IBM WebSphere Commerce

Programmer's Guide

Version 5.4

GC09-4951-02

IBM WebSphere Commerce

Programmer's Guide

Version 5.4

GC09-4951-02

Note:
Before using this information and the product it supports, be sure to read the information in the Notices section.

Second Edition (June 2002, Revision 1 released in September 2002)

This edition applies to the following products:
- IBM WebSphere Commerce Business Edition for Windows NT and Windows 2000, Version 5.4
- IBM WebSphere Commerce Business Edition for AIX, Version 5.4
- IBM WebSphere Commerce Business Edition for Solaris Operating Environment Software, Version 5.4
- IBMWebSphere Commerce Business Edition for the IBM @server iSeries 400, Version 5.4
- IBM WebSphere Commerce Business Edition for Linux, Version 5.4
- IBM WebSphere Commerce Business Edition for Linux for IBM @server zSeries and S/390, Version 5.4
- IBM WebSphere Commerce Studio, Business Developer Edition for Windows NT and Windows 2000, Version 5.4
- IBM WebSphere Commerce Professional Edition for Windows NT and Windows 2000, Version 5.4
- IBM WebSphere Commerce Professional Edition for AIX, Version 5.4
- IBM WebSphere Commerce Professional Edition for Solaris Operating Environment Software, Version 5.4
- IBMWebSphere Commerce Professional Edition for the IBM @server iSeries 400, Version 5.4
- IBM WebSphere Commerce Professional Edition for Linux, Version 5.4
- IBM WebSphere Commerce Professional Edition for Linux for IBM @server zSeries and S/390, Version 5.4
- IBM WebSphere Commerce Studio, Professional Developer Edition for Windows NT and Windows 2000, Version 5.4

and to all subsequent releases and modifications of the above listed products, until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product. (The program number for WebSphere Commerce and WebSphere Commerce Studio is 5724-A18.)

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

IBM welcomes your comments. You can send comments about this publication by one of the following methods:
1. Electronically to following e-mail address:

   `torrcf@ca.ibm.com`
2. By mail to the following address:

   IBM Canada Ltd. Laboratory
   B3/KB7/8200/MKM
   8200 Warden Avenue
   Markham, Ontario, L6G 1C7
   Canada

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.

Before you begin

The *IBM® WebSphere® Commerce Programmer's Guide* provides information about the WebSphere Commerce architecture and programming model. In particular, it provides details on the following topics:

- Component interactions
- Design patterns
- Persistent object model
- Access control
- Error handling and messages
- Command implementation
- Development tools
- Deployment of customized code

In addition, this book includes the following tutorials:

Creating new business logic
This tutorial demonstrates how to create new commands, data beans and enterprise beans following the WebSphere Commerce programming model. It also demonstrates how to integrate the logic into an existing store and deploy the code to a target WebSphere Commerce Server.

Modifying and extending existing business logic
This tutorial is divided into two sections. The first section demonstrates how to add new logic to an existing controller command. The second section demonstrates how to modify an existing task command and WebSphere Commerce entity bean. It also demonstrates how to integrate the modifications into an existing store and deploy the code to a target WebSphere Commerce Server.

Conventions used in this book

This book uses the following highlighting conventions:

Boldface type indicates commands or graphical user interface (GUI) controls such as names of fields, buttons, or menu choices.

`Monospaced type` indicates examples of text you enter exactly as shown, as well as directory paths.

Italic type is used for emphasis and variables for which you substitute your own values.

This icon marks a Tip — additional information that can help you complete a task.

Windows indicates information that is specific to WebSphere Commerce for Windows NT® and Windows® 2000.

AIX indicates information that is specific to WebSphere Commerce for AIX®.

Solaris indicates information that is specific to WebSphere Commerce for Solaris™ Operating Environment software.

400 indicates information specific to WebSphere Commerce for the IBM @server™ iSeries™ 400® (formerly called AS/400®)

Linux indicates information specific to the following products:

- WebSphere Commerce Business Edition for Linux.
- WebSphere Commerce Professional Edition for Linux.
- WebSphere Commerce Business Edition for Linux for IBM @server zSeries™ and S/390®.

DB2 indicates information specific to DB2® Universal Database

Oracle indicates information specific to Oracle®.

Professional indicates information specific to WebSphere Commerce Professional Edition.

Business indicates information specific to WebSphere Commerce Business Edition.

Knowledge requirements

This book should be read by Store Developers that need to understand how to customize a WebSphere Commerce application. Store Developers that are performing programmatic extensions should have knowledge in the following areas:

- Java™
- Enterprise JavaBeans (EJB) component architecture
- JavaServer Pages technology
- HTML
- Database technology
- VisualAge® for Java, Enterprise Edition, Version 3.5

What's new in this book

In support of the WebSphere Commerce Business Edition for Linux for IBM eServer zSeries and S/390 product, this book was updated in September 2002 to include the following highlighting convention:

▶ Linux indicates information specific to the following products:

- WebSphere Commerce Business Edition for Linux.
- WebSphere Commerce Professional Edition for Linux.
- WebSphere Commerce Business Edition for Linux for IBM @server zSeries and S/390.

Where to find more information

This book may be updated in the future. Check the following WebSphere Commerce Web sites for updates:

▶ Business

```
http://www.ibm.com/software/webservers/commerce/wc_be/lit-tech-general.html
```

▶ Professional

```
http://www.ibm.com/software/webservers/commerce/wc_pe/lit-tech-general.html
```

Updates may include new information, additional or updated tutorials, and sample code related to the tutorials.

Contents

Before you begin iii
Conventions used in this book iii
Knowledge requirements iv
What's new in this book v
Where to find more information v

Part 1. Concepts and architecture . 1

Chapter 1. Overview 3
WebSphere Commerce software components . 3
WebSphere Commerce application architecture 4
WebSphere Commerce run-time architecture . 6
  Servlet engine 8
  Adapter manager 8
  Protocol listeners 8
  Adapters 9
  Web controller 10
  Commands 11
  WebSphere Commerce entity beans . . . 12
  Data beans 13
  Data bean manager 13
  JavaServer Pages templates 13
  Instance_name.xml configuration file . . . 13
Summary for a request 13
Key differences between customization in previous releases 15

Part 2. Programming model 17

Chapter 2. Design patterns 19
Model-View-Controller design pattern . . . 19
Command design pattern 20
  Command framework. 21
  Command factory 23
  Command flow 24
  Command registration framework . . . 26
Display design pattern 36
  JSP templates and data beans 36
  Types of data beans 37
  Invoking controller commands from within a JSP template 40
  Lazy fetch data retrieval 41
Setting JSP attributes - overview 41
  Required property settings 43

Chapter 3. Persistent object model . . . 45
Implementation of WebSphere Commerce entity beans 45
  WebSphere Commerce entity beans - overview 45
  Deployment descriptors for WebSphere Commerce enterprise beans 47
  Extending the WebSphere Commerce object model 48
  Object life cycles 72
  Transactions 73
  Other considerations for entity beans . . 73
Using entity beans 77
Database considerations 78
  Database schema object naming considerations 78
  Database column datatype considerations 80
  Datatype differences between databases . . 82

Chapter 4. Access control. 85
Understanding access control 85
  Overview of resource protection in WebSphere Application Server 85
  Introduction to WebSphere Commerce access control policies 87
  Types of access control 96
  Access control interactions 98
  Protectable interface 101
  Groupable interface 101
  Finding more information about access control 102
Implementing access control 102
  Identifying protectable resources 102
  Implementing access control in enterprise beans 103
  Implementing access control in data beans 104
  Implementing access control in controller commands 106
  Implementing access control policies in views 108

Chapter 5. Error handling and messages 111
Command error handling 111
  Types of exceptions 111

Error message properties files 112
Exception handling flow 112
Exception handling in customized code 114
Creating messages 116
Execution flow tracing 119
JSP template error handling 121

Chapter 6. Command implementation . . 123
New commands - introduction 123
Packaging customized code 126
Command context 127
New controller commands 128
isGeneric method 129
isRetriable method 129
setRequestProperties method 130
validateParameters method 130
getResources method 130
performExecute method. 130
Long-running controller commands . . . 131
Formatting of input properties to view commands 132
Flattening input parameters into a query string for HttpRedirectView 133
Handling a limited length redirect URL 133
Setting attributes in the HttpServletRequest object for HttpForwardView. 134
Database commits and rollbacks for controller commands. 135
Example of transaction scope with a controller command 136
New task commands. 138
Customization of existing commands . . . 139
Customizing existing controller commands 139
Customizing existing task commands . . 143
Data bean customization 145

Chapter 7. Trading agreements and business policies (Business Edition) . . 147
Introduction 147
Business policy objects and commands. . . 148
ToolTech sample contract data 150
CONTRACT table sample data 150
TERMCOND table sample data 151
POLICYTC table sample data 151
POLICY table sample data 152
TRADEPOSCN table sample data . . . 152
SHIPMODE table sample data 152
Extending the existing contract model . . . 152
Creating a new business policy 153
Creating a new business policy type . . 153
Writing the new business policy command 155
Registering the new business policy and business policy command 157
Relating a terms and conditions object to a new business policy 158
Creating new terms and conditions . . . 159
Invoking the new business policy 173
Creating a contract 174
Contract customization scenarios. 174
Rebate scenario 175

