it must be multi-threaded, or be able to use connector agent parallelism and be configured for multiple processes. Set the Parallel Process Degree configuration property to a value greater than 1.

The ConcurrentEventTriggeredFlows property has no effect on connector polling, which is single-threaded and performed serially.

ContainerManagedEvents

This property allows a JMS-enabled connector with a JMS event store to provide guaranteed event delivery, in which an event is removed from the source queue and placed on the destination queue as a single JMS transaction.

There is no default value.

When ContainerManagedEvents is set to JMS, you must configure the following properties to enable guaranteed event delivery:

- PollQuantity = 1 to 500
- SourceQueue = /SOURCEQUEUE

You must also configure a data handler with the MimeType, DHClass (data handler class), and DataHandlerConfigMOName (the meta-object name, which is optional) properties. To set those values, use the **Data Handler** tab in Connector Configurator.

Thes properties are adapter-specific, but **example** values are:

- MimeType = text\xml
- DHClass = com.crossworlds.DataHandlers.text.xml
- DataHandlerConfigMOName = MO_DataHandler_Default

The fields for these values in the Data Handler tab will be displayed only if you have set ContainerManagedEvents to JMS.

Note: When ContainerManagedEvents is set to JMS, the connector does *not* call its pollForEvents() method, thereby disabling that method's functionality.

This property only appears if the DeliveryTransport property is set to the value JMS.

ControllerStoreAndForwardMode

Applicable only if RepositoryDirectory is <REMOTE>.

Sets the behavior of the connector controller after it detects that the destination application-specific component is unavailable.

If this property is set to true and the destination application-specific component is unavailable when an event reaches ICS, the connector controller blocks the request to the application-specific component. When the application-specific component becomes operational, the controller forwards the request to it.

However, if the destination application's application-specific component becomes unavailable **after** the connector controller forwards a service call request to it, the connector controller fails the request.

If this property is set to false, the connector controller begins failing all service call requests as soon as it detects that the destination application-specific component is unavailable.

The default is true.

ControllerTraceLevel

Applicable only if RepositoryDirectory is <REMOTE>.

Level of trace messages for the connector controller. The default is 0.

DeliveryQueue

Applicable only if DeliveryTransport is JMS.

The queue that is used by the connector to send business objects to the integration broker.

The default value is CONNECTORNAME/DELIVERYQUEUE.

DeliveryTransport

Specifies the transport mechanism for the delivery of events. Possible values are MQ for WebSphere MQ, IDL for CORBA IIOP, or JMS for Java Messaging Service.

- If the RepositoryDirectory is remote, the value of the DeliveryTransport property can be MQ, IDL, or JMS, and the default is IDL.
- If the RepositoryDirectory is a local directory, the value may only be JMS.

The connector sends service call requests and administrative messages over CORBA IIOP if the value configured for the DeliveryTransport property is MQ or IDL.

WebSphere MQ and IDL

Use WebSphere MQ rather than IDL for event delivery transport, unless you must have only one product. WebSphere MQ offers the following advantages over IDL:

- Asynchronous communication:
  WebSphere MQ allows the application-specific component to poll and persistently store events even when the server is not available.
- Server side performance:
  WebSphere MQ provides faster performance on the server side. In optimized mode, WebSphere MQ stores only the pointer to an event in the repository database, while the actual event remains in the WebSphere MQ queue. This saves having to write potentially large events to the repository database.
- Agent side performance:
  WebSphere MQ provides faster performance on the application-specific component side. Using WebSphere MQ, the connector's polling thread picks up an event, places it in the connector's queue, then picks up the next event. This is faster than IDL, which requires the connector's polling thread to pick up an event, go over the network into the server process, store the event persistently in the repository database, then pick up the next event.

JMS

Enables communication between the connector and client connector framework using Java Messaging Service (JMS).

If you select JMS as the delivery transport, additional JMS properties such as `jms.MessageBrokerName`, `jms.FactoryClassName`, `jms.Password`, and `jms.UserName`, appear in Connector Configurator. The first two of these properties are required for this transport.

Important: There may be a memory limitation if you use the JMS transport mechanism for a connector in the following environment:

- AIX 5.0
- WebSphere MQ 5.3.0.1
- When ICS is the integration broker

In this environment, you may experience difficulty starting both the connector controller (on the server side) and the connector (on the client side) due to memory use within the WebSphere MQ client. If your installation uses less than 768M of process heap size, IBM recommends that you set:

- The `LDR_CNTRL` environment variable in the `CWSharedEnv.sh` script.

  This script resides in the `\bin` directory below the product directory. With a text editor, add the following line as the first line in the `CWSharedEnv.sh` script:

  ```
  export LDR_CNTRL=MAXDATA=0x30000000
  ```

  This line restricts heap memory usage to a maximum of 768 MB (3 segments * 256 MB). If the process memory grows more than this limit, page swapping can occur, which can adversely affect the performance of your system.
- The `IPCCBaseAddress` property to a value of `11` or `12`. For more information on this property, see the *System Installation Guide for UNIX*.

DuplicateEventElimination

When you set this property to `true`, a JMS-enabled connector can ensure that duplicate events are not delivered to the delivery queue. To use this feature, the connector must have a unique event identifier set as the business object's **ObjectEventId** attribute in the application-specific code. This is done during connector development.

This property can also be set to `false`.

Note: When DuplicateEventElimination is set to true, you must also configure the MonitorQueue property to enable guaranteed event delivery.

FaultQueue

If the connector experiences an error while processing a message then the connector moves the message to the queue specified in this property, along with a status indicator and a description of the problem.

The default value is CONNECTORNAME/FAULTQUEUE.