Part 3. Development environment 183

Chapter 8. Development tools and deployment 185
Development environment 185
WebSphere Commerce Studio 186
Features and functions of VisualAge for Java 186
WebSphere Commerce code repository . . . 187
Code deployment 187
Information about EJB deployed code . . 188
Deployment of new commands and data beans 189
Deployment of new entity beans 190
Deployment of extensions to existing commands and data beans 192
Deployment of modified WebSphere Commerce public entity beans 192
Deployment of new data beans for use in Commerce Studio 194
Deployment of customized public entity beans for use in Commerce Studio . . . 195
Log files 195
Test payment method 196
Using a remote Payment Manager . . . 197

Part 4. Tutorials 199

Chapter 9. Tutorial: Creating new business logic 201
Tutorial environment. 201
Tutorial code deployment steps 201
Preparing the sample project 202
Writing commands 204
Write a controller command 204
Modify MyNewControllerCmd 209

Creating a new entity bean 237
Creating the new database table 237
Creating the BonusBean entity bean. . . 238
(Optional) Using the Debugger in VisualAge for Java 261
Adding the breakpoint to your code . . 261
Verifying the values of variables 262
Removing the breakpoint 262
Integrating MyNewControllerCmd with the sample store in the WebSphere Test Environment 263
(Optional) Deploying new business logic to a remote WebSphere Commerce Server . . . 264
Create the JAR file for the new command logic 264
Creating the JAR file for the new EJB group 265
Creating the implementation JAR file for the new enterprise bean. 266
Copy the JSP files to the target WebSphere Commerce Server 267
Copy the JAR files to the target WebSphere Commerce Server 267
Running the EJB deploy tool 268
Modify transaction isolation level for the Bonus bean 269
Updating the target database 270
Loading the access control policies for the new resources 272
Exporting the current enterprise application from WebSphere Application Server. 276
Exporting XML configuration information for the enterprise application 276
Assembling the new EJB group into the enterprise application 278
Importing the new enterprise application into WebSphere Application Server . . . 281
Test MyNewControllerCmd 282

Chapter 10. Modifying and extending existing business logic 285
Extending an existing controller command 285
Creating the new package for `OrderProcessCmdBonusImpl` 286
Creating the `OrderProcessCmdBonusImpl` class 286
Adding fields and methods to `OrderProcessCmdBonusImpl` 286
Modifying the command registry to use `OrderProcessCmdBonusImpl` 290
Modifying the `confirmation.jsp` template 291
Testing `OrderProcessCmdBonusImpl` within the WebSphere Test Environment . . . 292
(Optional) Deploying the customized business logic to a remote WebSphere Commerce Server 292
Modifying an existing entity bean and extending an existing task command . . . 296
Adding a new bonusPoint field to the User entity bean 300
Creating and populating the BONUS table 300
Updating the schema and table mapping 303
Generating the deployed code and access bean 305
Testing the modification using the test client 306
Creating the `GetNewProductContractUnitPriceCmd` interface 306
Creating the `GetNewContractUnitPriceCmdImpl` implementation class. 308
Creating the `NewProductDataBean` data bean 311
Adding the new bonus price to the product display template 313
Testing the enterprise bean extension . . 314
(Optional) Deploying the customized business logic to a remote WebSphere Commerce Server 316

Part 5. Appendixes 331

Appendix A. Starting and stopping the WebSphere Test Environment 333
Starting and stopping the persistent name server. 333
Starting and stopping the EJB server . . . 334
Starting and stopping the servlet engine . . 334

Appendix B. Deployment details 335
Mapping to the integrated file system (iSeries) 335
JAR files for customized commands and data beans 335
Creating JAR files for new entity beans . . 337

Creating the EJB 1.1 Export JAR file. . . . 337
Creating the implementation JAR file . . . 338
Creating JAR files for customized WebSphere Commerce entity beans 339
Creating the EJB 1.1 Export JAR file. . . . 340
Creating the client JAR file. 341
Storing assets on the target WebSphere Commerce Server 342
Updating the target database 345
Generating deployed code 346
Modifying transaction isolation level of entity beans. 349
Exporting the current WebSphere Commerce enterprise application 350
Exporting configuration information for enterprise beans 351
Assembling new enterprise beans into an enterprise application 357
Assembling modified enterprise beans into an enterprise application 361
Stopping and removing an enterprise application 366
Importing an enterprise application. . . . 366
Starting an enterprise application 368

Appendix C. Tips for VisualAge for Java 369
Changing properties for the servlet engine in the WebSphere Test Environment 369
Resolving persistent name server problems 370
Deleting compiled JSP files. 370

Notices 371
Trademarks and service marks 374

Index 375

Part 1. Concepts and architecture

Chapter 1. Overview

WebSphere Commerce software components

Before examining how the WebSphere Commerce Server functions, it is useful to look at the larger picture of software components that relate to the customization process. The following diagram shows a simplified view of these software products:

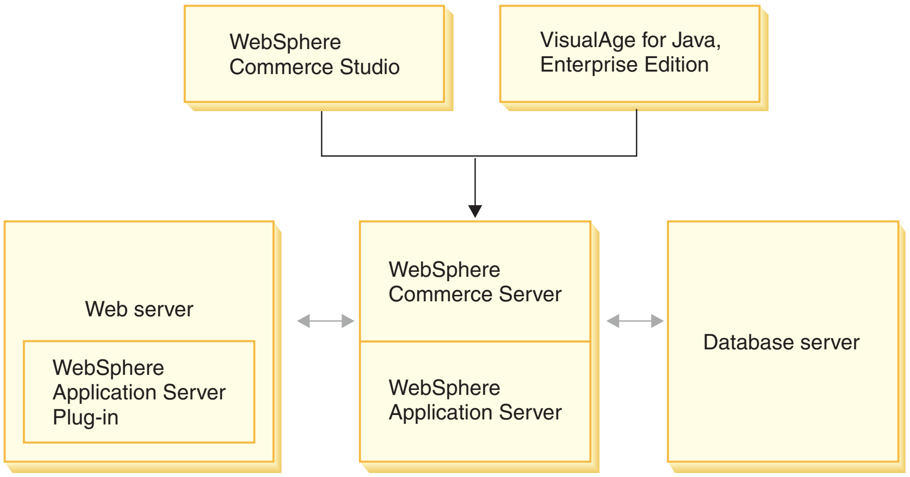

WebSphere Commerce Studio
VisualAge for Java, Enterprise Edition
Web server
WebSphere Application Server Plug-in
WebSphere Commerce Server
WebSphere Application Server
Database server

Figure 1.

The Web server is the first point of contact for incoming HTTP requests for your e-commerce application. In order to interface efficiently with the WebSphere Application Server, it uses the WebSphere Application Server plug-in.

The WebSphere Commerce Server runs within the WebSphere Application Server, allowing it to take advantage of many of the features of the application server. The database server holds most of your application’s data, including product and shopper data. In general, extensions to your application are made by modifying or extending the code for the WebSphere Commerce Server. In addition, you may have a need to store data that falls outside of the realm of the WebSphere Commerce database schema within your database.

Developers use two main tools to create customized business logic: WebSphere Commerce Studio and VisualAge for Java, Enterprise Edition.

WebSphere Commerce Studio is used to create and manage store front assets (for example, JSP templates). VisualAge for Java, Enterprise Edition is used to create new business logic, in Java, that either extends existing functionality or that creates completely new functions. If your application requires extensions to the database schema, database developers should use their preferred database development tools to create the new tables.

WebSphere Commerce application architecture

Now that you have seen how the various software components related to customization fit together, it is important to understand the application architecture. This will help you to understand which parts are foundation layers and which parts you can modify. The following diagram shows the various layers that comprise the application architecture:

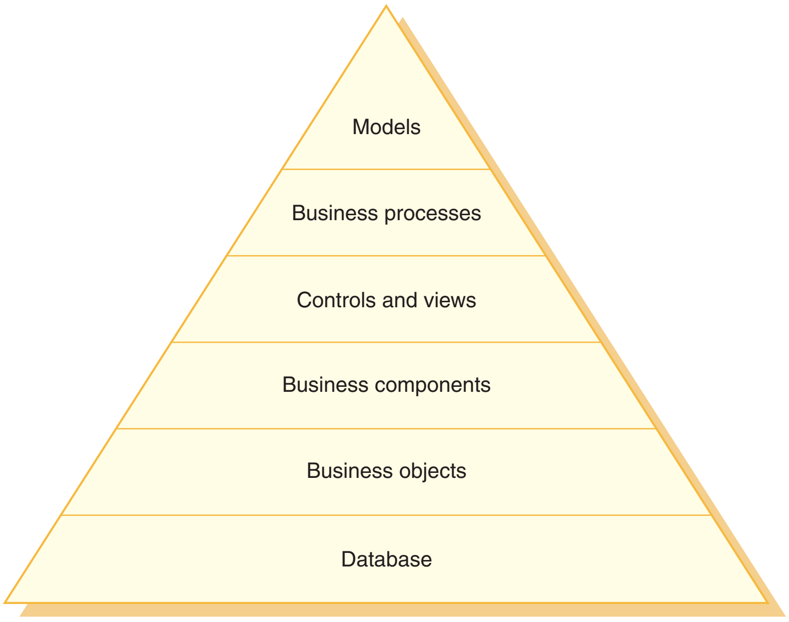


Figure 2.

Each layer of the application architecture is described below:

Database

WebSphere Commerce uses a database schema designed specifically for e-commerce applications and their data requirements. The following are examples of tables in this schema:

- User
- Order

- Product

Business objects

Business objects represent entities within the commerce domain and encapsulate the data-centric logic required to extract or interpret information contained within the database. These entities comply with the Enterprise JavaBeans specification.

These entity beans act as an interface between the commerce application and the database. In addition, the entity beans are easier to comprehend than complex relationships between columns in database tables.

Business components

Business components are units of business logic. They perform coarse-grained procedural business logic. The logic is implemented using the WebSphere Commerce model of controller commands and task commands. An example of this type of component is the OrderProcess controller command. This particular command encapsulates all of the business logic required to process a typical order. The e-commerce application calls the OrderProcess command, which in turn, calls several task commands to perform individual units of work. For example, individual task commands ensure that enough inventory is available to meet the requirements of the order, process the payment, update the status of the order and when the process has completed, decrement the inventory by the appropriate amount.

Controls and views

A Web controller determines the appropriate controller command implementation and view to be used. Implementations can be store specific.
Views display the results of commands and user actions. They are implemented using JSP templates. Examples of views include ProductDisplay (returns a product page showing relevant information for the shopper's selected product) and OrderPrepare (presents the shopper with a form to submit appropriate order information).

Business processes

Sets of business components and views together create workflow and siteflow processes that are known as business processes. Examples of business processes include:

User registration

This business process includes the business components (for example, the UserRegistrationAdd command that creates a registration record for a new user) and views related to all steps involved in the process of registering users.

Catalog navigation
This business process includes the business components (for example, the StoreCatalogDisplay and CategoryDisplay commands that respectively show the catalogs for a store and the categories within a catalog) and views related to all steps involved in the process of navigating through a catalog.

Models
When gathered together, the lower layers of the diagram make up e-commerce business models. One example of an e-commerce business model is the business-to-consumer model that is used by the InFashion sample store. Another example is the business-to-business model that is used by the ToolTech sample store.

WebSphere Commerce run-time architecture

The previous section introduced the application architecture, which depicts the various layers in the WebSphere Commerce application, from a business application point-of-view. This section describes how the run-time architecture is implemented.

The major components of the WebSphere Commerce run-time architecture are:

- Servlet engine
- Protocol listeners
- Adapter manager
- Adapters
- Web controller
- Commands
- Entity beans
- Data beans
- Data bean manager
- Display pages
- XML files

The interactions between WebSphere Commerce components is shown in the following diagram. More detail on each component can be found in subsequent sections.

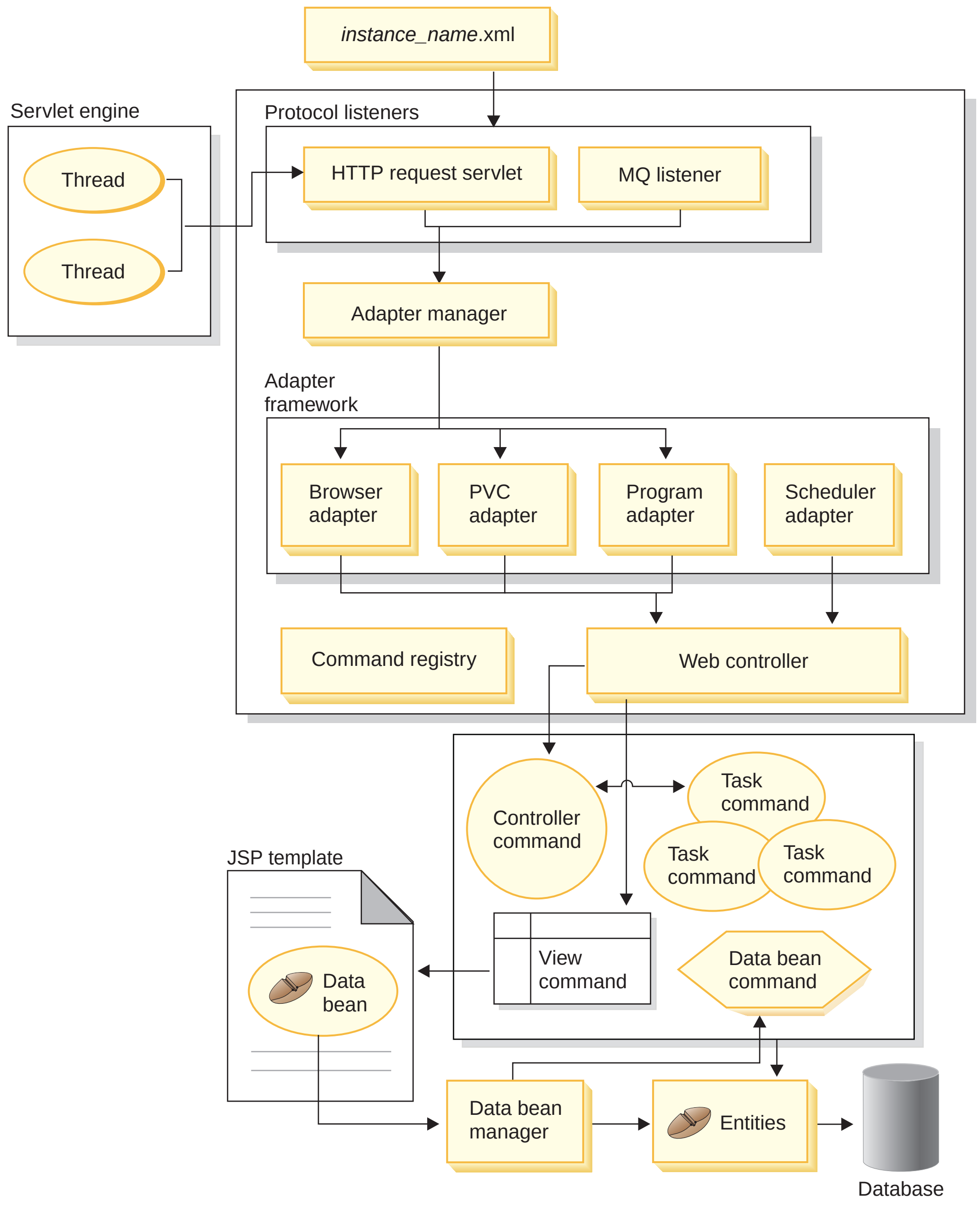

instance_name.xml
Servlet engine
Thread
Thread
Protocol listeners
HTTP request servlet
MQ listener
Adapter manager
Adapter framework
Browser adapter
PVC adapter
Program adapter
Scheduler adapter
Command registry
Web controller
Controller command
Task command
Task command
Task command
JSP template
Data bean
View command
Data bean command
Data bean manager
Entities
Database

Figure 3.

Servlet engine

The servlet engine is the part of the WebSphere Application Server run-time environment that acts as a request dispatcher for inbound URL requests. The servlet engine manages a pool of threads to handle requests. Each inbound request is executed on a separate thread.

Previous versions of WebSphere Commerce used a C++ application server that implemented its own task dispatcher for URL requests by maintaining a pre-allocated set of system processes. This old model, which used system processes, was more resource intensive than the new model that uses Java threads. By exploiting the servlet engine, the new WebSphere Commerce run-time architecture provides a more scalable commerce solution.

Adapter manager

The adapter manager determines which adapter is capable of handling the request and then forwards the request to that adapter.

Protocol listeners

WebSphere Commerce commands can be invoked from various devices. Examples of devices that can invoke commands include:

- Typical Internet browsers
- Mobile phones using Internet browsers
- Business-to-business applications sending XML messages using MQSeries®
- Procurement systems sending requests using XML over HTTP
- The WebSphere Commerce scheduler that executes a background job

Devices can use a variety of communication protocols. A protocol listener is a run-time component that receives inbound requests from transports and then dispatches the requests to the appropriate adapters, based upon the protocol used. The protocol listeners include:

- Request servlet
- MQSeries listener

When the request servlet receives a URL request from the servlet engine, it passes the request to the adapter manager. The adapter manager then queries the adapter types to determine which adapter can process the request. Once the specific adapter is determined, the request is passed to the adapter.