JvmMaxHeapSize

The maximum heap size for the agent (in megabytes). This property is applicable only if the RepositoryDirectory value is <REMOTE>.

The default value is 128m.

JvmMaxNativeStackSize

The maximum native stack size for the agent (in kilobytes). This property is applicable only if the RepositoryDirectory value is <REMOTE>.

The default value is 128k.

JvmMinHeapSize

The minimum heap size for the agent (in megabytes). This property is applicable only if the RepositoryDirectory value is <REMOTE>.

The default value is 1m.

jms.FactoryClassName

Specifies the class name to instantiate for a JMS provider. You *must* set this connector property when you choose JMS as your delivery transport mechanism (DeliveryTransport).

The default is CxCommon.Messaging.jms.IBMMQSeriesFactory.

jms.MessageBrokerName

Specifies the broker name to use for the JMS provider. You *must* set this connector property when you choose JMS as your delivery transport mechanism (DeliveryTransport).

The default is crossworlds.queue.manager. Use the default when connecting to a local message broker.

When you connect to a remote message broker, this property takes the following (mandatory) values:
QueueMgrName:<Channel>:<HostName>:<PortNumber>,
where the variables are:
QueueMgrName: The name of the queue manager.
Channel: The channel used by the client.
HostName: The name of the machine where the queue manager is to reside.
PortNumber: The port number to be used by the queue manager for listening.

For example:
`jms.MessageBrokerName = WBIMB.Queue.Manager:CHANNEL1:RemoteMachine:1456`

jms.NumConcurrentRequests

Specifies the maximum number of concurrent service call requests that can be sent to a connector at the same time. Once that maximum is reached, new service calls block and wait for another request to complete before proceeding.

The default value is 10.

jms.Password

Specifies the password for the JMS provider. A value for this property is optional.

There is no default.

jms.UserName

Specifies the user name for the JMS provider. A value for this property is optional.

There is no default.

ListenerConcurrency

This property supports multi-threading in MQ Listener when ICS is the integration broker. It enables batch writing of multiple events to the database, thus improving system performance. The default value is 1.

This property applies only to connectors using MQ transport. The `DeliveryTransport` property must be set to `MQ`.

Locale

Specifies the language code, country or territory, and, optionally, the associated character code set. The value of this property determines such cultural conventions as collation and sort order of data, date and time formats, and the symbols used in monetary specifications.

A locale name has the following format:
ll_TT.codeset

where:

ll	a two-character language code (usually in lower case)
TT	a two-letter country or territory code (usually in upper case)
codeset	the name of the associated character code set; this portion of the name is often optional.

By default, only a subset of supported locales appears in the drop-down list. To add other supported values to the drop-down list, you must manually modify the `\Data\Std\stdConnProps.xml` file in the product directory. For more information, refer to the sections on Connector Configurator in this guide.

The default value is en_US. If the connector has not been globalized, the only valid value for this property is en_US. To determine whether a specific connector has been globalized, see the connector version list on these websites:

http://www.ibm.com/software/websphere/wbiadapters/infocenter, or
http://www.ibm.com/websphere/integration/wicserver/infocenter

LogAtInterchangeEnd

Applicable only if RespositoryDirectory is <REMOTE>.

Specifies whether to log errors to the integration broker's log destination. Logging to the broker's log destination also turns on e-mail notification, which generates e-mail messages for the MESSAGE_RECIPIENT specified in the InterchangeSystem.cfg file when errors or fatal errors occur.

For example, when a connector loses its connection to its application, if LogAtInterChangeEnd is set to true, an e-mail message is sent to the specified message recipient. The default is false.

MaxEventCapacity

The maximum number of events in the controller buffer. This property is used by flow control and is applicable only if the value of the RepositoryDirectory property is <REMOTE>.

The value can be a positive integer between 1 and 2147483647. The default value is 2147483647.

MessageFileName

The name of the connector message file. The standard location for the message file is \connectors\messages in the product directory. Specify the message filename in an absolute path if the message file is not located in the standard location.

If a connector message file does not exist, the connector uses InterchangeSystem.txt as the message file. This file is located in the product directory.

Note: To determine whether a specific connector has its own message file, see the individual adapter user guide.

MonitorQueue

The logical queue that the connector uses to monitor duplicate events. It is used only if the DeliveryTransport property value is JMS and DuplicateEventElimination is set to TRUE.

The default value is CONNECTORNAME/MONITORQUEUE

OADAutoRestartAgent

Valid only when the RepositoryDirectory is <REMOTE>.

Specifies whether the connector uses the automatic and remote restart feature. This feature uses the MQ-triggered Object Activation Daemon (OAD) to restart the connector after an abnormal shutdown, or to start a remote connector from System Monitor.

This property must be set to `true` to enable the automatic and remote restart feature. For information on how to configure the MQ-triggered OAD feature. see the *Installation Guide for Windows* or *for UNIX*.

The default value is `false`.

OADMaxNumRetry

Valid only when the `RepositoryDirectory` is <REMOTE>.

Specifies the maximum number of times that the MQ-triggered OAD automatically attempts to restart the connector after an abnormal shutdown. The `OADAutoRestartAgent` property must be set to `true` for this property to take effect.

The default value is `1000`.

OADRetryTimeInterval

Valid only when the `RepositoryDirectory` is <REMOTE>.

Specifies the number of minutes in the retry-time interval for the MQ-triggered OAD. If the connector agent does not restart within this retry-time interval, the connector controller asks the OAD to restart the connector agent again. The OAD repeats this retry process as many times as specified by the `OADMaxNumRetry` property. The `OADAutoRestartAgent` property must be set to `true` for this property to take effect.

The default is `10`.

PollEndTime

Time to stop polling the event queue. The format is `HH:MM`, where *HH* represents 0-23 hours, and *MM* represents 0-59 seconds.