When the request servlet is initialized, it reads the `*instance_name*.xml` configuration file. One of the configuration blocks in the XML file defines all of the adapters. The init() method of the request servlet initializes all defined adapters.

The MQSeries listener receives XML-based MQSeries messages from remote programs and dispatches the requests to the non-HTTP adapter manager.

The Job Scheduler does not require a protocol listener.

Adapters

WebSphere Commerce adapters are device-specific components that perform processing functions before passing a request to the Web controller. Examples of processing tasks performed by an adapter include:

- Instructing the Web controller to process the request in a manner specific to the type of device. For example, a pervasive computing (PvC) device adapter can instruct the Web controller to ignore HTTPS checking in the original request.
- Transforming the message format of the inbound request into a set of properties that WebSphere Commerce commands can parse.
- Providing device-specific session persistence.

The following diagram shows the implementation class hierarchy for the WebSphere Commerce adapter framework.

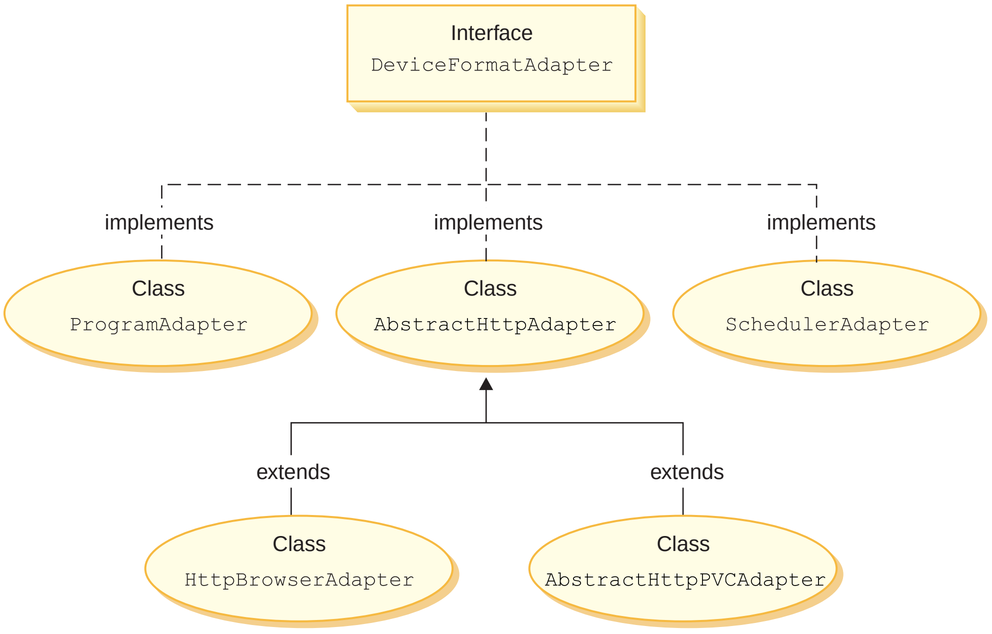


Figure 4.

As displayed in the preceding diagram, all adapters implement the DeviceFormatAdapter interface. The following are the adapters that are used by the WebSphere Commerce run-time environment:

Program adapter
The program adapter provides support for remote programs invoking WebSphere Commerce commands. The program adapter receives requests and uses a message mapper to convert the request into a CommandProperty object. After the conversion, the program adapter uses the CommandProperty object and executes the request.

Scheduler adapter
The scheduler adapter provides support for WebSphere Commerce commands that are run as background jobs.

HTTP browser adapter
The HTTP browser adapter provides support for requests to invoke WebSphere Commerce commands that are received from HTTP browsers.

HTTP PvC adapter
This is an abstract adapter class that can be used to develop specific PvC device adapters. For example, if you needed to develop an adapter for a particular cellular phone application, you would extend from this adapter.

If required, the adapter framework can be extended in the following two ways:

- Create an adapter for a specific PvC device (for example, create an HttpIModePVCAdapterImpl class to provide support for i-mode devices). An adapter of this type must extend the AbstractHttpAdapterImpl class.
- Create a new adapter that connects to a new protocol listener. This new adapter must implement the DeviceFormatAdapter interface.

Web controller

The WebSphere Commerce Web controller is an application container that follows a design pattern similar to that of an EJB container. This container simplifies the role of commands, by providing such services as session management (based upon the session persistence established by the adapter), transaction control, access control and authentication.

The Web controller also plays a role in enforcing the programming model for the commerce application. For example, the programming model defines the types of commands that an application should write. Each type of command serves a specific purpose. Business logic must be implemented in controller commands and view logic must be implemented in view commands. The Web controller expects the controller command to return a view name. If a view name is not returned, an exception is thrown.

For HTTP requests, the Web controller performs the following tasks:

- Begins the transaction using the UserTransaction interface from the javax.transaction package.
- Gets session data from the adapter.
- Determines whether the user must be logged on before invoking the command. If required, it redirects the user's browser to a logon URL.
- Checks if secure HTTPS is required for the URL. If it is required but the current request is not using HTTPS, it redirects the Web browser to an HTTPS URL.
- Invokes the controller command and passes it the command context and input properties objects.
- If a transaction rollback exception occurs and the controller command can be retried, it retries the controller command.
- A controller command normally returns a view name when there is a view command to be sent back to the client. The Web controller invokes the view command for the corresponding view. There are a number of ways to form a response view. These include redirecting to a different URL, forwarding to a JSP template or writing an HTML document to the response object.
- Saves the session data.
- Commits the session data.
- Commits the current transaction if it is successful.
- Rolls back the current transaction in case of failure (depending upon circumstances).

Commands

WebSphere Commerce commands are beans that contain the programming logic associated with handling a particular request.

There are four main types of WebSphere Commerce commands:

Controller command

A controller command encapsulates the logic related to a particular business process. Examples of controller commands include the *OrderProcessCmd* command for order processing and the *UserRegistrationAddCmd* command for creating new registered users. In general, a controller command contains the control statements (for example, if, then, else) and invokes task commands to perform individual tasks in the business process. Upon completion, a controller command returns a view name. The Web controller then determines the appropriate implementation class for the view command and executes the view command.

Task command

A task command implements a specific unit of application logic. In general, a controller command and a set of task commands together implement the application logic for a URL request. A task command is executed in the same container as the controller command.

Data bean command
A data bean command is invoked by a JSP template when a data bean is instantiated. The primary function of a data bean command is to populate the data bean with data.

View command
A view command composes a view as a response to a client request. There are three types of view commands:

Redirect view command
This view command sends the view using a redirect protocol, such as the URL redirect. A controller command should return a view command in this view type to return a view using a redirect protocol. When a redirect protocol is used, it changes the URL stacks in the browser. When a reload key is entered, the redirected URL executes instead of the original URL.

Direct view command
This view command sends the response view directly to the client.

Forward view command
This view command forwards the view request to another Web component, such as a JSP template.

There are three ways in which a view command can be invoked:

- A controller command specifies a view command name when the request has successfully completed.
- A client requests a view directly.
- A command detects an error and an error task must be executed to process the error. The command throws an exception with a view command name. When the exception propagates to the Web controller, it executes the error view command and returns the response to the client.

WebSphere Commerce entity beans

Entity beans are the persistent, transactional commerce objects provided by WebSphere Commerce. If you are familiar with the commerce domain, entity beans represent WebSphere Commerce data in an intuitive way. That is, rather than having to understand the whole the database schema, you can access data from an entity bean which more closely models concepts and objects in the commerce domain. You can extend existing entity beans. In addition, for your own application-specific business requirements, you can deploy entirely new entity beans.

Entity beans are implemented according to the Enterprise JavaBeans (EJB) component model.

For more information about entity beans, refer to "Implementation of WebSphere Commerce entity beans" on page 45.

Data beans

Data beans are Java beans that are primarily used by Web designers. Most commonly, they provide access to a WebSphere Commerce entity. A Web designer can place these beans on a JSP template, allowing dynamic information to be populated on the page at display time. The Web designer need only understand what data the bean can provide and what data the bean requires as input. Consistent with the theme of separating display from business logic, there is no need for the Web designer to understand how the bean works.

Data bean manager

When a WebSphere Commerce data bean is inserted into a JSP template using WebSphere Studio Page Designer, a line of code is generated that populates the data bean, at run time, by invoking the data bean manager.

The following is a sample of code from Page Designer:

```
com.ibm.commerce.beans.DataBeanManager.activate(data_bean, request)
```

JavaServer Pages templates

JSP templates are specialized servlets that are typically used for display purposes. Upon completion of a URL request, the Web controller invokes a view command that invokes a JSP template. A client can also invoke a JSP template directly from the browser without an associated command. In this case, the URL for the JSP template must include the request servlet in its path, so that all of the data beans required by a JSP template can be activated within a single transaction. The request servlet can forward a URL request to a JSP template and execute the JSP template within a single transaction.

The data bean manager rejects any URL for a JSP template that does not include the request servlet in its path. For more information about protection of JSP templates and other resources, refer to Chapter 4, "Access control" on page 85.

Instance_name.xml configuration file

The *Instance_name*.xml configuration file sets configuration information for the instance. It is read when the request servlet is initialized.

Summary for a request

This section provides a summary of the interaction flow between components when forming a response to a request.

A description of each of the steps follows the diagram.

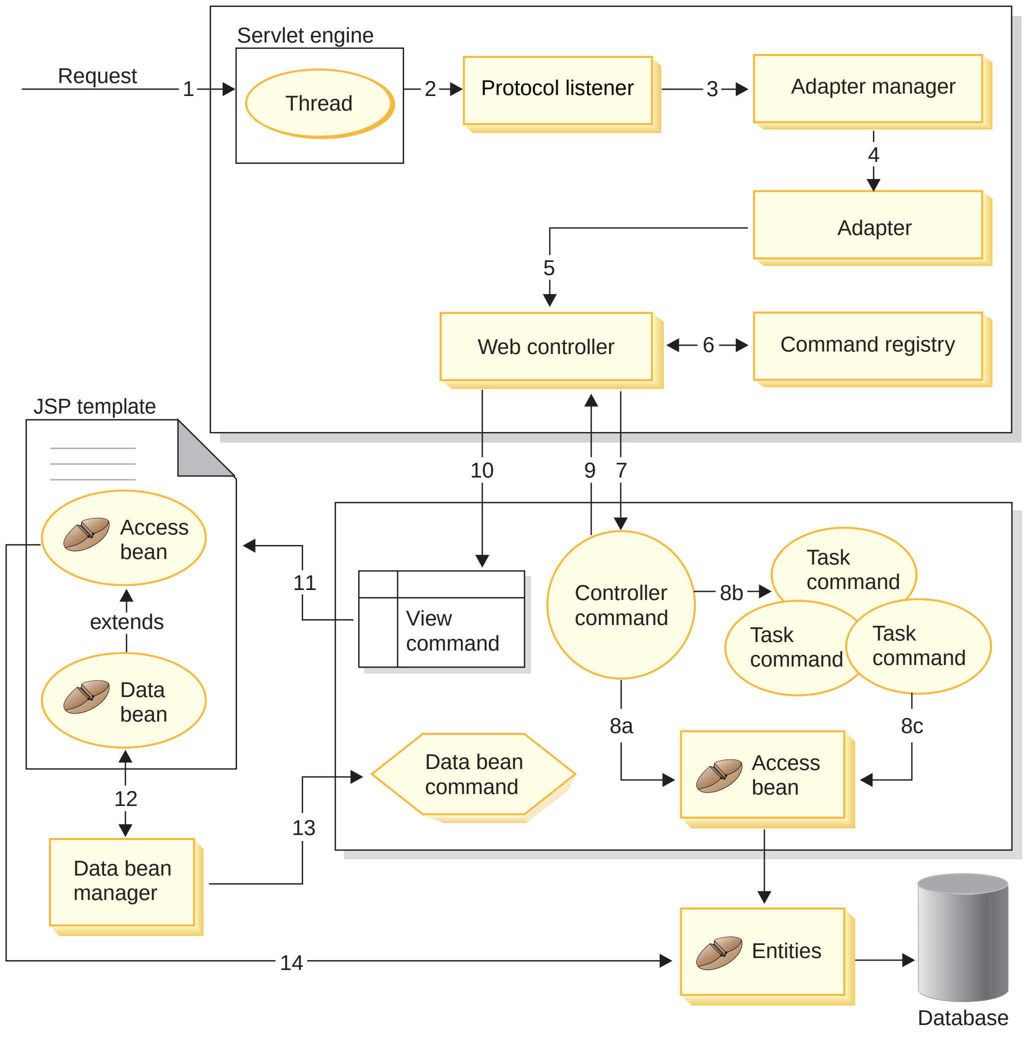


Figure 5.

The following information corresponds to the preceding diagram.

1. The request is directed to the servlet engine by the WebSphere Application Server plug-in.
2. The request is executed in its own thread. The servlet engine dispatches the request to a protocol listener. The protocol listener can be the HTTP request servlet or the MQ Listener.
3. The protocol listener passes the request to the adapter manager.

4. The adapter manger determines which adapter is capable of handling the request and then forwards the request to the appropriate adapter. For example, if the request came from an Internet browser, the adapter manager forwards the request to the HTTP browser adapter.
5. The adapter passes the request to the Web controller.
6. The Web controller determines which command to invoke, by querying the command registry.
7. Assuming that the request requires the use of a controller command, the Web controller invokes the appropriate controller command.
8. Once a controller command begins execution, there are a few possible paths:
   a. The controller command can access the database using an access bean and its corresponding entity bean.
   b. The controller command can invoke one or more task commands. Then task commands can access the database, using access beans and their corresponding entity beans (shown in 8(c)).
9. Upon completion, the controller command returns a view name to the Web controller.
10. The Web controller looks up the view name in the VIEWREG table. It invokes the view command implementation that is registered for the device type of the requester.
11. The view command forwards the request to a JSP template.
12. Within the JSP template, a data bean is required to retrieve dynamic information from the database. The data bean manager activates the data bean.
13. The data bean manager invokes a data bean command, if required.
14. The access bean from which the data bean is extended accesses the database using its corresponding entity bean.

Key differences between customization in previous releases

Beginning with WebSphere Commerce Suite Version 5.1, the WebSphere Commerce Server has been written entirely in Java. Therefore, you must use Java to customize functionality. This is very different from the model that had been used in WebSphere Commerce Suite, Version 4.1 (and earlier versions of Net.Commerce™) in which C++ and Net.Data® macros were used for customization.

The following table summarizes the major changes.

	Version 4.1 (and earlier versions) in C++	**Version 5.1 (and later versions) in Java**
Application models	Commands	Controller commands
	Tasks	Task commands
	Overridable functions	Task commands
	View tasks	View commands
	Error tasks	View commands
Display models	Net.Data macros	JSP templates, data beans, data bean commands, and view commands
Persistence models	Net.Data and ODBC	Entity beans
URL dispatcher	WebSphere Commerce C++ URL dispatcher	WebSphere servlet engine
Task models	System processes	Java threads
Command adapter	None	Web controller and adapters

This publication does not describe the migration process. For more information about migration, refer to the *WebSphere Commerce Migration Guide*.

Part 2. Programming model

Chapter 2. Design patterns

A variety of design patterns and mechanisms are used to develop the WebSphere Commerce framework. WebSphere Commerce provides a high-level design pattern to which each WebSphere Commerce application should adhere. The following design patterns are discussed in this chapter:

- The model-view-controller design pattern
- The command design pattern
- The display design pattern

Model-View-Controller design pattern

The model-view-controller (MVC) design pattern specifies that an application consist of a data model, presentation information and control information. The pattern requires that each of these be separated into different objects.

The *model* (for example, the data information) contains only the pure application data; it contains no logic describing how to present the data to a user.

The *view* (for example, the presentation information) presents the model's data to the user. The view knows how to access the model's data, but it does not know what this data means or what the user can do to manipulate it.

Finally, the *controller* (for example, the control information) exists between the view and the model. It listens to events triggered by the view (or another external source) and executes the appropriate reaction to these events. In most cases, the reaction is to call a method on the model. Since the view and the model are connected through a notification mechanism, the result of this action is then automatically reflected in the view.

Most applications today follow this pattern, many with slight variations. For example, some applications combine the view and the controller into one class because they are already very tightly coupled. All of the variations strongly encourage separation of data and its presentation. This not only makes the structure of an application simpler, it also enables code reuse.

Since there are many publications describing the pattern, as well as numerous samples, this document does not describe the pattern in great detail.

The following diagram shows how the MVC design pattern applies to WebSphere Commerce.