You must provide a valid value for this property. The default value is `HH:MM`, but must be changed.

PollFrequency

This is the interval between the end of the last poll and the start of the next poll. `PollFrequency` specifies the amount of time (in milliseconds) between the end of one polling action, and the start of the next polling action. This is not the interval between polling actions. Rather, the logic is as follows:

- Poll to obtain the number of objects specified by the value of `PollQuantity`.
- Process these objects. For some adapters, this may be partly done on separate threads, which execute asynchronously to the next polling action.
- Delay for the interval specified by `PollFrequency`.
- Repeat the cycle.

Set `PollFrequency` to one of the following values:

- The number of milliseconds between polling actions (an integer).
- The word `key`, which causes the connector to poll only when you type the letter `p` in the connector's Command Prompt window. Enter the word in lowercase.
- The word `no`, which causes the connector not to poll. Enter the word in lowercase.

The default is 10000.

Important: Some connectors have restrictions on the use of this property. Where they exist, these restrictions are documented in the chapter on installing and configuring the adapter.

PollQuantity

Designates the number of items from the application that the connector should poll for. If the adapter has a connector-specific property for setting the poll quantity, the value set in the connector-specific property will override the standard property value.

FIX

An email message is also considerd an event. The connector behaves as follows when it is polled for email.

Polled once - connector goes to pick 1. the body of the message as it is also considered an attachment also. Since no DH was specified for this mime type, it it will ignore the body. 2. conector process first PO attachment. DH is avaiable for this mime type so it sends the business object to the Visual Test Connector. If the 3. accept in VTC again no BO should come thru Polled second time 1. conector process second PO attachment. DH is avaiable for this mime type so it sends teh BO to VTC2. accept in VTC again now the third PO attachment should come through. This is the correct behaviour.

PollStartTime

The time to start polling the event queue. The format is *HH:MM*, where *HH* represents 0-23 hours, and *MM* represents 0-59 seconds.

You must provide a valid value for this property. The default value is HH:MM, but must be changed.

RequestQueue

The queue that is used by the integration broker to send business objects to the connector.

The default value is CONNECTOR/REQUESTQUEUE.

RepositoryDirectory

The location of the repository from which the connector reads the XML schema documents that store the meta-data for business object definitions.

When the integration broker is ICS, this value must be set to <REMOTE> because the connector obtains this information from the InterChange Server repository.

When the integration broker is a WebSphere message broker or WAS, this value must be set to <*local directory*>.

ResponseQueue

Applicable only if DeliveryTransport is JMS and required only if RepositoryDirectory is <REMOTE>.

Designates the JMS response queue, which delivers a response message from the connector framework to the integration broker. When the integration broker is ICS, the server sends the request and waits for a response message in the JMS response queue.

RestartRetryCount

Specifies the number of times the connector attempts to restart itself. When used for a parallel connector, specifies the number of times the master connector application-specific component attempts to restart the slave connector application-specific component.

The default is 3.

RestartRetryInterval

Specifies the interval in minutes at which the connector attempts to restart itself. When used for a parallel connector, specifies the interval at which the master connector application-specific component attempts to restart the slave connector application-specific component. Possible values ranges from 1 to 2147483647.

The default is 1.

RHF2MessageDomain

WebSphere message brokers and WAS only.

This property allows you to configure the value of the field domain name in the JMS header. When data is sent to WMQI over JMS transport, the adapter framework writes JMS header information, with a domain name and a fixed value of `mrm`. A connfigurable domain name enables users to track how the WMQI broker processes the message data.

A sample header would look like this:

```
<mcd><Msd>mrm</Msd><Set>3</Set><Type>
Retek_POPhyDesc</Type><Fmt>CwXML</Fmt></mcd>
```

The default value is `mrm`, but it may also be set to `xml`. This property only appears when DeliveryTransport is set to `JMS`and WireFormat is set to `CwXML`.

SourceQueue

Applicable only if `DeliveryTransport` is JMS and ContainerManagedEvents is specified.

Designates the JMS source queue for the connector framework in support of guaranteed event delivery for JMS-enabled connectors that use a JMS event store. For further information, see “ContainerManagedEvents” on page 60.

The default value is `CONNECTOR/SOURCEQUEUE`.

SynchronousRequestQueue

Applicable only if `DeliveryTransport` is JMS.

Delivers request messages that require a synchronous response from the connector framework to the broker. This queue is necessary only if the connector uses synchronous execution. With synchronous execution, the connector framework

sends a message to the SynchronousRequestQueue and waits for a response back from the broker on the SynchronousResponseQueue. The response message sent to the connector bears a correlation ID that matches the ID of the original message.

The default is CONNECTORNAME/SYNCHRONOUSREQUESTQUEUE

SynchronousResponseQueue

Applicable only if DeliveryTransport is JMS.

Delivers response messages sent in reply to a synchronous request from the broker to the connector framework. This queue is necessary only if the connector uses synchronous execution.

The default is CONNECTORNAME/SYNCHRONOUSRESPONSEQUEUE

SynchronousRequestTimeout

Applicable only if DeliveryTransport is JMS.

Specifies the time in minutes that the connector waits for a response to a synchronous request. If the response is not received within the specified time, then the connector moves the original synchronous request message into the fault queue along with an error message.

The default value is 0.

WireFormat

Message format on the transport.

- If the RepositoryDirectory is a local directory, the setting is CwXML.
- If the value of RepositoryDirectory is <REMOTE>, the setting isCwBO.

WsifSynchronousRequestTimeout

WAS integration broker only.

Specifies the time in minutes that the connector waits for a response to a synchronous request. If the response is not received within the specified, time then the connector moves the original synchronous request message into the fault queue along with an error message.

The default value is 0.

XMLNameSpaceFormat

WebSphere message brokers and WAS integration broker only.

A strong property that allows the user to specify short and long name spaces in the XML format of business object definitions.

The default value is short.

Appendix B. Connector Configurator

This appendix describes how to use Connector Configurator to set configuration property values for your adapter.

You use Connector Configurator to:

- Create a connector-specific property template for configuring your connector
- Create a configuration file
- Set properties in a configuration file

Note:

In this document, backslashes (\) are used as the convention for directory paths. For UNIX installations, substitute slashes (/) for backslashes and follow the conventions for each operating system.

The topics covered in this appendix are:

Overview of Connector Configurator

Connector Configurator allows you to configure the connector component of your adapter for use with these integration brokers:

- WebSphere InterChange Server (ICS)
- WebSphere MQ Integrator, WebSphere MQ Integrator Broker, and WebSphere Business Integration Message Broker, collectively referred to as the WebSphere Message Brokers (WMQI)
- WebSphere Application Server (WAS)

You use Connector Configurator to:

- Create a **connector-specific property template** for configuring your connector.
- Create a **connector configuration file**; you must create one configuration file for each connector you install.
- Set properties in a configuration file.
  You may need to modify the default values that are set for properties in the connector templates. You must also designate supported business object definitions and, with ICS, maps for use with collaborations as well as specify messaging, logging and tracing, and data handler parameters, as required.

The mode in which you run Connector Configurator, and the configuration file type you use, may differ according to which integration broker you are running. For example, if WMQI is your broker, you run Connector Configurator directly, and not from within System Manager (see "Running Configurator in stand-alone mode" on page 72).

Connector configuration properties include both standard configuration properties (the properties that all connectors have) and connector-specific properties (properties that are needed by the connector for a specific application or technology).

Because **standard properties** are used by all connectors, you do not need to define those properties from scratch; Connector Configurator incorporates them into your configuration file as soon as you create the file. However, you do need to set the value of each standard property in Connector Configurator.

The range of standard properties may not be the same for all brokers and all configurations. Some properties are available only if other properties are given a specific value. The Standard Properties window in Connector Configurator will show the properties available for your particular configuration.

For **connector-specific properties**, however, you need first to define the properties and then set their values. You do this by creating a connector-specific property template for your particular adapter. There may already be a template set up in your system, in which case, you simply use that. If not, follow the steps in "Creating a new template" on page 73 to set up a new one.

Note: Connector Configurator runs only in a Windows environment. If you are running the connector in a UNIX environment, use Connector Configurator in Windows to modify the configuration file and then copy the file to your UNIX environment.

Starting Connector Configurator

You can start and run Connector Configurator in either of two modes:

- Independently, in stand-alone mode
- From System Manager

Running Configurator in stand-alone mode

You can run Connector Configurator independently and work with connector configuration files, irrespective of your broker.

To do so:

- From **Start>Programs**, click **IBM WebSphere InterChange Server>IBM WebSphere Business Integration Tools>Connector Configurator**.
- Select **File>New>Connector Configuration**.
- When you click the pull-down menu next to **System Connectivity Integration Broker**, you can select ICS, WebSphere Message Brokers or WAS, depending on your broker.

You may choose to run Connector Configurator independently to generate the file, and then connect to System Manager to save it in a System Manager project (see "Completing a configuration file" on page 77.)

Running Configurator from System Manager

You can run Connector Configurator from System Manager.

To run Connector Configurator:

1. Open the System Manager.

2. In the System Manager window, expand the **Integration Component Libraries** icon and highlight **Connectors**.
3. From the System Manager menu bar, click **Tools>Connector Configurator**. The Connector Configurator window opens and displays a **New Connector** dialog box.
4. When you click the pull-down menu next to **System Connectivity Integration Broker**, you can select ICS, WebSphere Message Brokers or WAS, depending on your broker.

To edit an existing configuration file:

- In the System Manager window, select any of the configuration files listed in the Connector folder and right-click on it. Connector Configurator opens and displays the configuration file with the integration broker type and file name at the top.
- From Connector Configurator, select **File>Open**. Select the name of the connector configuration file from a project or from the directory in which it is stored.
- Click the Standard Properties tab to see which properties are included in this configuration file.

Creating a connector-specific property template

To create a configuration file for your connector, you need a connector-specific property template as well as the system-supplied standard properties.

You can create a brand-new template for the connector-specific properties of your connector, or you can use an existing connector definition as the template.

- To create a new template, see "Creating a new template" on page 73.
- To use an existing file, simply modify an existing template and save it under the new name. You can find existing templates in your `\WebSphereAdapters\bin\Data\App` directory.

Creating a new template

This section describes how you create properties in the template, define general characteristics and values for those properties, and specify any dependencies between the properties. Then you save the template and use it as the base for creating a new connector configuration file.

To create a template in Connector Configurator:

1. Click **File>New>Connector-Specific Property Template**.
2. The **Connector-Specific Property Template** dialog box appears.
   - Enter a name for the new template in the **Name** field below **Input a New Template Name.** You will see this name again when you open the dialog box for creating a new configuration file from a template.
   - To see the connector-specific property definitions in any template, select that template's name in the **Template Name** display. A list of the property definitions contained in that template appears in the **Template Preview** display.
3. You can use an existing template whose property definitions are similar to those required by your connector as a starting point for your template. If you do not see any template that displays the connector-specific properties used by your connector, you will need to create one.

- If you are planning to modify an existing template, select the name of the template from the list in the **Template Name** table below **Select the Existing Template to Modify: Find Template.**
- This table displays the names of all currently available templates. You can also search for a template.