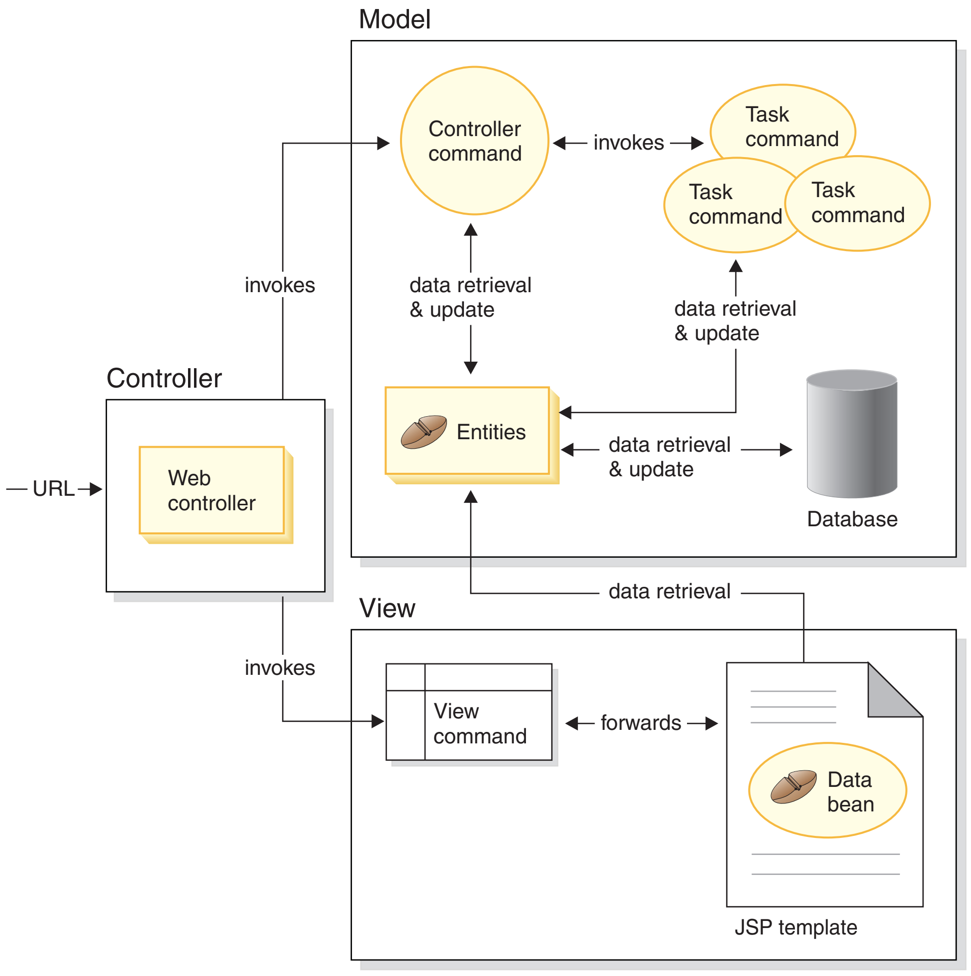


Figure 6.

Command design pattern

The WebSphere Commerce Server accepts requests from browser-based thin-client applications, as well as from other remote applications. For example, a request may come from a remote procurement system, or from another commerce server.

All requests, in their variety of formats, are translated into a common format by the adapters that make up the adapter framework. Once the requests are in this common format, they can be understood by WebSphere Commerce commands.

Commands are beans that perform business logic. They represent procedural logic either in the form of high-level process logic or discrete business logic tasks. A process-based command acts as a controller that spans multiple entities and other commands, while a task command performs a specific task and may only access a single object.

Command framework

Command beans follow a specific design pattern. Every command includes both an interface class (for example, `CategoryDisplayCmd`) and an implementation class (for example, `CategoryDisplayCmdImpl`). From a caller's perspective, the invocation logic involves setting input properties, invoking an `execute()` method, and retrieving output properties.

From the perspective of the command implementer, commands follow the WebSphere command framework, which implements the standard command design pattern allowing a level of indirection between the caller and the implementation. The key mechanisms enabled within this level of indirection include:

1. The ability to invoke an access control policy manager that determines if the user is allowed to invoke the command.
2. The ability to execute a different command implementation for different stores, based upon the store identifier.
3. The ability to execute a different view implementation based upon the device type of the requester.

The following diagram shows a conceptual overview of the interfaces for the four main types of commands:

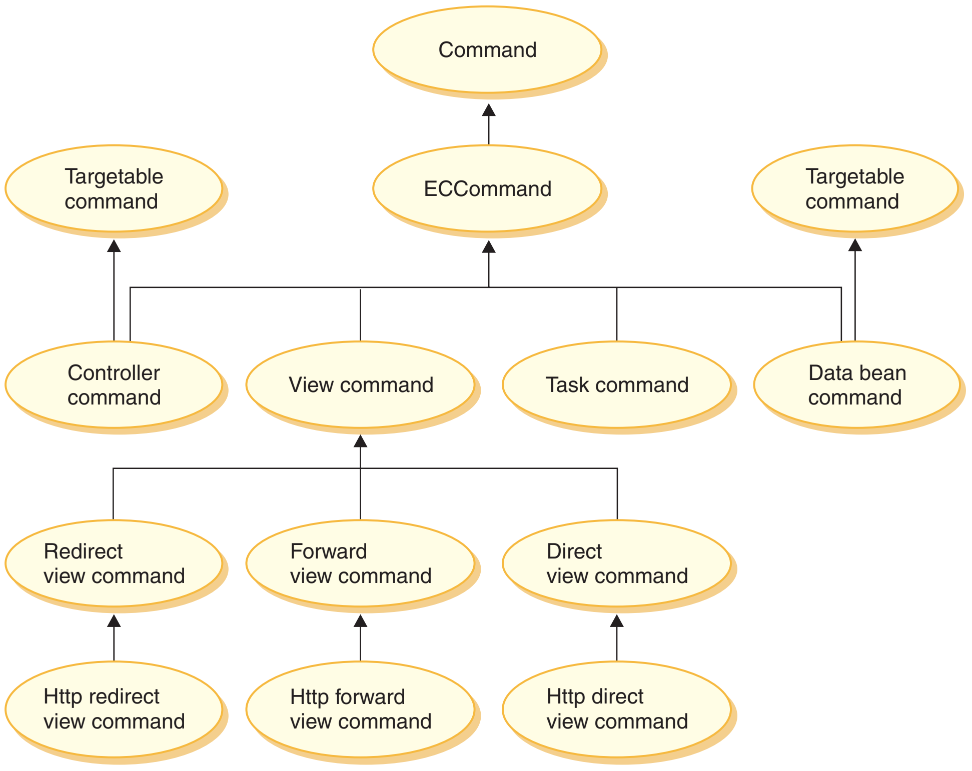


Figure 7.

Controller command

A controller command encapsulates the logic related to a particular business process. Examples of controller commands include the *OrderProcessCmd* command for order processing and the *UserRegistrationAddCmd* command for creating new registered users. In general, a controller command contains the control statements (for example, if, then, else) and invokes task commands to perform individual tasks in the business process. Upon completion, a controller command returns a view name. The Web controller then determines the appropriate implementation class for the view task and executes the view task.

While a controller command is a targetable command, only the local target is supported.

Task command

A task command implements a specific unit of application logic. In general, a controller command and a set of task commands together implement the application logic for a URL request. A task command is executed in the same container as the controller command.

Data bean command

A data bean command is invoked by a JSP page when a data bean is instantiated. The primary function of a data bean command is to populate the fields of the data bean.

While a data bean command is a targetable command, only the local target is supported.

View command

A view command composes a view as a response to a client request. There are three ways in which a view command can be invoked:

- A controller command specifies a view command name on successful completion of the request.
- A client can request a view directly.
- A controller or task command detects an error and decides that an error task must be executed to process the error and throws an exception with a view command name. When the exception propagates to the Web controller, it executes the view command and returns the response to the client.

There are three types of view commands:

Redirect view command

This view command sends the view using a redirect protocol, such as the URL redirect. A controller command should return a view command of this view type when a redirect protocol is required. When a redirect protocol is used, it changes the URL stacks in the browser. When a reload key is entered, the redirected URL executes instead of the original URL.

Direct view command

This view command sends the response view directly to the client.

Forward view command

This view command forwards the view request to another Web component, such as a JSP template.

Command factory

In order to create new command objects, the caller of the command uses the *command factory*. The command factory is a bean that is used to instantiate commands. It is based on the factory design pattern, which defers instantiation of an object away from the invoking class, to the factory class that understands which implementation class to instantiate.

The factory provides a smart way to instantiate new objects. In this case, the command factory provides a way to determine the correct implementation class when creating a new command object, based upon the individual store. The command interface name and the particular store identifier are passed into the new command object, upon instantiation.

There are two ways for the implementation class of a command to be specified. A default implementation class can be specified directly in the code for the command interface, using the `defaultCommandClassName` variable. For example, the following code exists in the CategoryDisplayCmd interface:

```
String defaultCommandClassName =
     "com.ibm.commerce.catalog.commands.CategoryDisplayCmdImpl"
```

The second way to specify the implementation class is to use the WebSphere Commerce command registry. The command registry should always be used when the implementation class varies from one store to another. More information about the command registry can be found on page 26.

In the case where a default implementation class is specified in the code for the interface and a different implementation class is specified in the command registry, the command registry takes precedence.

The syntax for using the command factory is as follows:

```
cmd = CommandFactory.createCommand(interfaceName, commandContext.getStoreId())
```

where *`interfaceName`* is the interface name for the new command bean and the getStoreId method determines the store for which the command is to be used.

Note: The syntax for using the command factory to create business policy commands is different from the preceding code snippet. For more information about using the command factory to create business policy commands, refer to “Invoking the new business policy” on page 173.

Command flow

This section provides an overview of the logical flow between commands and the WebSphere Commerce database. The following diagram and descriptions depict this flow.