Specifying general characteristics

When you click **Next** to select a template, the **Properties - Connector-Specific Property Template** dialog box appears. The dialog box has tabs for General characteristics of the defined properties and for Value restrictions. The General display has the following fields:

- **General:**
  Property Type
  Updated Method
  Description
- **Flags**
  Standard flags
- **Custom Flag**
  Flag

After you have made selections for the general characteristics of the property, click the **Value** tab.

Specifying values

The **Value** tab enables you to set the maximum length, the maximum multiple values, a default value, or a value range for the property. It also allows editable values. To do so:

1. Click the **Value** tab. The display panel for Value replaces the display panel for General.
2. Select the name of the property in the **Edit properties** display.
3. In the fields for **Max Length** and **Max Multiple Values**, enter your values.

To create a new property value:

1. Select the property in the **Edit properties** list and right-click on it.
2. From the dialog box, select **Add**.
3. Enter the name of the new property value and click OK. The value appears in the **Value** panel on the right.

The **Value** panel displays a table with three columns:

The **Value** column shows the value that you entered in the **Property Value** dialog box, and any previous values that you created.

The **Default Value** column allows you to designate any of the values as the default.

The **Value Range** shows the range that you entered in the **Property Value** dialog box.

After a value has been created and appears in the grid, it can be edited from within the table display.

To make a change in an existing value in the table, select an entire row by clicking on the row number. Then right-click in the **Value** field and click **Edit Value**.

Setting dependencies

When you have made your changes to the **General** and **Value** tabs, click **Next**. The **Dependencies - Connector-Specific Property Template** dialog box appears.

A dependent property is a property that is included in the template and used in the configuration file *only if* the value of another property meets a specific condition. For example, `PollQuantity` appears in the template only if JMS is the transport mechanism and `DuplicateEventElimination` is set to `True`.
To designate a property as dependent and to set the condition upon which it depends, do this:

1. In the **Available Properties** display, select the property that will be made dependent.
2. In the **Select Property** field, use the drop-down menu to select the property that will hold the conditional value.
3. In the **Condition Operator** field, select one of the following:

   == (equal to)

   != (not equal to)

   > (greater than)

   < (less than)

   >= (greater than or equal to)

   <=(less than or equal to)
4. In the **Conditional Value** field, enter the value that is required in order for the dependent property to be included in the template.
5. With the dependent property highlighted in the **Available Properties** display, click an arrow to move it to the **Dependent Property** display.
6. Click **Finish**. Connector Configurator stores the information you have entered as an XML document, under `\data\app` in the`\bin` directory where you have installed Connector Configurator.

Creating a new configuration file

When you create a new configuration file, you must name it and select an integration broker.

- In the System Manager window, right-click on the **Connectors** folder and select **Create New Connector**. Connector Configurator opens and displays the **New Connector** dialog box.
- In stand-alone mode: from Connector Configurator, select **File>New>Connector Configuration**. In the New Connector window, enter the name of the new connector.

You also need to select an integration broker. The broker you select determines the properties that will appear in the configuration file. To select a broker:

- In the **Integration Broker** field, select ICS, WebSphere Message Brokers or WAS connectivity.
- drop-downte the remaining fields in the **New Connector** window, as described later in this chapter.

Creating a configuration file from a connector-specific template

Once a connector-specific template has been created, you can use it to create a configuration file:

1. Click **File>New>Connector Configuration**.
2. The **New Connector** dialog box appears, with the following fields:
   - **Name**

     Enter the name of the connector. Names are case-sensitive. The name you enter must be unique, and must be consistent with the file name for a connector that is installed on the system.

     Important: Connector Configurator does not check the spelling of the name that you enter. You must ensure that the name is correct.
   - **System Connectivity**

     Click ICS or WebSphere Message Brokers or WAS.
   - **Select Connector-Specific Property Template**

     Type the name of the template that has been designed for your connector. The available templates are shown in the **Template Name** display. When you select a name in the Template Name display, the **Property Template Preview** display shows the connector-specific properties that have been defined in that template.

     Select the template you want to use and click **OK**.
3. A configuration screen appears for the connector that you are configuring. The title bar shows the integration broker and connector name. You can fill in all the field values to drop-downte the definition now, or you can save the file and complete the fields later.
4. To save the file, click **File>Save>To File** or **File>Save>To Project**. To save to a project, System Manager must be running.
   If you save as a file, the **Save File Connector** dialog box appears. Choose `*.cfg` as the file type, verify in the File Name field that the name is spelled correctly and has the correct case, navigate to the directory where you want to locate the file, and click **Save**. The status display in the message panel of Connector Configurator indicates that the configuration file was successfully created.

   Important: The directory path and name that you establish here must match the connector configuration file path and name that you supply in the startup file for the connector.
5. To complete the connector definition, enter values in the fields for each of the tabs of the Connector Configurator window, as described later in this chapter.

Using an existing file

You may have an existing file available in one or more of the following formats:

- A connector definition file.
  This is a text file that lists properties and applicable default values for a specific connector. Some connectors include such a file in a `\repository` directory in their delivery package (the file typically has the extension `.txt`; for example, CN_XML.txt for the XML connector).
- An ICS repository file.
  Definitions used in a previous ICS implementation of the connector may be available to you in a repository file that was used in the configuration of that connector. Such a file typically has the extension `.in` or `.out`.
- A previous configuration file for the connector.
  Such a file typically has the extension *.cfg.

Although any of these file sources may contain most or all of the connector-specific properties for your connector, the connector configuration file will not be complete until you have opened the file and set properties, as described later in this chapter.

To use an existing file to configure a connector, you must open the file in Connector Configurator, revise the configuration, and then resave the file.

Follow these steps to open a *.txt, *.cfg, or *.in file from a directory:

1. In Connector Configurator, click **File>Open>From File**.
2. In the **Open File Connector** dialog box, select one of the following file types to see the available files:
   - Configuration (`*.cfg`)
   - ICS Repository (`*.in`, `*.out`)

     Choose this option if a repository file was used to configure the connector in an ICS environment. A repository file may include multiple connector definitions, all of which will appear when you open the file.
   - All files (*.*)

     Choose this option if a `*.txt` file was delivered in the adapter package for the connector, or if a definition file is available under another extension.
3. In the directory display, navigate to the appropriate connector definition file, select it, and click **Open**.

Follow these steps to open a connector configuration from a System Manager project:

1. Start System Manager. A configuration can be opened from or saved to System Manager only if System Manager has been started.
2. Start Connector Configurator.
3. Click **File>Open>From Project**.

Completing a configuration file

When you open a configuration file or a connector from a project, the Connector Configurator window displays the configuration screen, with the current attributes and values.

The title of the configuration screen displays the integration broker and connector name as specified in the file. Make sure you have the correct broker. If not, change the broker value before you configure the connector. To do so:

1. Under the **Standard Properties** tab, select the value field for the `BrokerType` property. In the drop-down menu, select the value `ICS`, `WMQI`, or `WAS`.
2. The Standard Properties tab will display the properties associated with the selected broker. You can save the file now or complete the remaining configuration fields, as described in “Specifying supported business object definitions” on page 80..
3. When you have finished your configuration, click **File>Save>To Project** or **File>Save>To File**.

   If you are saving to file, select `*.cfg` as the extension, select the correct location for the file and click **Save**.

   If multiple connector configurations are open, click **Save All to File** to save all of the configurations to file, or click **Save All to Project** to save all connector configurations to a System Manager project.

Before it saves the file, Connector Configurator checks that values have been set for all required standard properties. If a required standard property is missing a value, Connector Configurator displays a message that the validation failed. You must supply a value for the property in order to save the configuration file.

Setting the configuration file properties

When you create and name a new connector configuration file, or when you open an existing connector configuration file, Connector Configurator displays a configuration screen with tabs for the categories of required configuration values.

Connector Configurator requires values for properties in these categories for connectors running on all brokers:

- Standard Properties
- Connector-specific Properties
- Supported Business Objects
- Trace/Log File values
- Data Handler (applicable for connectors that use JMS messaging with guaranteed event delivery)

Note: For connectors that use JMS messaging, an additional category may display, for configuration of data handlers that convert the data to business objects.

For connectors running on **ICS**, values for these properties are also required:

- Associated Maps
- Resources
- Messaging (where applicable)

Important: Connector Configurator accepts property values in either English or non-English character sets. However, the names of both standard and connector-specific properties, and the names of supported business objects, must use the English character set only.

Standard properties differ from connector-specific properties as follows:

- Standard properties of a connector are shared by both the application-specific component of a connector and its broker component. All connectors have the same set of standard properties. These properties are described in Appendix A of each adapter guide. You can change some but not all of these values.
- Application-specific properties apply only to the application-specific component of a connector, that is, the component that interacts directly with the application. Each connector has application-specific properties that are unique to its application. Some of these properties provide default values and some do not; you can modify some of the default values. The installation and configuration chapters of each adapter guide describe the application-specific properties and the recommended values.

The fields for **Standard Properties** and **Connector-Specific Properties** are color-coded to show which are configurable:

- A field with a grey background indicates a standard property. You can change the value but cannot change the name or remove the property.

- A field with a white background indicates an application-specific property. These properties vary according to the specific needs of the application or connector. You can change the value and delete these properties.
- Value fields are configurable.
- The **Update Method** field is displayed for each property. It indicates whether a component or agent restart is necessary to activate changed values. You cannot configure this setting.

Setting standard connector properties

To change the value of a standard property:

1. Click in the field whose value you want to set.
2. Either enter a value, or select one from the drop-down menu if it appears.
3. After entering all the values for the standard properties, you can do one of the following:
   - To discard the changes, preserve the original values, and exit Connector Configurator, click **File>Exit** (or close the window), and click **No** when prompted to save changes.
   - To enter values for other categories in Connector Configurator, select the tab for the category. The values you enter for **Standard Properties** (or any other category) are retained when you move to the next category. When you close the window, you are prompted to either save or discard the values that you entered in all the categories as a whole.
   - To save the revised values, click **File>Exit** (or close the window) and click **Yes** when prompted to save changes. Alternatively, click **Save>To File** from either the File menu or the toolbar.

Setting application-specific configuration properties

For application-specific configuration properties, you can add or change property names, configure values, delete a property, and encrypt a property. The default property length is 255 characters.

1. Right-click in the top left portion of the grid. A pop-up menu bar will appear. Click **Add** to add a property. To add a child property, right-click on the parent row number and click **Add child**.
2. Enter a value for the property or child property.
3. To encrypt a property, select the **Encrypt** box.
4. Choose to save or discard changes, as described for "Setting standard connector properties."

The Update Method displayed for each property indicates whether a component or agent restart is necessary to activate changed values.

Important: Changing a preset application-specific connector property name may cause a connector to fail. Certain property names may be needed by the connector to connect to an application or to run properly.

Encryption for connector properties

Application-specific properties can be encrypted by selecting the **Encrypt** check box in the Connector-specific Properties window. To decrypt a value, click to clear the **Encrypt** check box, enter the correct value in the **Verification** dialog box, and click **OK**. If the entered value is correct, the value is decrypted and displays.

The adapter user guide for each connector contains a list and description of each property and its default value.