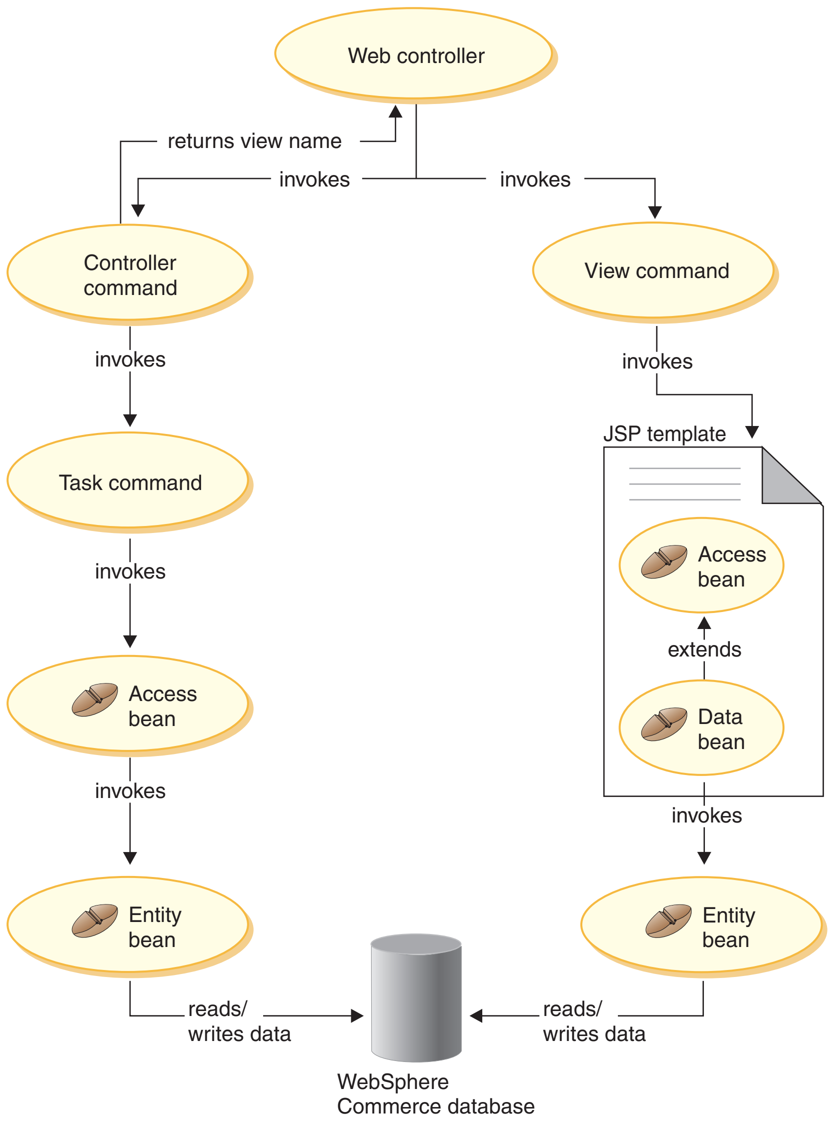


Figure 8.

When the Web controller receives a request, it determines whether the request requires the invocation of a controller command or a view command. In either case, the Web controller also determines the implementation class for the command, and then invokes it.

First examine the left side of the diagram. Since controller commands encapsulate the logic for a business process, they frequently invoke individual task commands to perform specific units of work in the business process.

Access beans are invoked when information in the database must be retrieved or updated. Either a task or controller command can invoke access beans. Requests then flow from access beans to entity beans that can read from, and write to, the WebSphere Commerce database.

Now examine the right side of the diagram. A view command is invoked by the Web controller, either when a controller command has completed processing and it returns the name of a view command to invoke, or when an error occurs and an error view must be displayed.

View commands typically invoke a JSP template to display the response to the client. Within the JSP template, data beans are used to populate dynamic information onto the page. Data beans are activated by the data bean manager. The data bean (which extends from an access bean) invokes its corresponding entity bean. When accessed indirectly from a JSP template, an entity bean typically retrieves information from the database (rather than writing information to the database).

Command registration framework

WebSphere Commerce controller and task commands are registered in the command registry. The following three tables comprise the command registry:

- URLREG
- CMDREG
- VIEWREG

Note: Business This section does not apply to the registration of business policy commands. For information about registering new business policy commands, refer to “Registering the new business policy and business policy command” on page 157.

URLREG table

The URLREG table maps URIs (Universal Resource Indicator) to controller command interfaces. URIs provide a simple and extensible mechanism for resource identification. A URI is a relatively short string of characters used to identify an abstract or physical resource. In WebSphere Commerce, the URI contains only command information. In the following URL, the URI section is shown in bold:

```
http://hostname/webapp/wcs/stores/servlet/StoreCatalogDisplay?
     storeId=store_Id&catalogId=catalog_Id&langId=-1
```

While there is a one-to-one mapping between a URI and an interface name, each store can specify whether HTTPS or AUTHENTICATION is required for the command. For each inbound URL request, the Web controller looks up the interface name for the controller command and then uses that name to determine the correct implementation class, as registered in the CMDREG table.

The following table describes information contained in the URLREG database table.

Column name	Description	Comments
URL	URI name	For example `MyNewCommand` or `ProductDisplay`
STOREENT_ID	Store entity identifier	This can be set to `0` to use the command for all stores, or to a unique store identifier to indicate that the command is used only for a particular store.
INTERFACENAME	Controller command interface name	For example `com.ibm.commerce.catalog.commands.GetProductDisplay.TemplateCmd`
HTTPS	Secure HTTP required for this URL request	Use `1` when HTTPS is required and `0` when it is not.
DESCRIPTION	Description of URI	For example, `This command is used for testing purposes.`
AUTHENTICATED	User log on is required for this URL request	Use `1` when authentication is required and `0` when it is not.
INTERNAL	Indicates whether or not the command is internal to WebSphere Commerce	Use `1` when the command is internal and `0` when it is external.

When the Web controller receives a URL request, it retrieves the interface name for the requested controller command and uses it to look up the implementation class name from the CMDREG table. It also determines if HTTPS is required for the URL request by checking the HTTPS column in the URLREG table.

Only commands that are invoked by way of URL requests need to be registered in the URLREG table. Therefore, only controller commands must be registered here, not task or view commands.

The following SQL statement creates an entry for MyNewControllerCommand which is used by a particular store (whose store identifier is `5`):

```
insert into URLREG (URL, STOREENT_ID, INTERFACENAME, HTTPS,
DESCRIPTION, AUTHENTICATED) values ('MyNewControllerCommand',
5,'com.ibm.commerce.commands.MyNewControllerCommand',0,
'This is a test command.',null)
```

The generic syntax for the insert statement is as follows:

```
insert into table_name (column_name1,column_name2, ... ,column_namen)
values (column1_value,column2_value,...,columnn_value)
```

String values should be enclosed in single quotes.

CMDREG table

CMDREG is the command registration table. This table provides a mechanism for mapping the command interface to its implementation class. Multiple implementations of an interface allow for command customization on a per store basis.

Only controller commands and task commands are registered in the CMDREG table. View commands are registered in the VIEWREG table.

The following describes information contained in the CMDREG database table.

Column name	Description	Comments
STOREENT_ID	Store entity identifier	This can be set to `0` to use the command for all stores, or to a unique store identifier to indicate that the command is used only for a particular store.
INTERFACENAME	Command interface name	This defines the interface; use the same name as you did in the URLREG table.
DESCRIPTION	Description of this command	For example, `This command is used for testing purposes`.
CLASSNAME	Command implementation class name	Typically the interface name with ″Impl″ appended to end.
PROPERTIES	Default name-value pairs set as input properties to the command	Format is same as URL query string. For example ″parm1=val1&parm2=val2″
LASTUPDATE	Last update on this command entry	
TARGET	Command target name. This is where the command is actually executed.	Only local target is supported.

In general, when you create a new controller or task command, you should create corresponding entry in the CMDREG table. For example, the following SQL statement creates an entry for MyNewCommand which is used by a particular store (whose store identifier is 5):

```
insert into CMDREG (STOREENT_ID, INTERFACENAME, DESCRIPTION, CLASSNAME,
PROPERTIES, LASTUPDATE, TARGET) values
(5,'com.ibm.commerce.catalog.commands.MyNewCommand', 'This is a test
command', 'com.ibm.commerce.catalog.commands.MyNewCommandImpl',
'myDefaulParm1=myDefaultVal1', '0000-12-01', 'Local')
```

The generic syntax for the insert statement is as follows:

```
 insert into table_name (column_name1,column_name2, ... ,column_namen)
values (column1_value,column2_value,...,columnn_value)
```

String values should be enclosed in single quotes.

If the command you are writing always uses the same implementation class, you do not necessarily have to register the command in the CMDREG table. In this case, you can use the defaultCommandClassName attribute in the interface to specify the implementation class. For example, in the code for the interface, you would include the following:

```
String defaultCommandClassName =
     "com.ibm.commerce.command.MyNewCommandImpl"
```

If you specify the implementation class in this manner, you cannot pass default properties to the implementation class and the same implementation class must be used for all stores.