If a property has multiple values, the **Encrypt** check box will appear for the first value of the property. When you select **Encrypt**, all values of the property will be encrypted. To decrypt multiple values of a property, click to clear the **Encrypt** check box for the first value of the property, and then enter the new value in the **Verification** dialog box. If the input value is a match, all multiple values will decrypt.

Update method

Refer to the descriptions of update methods found in the *Standard configuration properties for connectors* appendix, under "Setting and updating property values" on page 54.

Specifying supported business object definitions

Use the **Supported Business Objects** tab in Connector Configurator to specify the business objects that the connector will use. You must specify both generic business objects and application-specific business objects, and you must specify associations for the maps between the business objects.

Note: Some connectors require that certain business objects be specified as supported in order to perform event notification or additional configuration (using meta-objects) with their applications. For more information, see the *Connector Development Guide for C++* or the *Connector Development Guide for Java*.

If ICS is your broker

To specify that a business object definition is supported by the connector, or to change the support settings for an existing business object definition, click the **Supported Business Objects** tab and use the following fields.

Business object name: To designate that a business object definition is supported by the connector, with System Manager running:

1. Click an empty field in the **Business Object Name** list. A drop-down list displays, showing all the business object definitions that exist in the System Manager project.
2. Click on a business object to add it.
3. Set the **Agent Support** (described below) for the business object.
4. In the File menu of the Connector Configurator window, click **Save to Project**. The revised connector definition, including designated support for the added business object definition, is saved to an ICL (Integration Component Library) project in System Manager.

To delete a business object from the supported list:

1. To select a business object field, click the number to the left of the business object.
2. From the **Edit** menu of the Connector Configurator window, click **Delete Row**. The business object is removed from the list display.
3. From the **File** menu, click **Save to Project**.

Deleting a business object from the supported list changes the connector definition and makes the deleted business object unavailable for use in this implementation

of this connector. It does not affect the connector code, nor does it remove the business object definition itself from System Manager.

Agent support: If a business object has Agent Support, the system will attempt to use that business object for delivering data to an application via the connector agent.

Typically, application-specific business objects for a connector are supported by that connector's agent, but generic business objects are not.

To indicate that the business object is supported by the connector agent, check the **Agent Support** box. The Connector Configurator window does not validate your Agent Support selections.

Maximum transaction level: The maximum transaction level for a connector is the highest transaction level that the connector supports.

For most connectors, `Best Effort` is the only possible choice.

You must restart the server for changes in transaction level to take effect.

If a WebSphere Message Broker is your broker

If you are working in stand-alone mode (not connected to System Manager), you must enter the business object name manually.

If you have System Manager running, you can select the empty box under the **Business Object Name** column in the **Supported Business Objects** tab. A combo box appears with a list of the business object available from the Integration Component Library project to which the connector belongs. Select the business object you want from the list.

The **Message Set ID** is an optional field for WebSphere Business Integration Message Broker 5.0, and need not be unique if supplied. However, for WebSphere MQ Integrator and Integrator Broker 2.1, you must supply a unique **ID**.

If WAS is your broker

When WebSphere Application Server is selected as your broker type, Connector Configurator does not require message set IDs. The **Supported Business Objects** tab shows a **Business Object Name** column only for supported business objects.

If you are working in stand-alone mode (not connected to System Manager), you must enter the business object name manually.

If you have System Manager running, you can select the empty box under the Business Object Name column in the Supported Business Objects tab. A combo box appears with a list of the business objects available from the Integration Component Library project to which the connector belongs. Select the business object you want from this list.

Associated maps (ICS only)

Each connector supports a list of business object definitions and their associated maps that are currently active in WebSphere InterChange Server. This list appears when you select the **Associated Maps** tab.

The list of business objects contains the application-specific business object which the agent supports and the corresponding generic object that the controller sends

to the subscribing collaboration. The association of a map determines which map will be used to transform the application-specific business object to the generic business object or the generic business object to the application-specific business object.

If you are using maps that are uniquely defined for specific source and destination business objects, the maps will already be associated with their appropriate business objects when you open the display, and you will not need (or be able) to change them.

If more than one map is available for use by a supported business object, you will need to explicitly bind the business object with the map that it should use.

The **Associated Maps** tab displays the following fields:

- **Business Object Name**

  These are the business objects supported by this connector, as designated in the **Supported Business Objects** tab. If you designate additional business objects under the Supported Business Objects tab, they will be reflected in this list after you save the changes by choosing **Save to Project** from the **File** menu of the Connector Configurator window.
- **Associated Maps**

  The display shows all the maps that have been installed to the system for use with the supported business objects of the connector. The source business object for each map is shown to the left of the map name, in the **Business Object Name** display.
- **Explicit**

  In some cases, you may need to explicitly bind an associated map.

  Explicit binding is required only when more than one map exists for a particular supported business object. When ICS boots, it tries to automatically bind a map to each supported business object for each connector. If more than one map takes as its input the same business object, the server attempts to locate and bind one map that is the superset of the others.

  If there is no map that is the superset of the others, the server will not be able to bind the business object to a single map, and you will need to set the binding explicitly.

  To explicitly bind a map:

  1. In the **Explicit** column, place a check in the check box for the map you want to bind.
  2. Select the map that you intend to associate with the business object.
  3. In the **File** menu of the Connector Configurator window, click **Save to Project**.
  4. Deploy the project to ICS.
  5. Reboot the server for the changes to take effect.

Resources (ICS)

The **Resource** tab allows you to set a value that determines whether and to what extent the connector agent will handle multiple processes concurrently, using connector agent parallelism.

Not all connectors support this feature. If you are running a connector agent that was designed in Java to be multi-threaded, you are advised not to use this feature, since it is usually more efficient to use multiple threads than multiple processes.

Messaging (ICS)

The messaging properties are available only if you have set MQ as the value of the `DeliveryTransport` standard property and ICS as the broker type. These properties affect how your connector will use queues.

Setting trace/log file values

When you open a connector configuration file or a connector definition file, Connector Configurator uses the logging and tracing values of that file as default values. You can change those values in Connector Configurator.

To change the logging and tracing values:

1. Click the **Trace/Log Files** tab.
2. For either logging or tracing, you can choose to write messages to one or both of the following:
   - To console (STDOUT):
     Writes logging or tracing messages to the STDOUT display.

     Note: You can only use the STDOUT option from the **Trace/Log Files** tab for connectors running on the Windows platform.
   - To File:
     Writes logging or tracing messages to a file that you specify. To specify the file, click the directory button (ellipsis), navigate to the preferred location, provide a file name, and click **Save**. Logging or tracing message are written to the file and location that you specify.

     Note: Both logging and tracing files are simple text files. You can use the file extension that you prefer when you set their file names. For tracing files, however, it is advisable to use the extension `.trace` rather than `.trc`, to avoid confusion with other files that might reside on the system. For logging files, `.log` and `.txt` are typical file extensions.

Data handlers

The data handlers section is available for configuration only if you have designated a value of JMS for DeliveryTransport and a value of JMS for ContainerManagedEvents. Not all adapters make use of data handlers.

See the descriptions under `ContainerManagedEvents` in Appendix A, Standard Properties, for values to use for these properties. For additional details, see the *Connector Development Guide for C++* or the *Connector Development Guide for Java.*

Saving your configuration file

When you have finished configuring your connector, save the connector configuration file. Connector Configurator saves the file in the broker mode that you selected during configuration. The title bar of Connector Configurator always displays the broker mode (ICS, WMQI or WAS) that it is currently using.

The file is saved as an XML document. You can save the XML document in three ways:

- From System Manager, as a file with a `*.con` extension in an Integration Component Library, or
- In a directory that you specify.

- In stand-alone mode, as a file with a `*.cfg` extension in a directory folder. By default, the file is saved to `\WebSphereAdapters\bin\Data\App`.
- You can also save it to a WebSphere Application Server project if you have set one up.

For details about using projects in System Manager, and for further information about deployment, see the following implementation guides:

- For ICS: *Implementation Guide for WebSphere InterChange Server*
- For WebSphere Message Brokers: *Implementing Adapters with WebSphere Message Brokers*
- For WAS: *Implementing Adapters with WebSphere Application Server*

Changing a configuration file

You can change the integration broker setting for an existing configuration file. This enables you to use the file as a template for creating a new configuration file, which can be used with a different broker.

Note: You will need to change other configuration properties as well as the broker mode property if you switch integration brokers.

To change your broker selection within an existing configuration file (optional):

- Open the existing configuration file in Connector Configurator.
- Select the **Standard Properties** tab.
- In the **BrokerType** field of the Standard Properties tab, select the value that is appropriate for your broker.
  When you change the current value, the available tabs and field selections on the properties screen will immediately change, to show only those tabs and fields that pertain to the new broker you have selected.

Completing the configuration

After you have created a configuration file for a connector and modified it, make sure that the connector can locate the configuration file when the connector starts up.

To do so, open the startup file used for the connector, and verify that the location and file name used for the connector configuration file match exactly the name you have given the file and the directory or path where you have placed it.

Using Connector Configurator in a globalized environment

Connector Configurator is globalized and can handle character conversion between the configuration file and the integration broker. Connector Configurator uses native encoding. When it writes to the configuration file, it uses UTF-8 encoding.

Connector Configurator supports non-English characters in:

- All value fields
- Log file and trace file path (specified in the **Trace/Log files** tab)

The drop list for the `CharacterEncoding` and `Locale` standard configuration properties displays only a subset of supported values. To add other values to the drop list, you must manually modify the `\Data\Std\stdConnProps.xml` file in the product directory.

For example, to add the locale en_GB to the list of values for the Locale property, open the stdConnProps.xml file and add the line in boldface type below:

```
<Property name="Locale"
isRequired="true"
updateMethod="component restart">
                <ValidType>String</ValidType>
            <ValidValues>
                                <Value>ja_JP</Value>
                                <Value>ko_KR</Value>
                                <Value>zh_CN</Value>
                                <Value>zh_TW</Value>
                                <Value>fr_FR</Value>
                                <Value>de_DE</Value>
                                <Value>it_IT</Value>
                                <Value>es_ES</Value>
                                <Value>pt_BR</Value>
                                <Value>en_US</Value>
                                <Value>en_GB</Value>
                    <DefaultValue>en_US</DefaultValue>
            </ValidValues>
    </Property>
```

Notices

IBM may not offer the products, services, or features discussed in this document in all 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.

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 PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES 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 publication. IBM may make improvements and/or changes in the product(s) and/or 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:

IBM Burlingame Laboratory Director
IBM Burlingame Laboratory
577 Airport Blvd., Suite 800

Burlingame, CA 94010
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 Customer Agreement, 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 measurement 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 necessarily 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.

This information may contain 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.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

COPYRIGHT LICENSE

This information may contain sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Programming interface information

Programming interface information, if provided, is intended to help you create application software using this program.

General-use programming interfaces allow you to write application software that obtain the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification and tuning information is provided to help you debug your application software.

Warning: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and service marks

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States or other countries, or both:

IBM
the IBM logo
AIX
CrossWorlds
DB2
DB2 Universal Database
Domino
Lotus
Lotus Notes
MQIntegrator
MQSeries
Tivoli
WebSphere

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

MMX, Pentium, and ProShare are trademarks or registered trademarks of Intel Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product or service names may be trademarks or service marks of others.

WebSphere Business Integration Adapter Framework V2.4.0

Printed in USA