Example of a registered controller command

Consider a scenario in which your site has two stores: StoreA and StoreB. Each store has different security requirements for the MyUrl controller command as well as different implementations of the command. This section shows how the command registry is used to enable this customization.

The following table shows the entries for StoreA and StoreB in the URLREG table:

Column name	Entry for StoreA	Entry for StoreB
URL	MyUrl	MyUrl
STOREENT_ID	11	22
INTERFACENAME	com.ibm.commerce. mycommands.myUrl	com.ibm.commerce. mycommands.myUrl
HTTPS	1	1

Column name	Entry for StoreA	Entry for StoreB
DESCRIPTION	Example entry in the URLREG table.	Example entry in the URLREG table.
AUTHENTICATED	1	0
INTERNAL	null	null

Note: The spaces in values for INTERFACENAME are for display purposes only. Each value is actually one continuous string.

Based upon entries in the URLREG table, the Web controller determines that the interface name for the MyURL URI is com.ibm.commerce.mycommands.MyUrl. It also determines that StoreA requires the command to be executed using both HTTPS and authentication, but StoreB requires HTTPS only. The values for HTTPS and authentication are used by the Web controller, not by the interface.

The following diagram shows this flow:

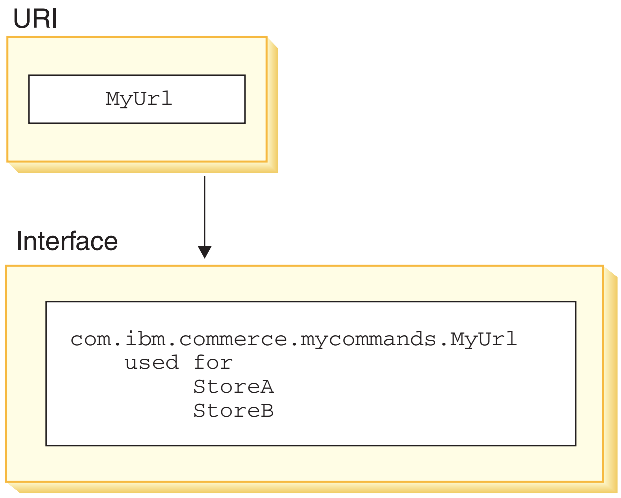


Figure 9.

The following table shows the entries in the CMDREG table. Only columns required for the purpose of this example are displayed:

Column name	Entry for StoreA	Entry for StoreB
STOREENT_ID	11	22
INTERFACENAME	com.ibm.commerce. mycommands.myUrl	com.ibm.commerce. mycommands.myUrl

Column name	Entry for StoreA	Entry for StoreB
CLASSNAME	com.ibm.commerce. mycommands. myUrlStoreAImpl	com.ibm.commerce. mycommands.myUrlStoreBImpl

Note: The spaces in values for INTERFACENAME and CLASSNAME are for display purposes only. Each value is actually one continuous string.

Based upon entries in the CMDREG table, the Web controller determines that for StoreA, the implementation class for the com.ibm.commerce.mycommands.MyUrl interface is com.ibm.commerce.mycommands.MyUrlStoreAImpl. It also determines for StoreB, the implementation class for the same interface is com.ibm.commerce.mycommands.MyUrlStoreBImpl. The following diagram shows this flow:

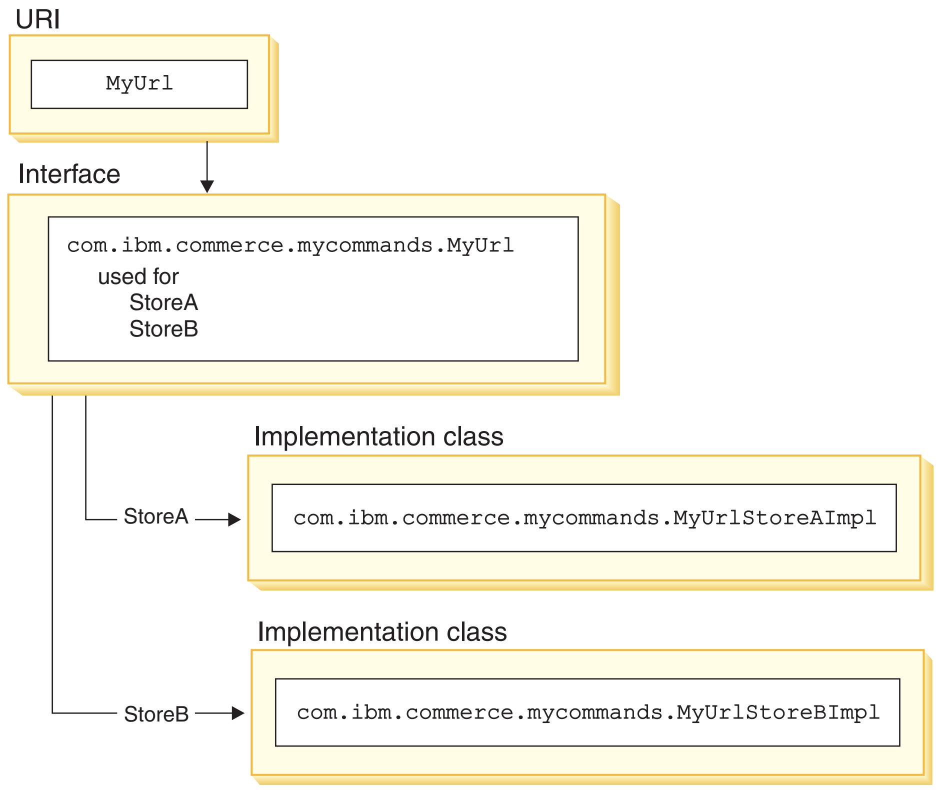

URI
MyUrl
Interface
com.ibm.commerce.mycommands.MyUrl
used for
StoreA
StoreB
Implementation class
StoreA
com.ibm.commerce.mycommands.MyUrlStoreAImpl
Implementation class
StoreB
com.ibm.commerce.mycommands.MyUrlStoreBImpl

Figure 10.

VIEWREG table

The VIEWREG table allows registration of device-specific view command implementations. Using this table, multiple implementations of a view can be registered. The command framework is then capable of returning different views to various clients.

When a view command name is returned from a controller command or specified in an exception, the Web controller determines the view command class from the VIEWREG table. Multiple view command names can be mapped to the same implementation class.

Column name	**Description**	**Comments**
VIEWNAME	View name	For example, `AddressForm`
DEVICEFMT_ID	Device type identifier	Available options include: • BROWSER (default value) • I_MODE • E-mail • MQXML • MQNC
STOREENT_ID	Store entity identifier	This can be set to `0` to use the command for all stores, or to a unique store identifier to indicate that the command is used only for a particular store.
INTERFACENAME	View command interface name	Default options are ForwardView, DirectView and RedirectView.
CLASSNAME	View command implementation class name	Can use the default implementation.
PROPERTIES	Default name-value pairs set as input properties to the command	If the same page is always displayed, set the JSP file name in this property (`docname=`*`jsp_name`*`.jsp`). If the same JSP template is used for all stores, set `storeDir=no` to prevent a store specific directory from being used. If a generic user can invoke the command, set `isGeneric=true`.
DESCRIPTION	Description of this command	

Column name	Description	Comments
HTTPS	Secure HTTP required for this URL request	Use 1 when HTTPS is required and `0` when it is not.
LASTUPDATE	Last update on this entry	
INTERNAL	Indicates whether or not the command is internal to WebSphere Commerce	Use 1 when the command is internal and `0` when it is external.

When you create a new view command, you may need to create a corresponding entry in the VIEWREG table. If one of the following conditions is met, the view command must be registered in the VIEWREG table:

- The view command is executed under access control
- There are multiple implementations of the view command
- Properties are set in the PROPERTIES column

Registered view commands can either be accessed through the command registry using the view name, or directly by using the actual display file name. Views that are not registered in the VIEWREG table can only be accessed when a client uses the actual display file name.

Consider the example of a view named *MyView*, with the VIEWREG entry as follows:

Column name	Entry
VIEWNAME	`MyView`
DEVICEFMT_ID	`BROWSER`
STOREENT_ID	`0`
INTERFACENAME	`com.ibm.commerce.commands.ForwardViewCommand`
CLASSNAME	`com.ibm.commerce.commands.HTTPForwardViewCommandImp`
PROPERTIES	`docname=MyView.jsp`
DESCRIPTION	`An example for calling a JSP template using either the view name or directly from a URL.`
HTTPS	`0`
LASTUPDATE	`2000-11-30`
INTERNAL	`0`

Since MyView is a registered view, a client can access the view either by using the command name, or by substituting the actual display file name for the command name. Using the view name, a sample URL is:

```
http://hostname.com/webapp/wcs/stores/servlet/MyView
```

and using the file name, a sample URL is:

```
http://hostname.com/webapp/wcs/stores/servlet/MyView.jsp
```

If there is a possibility that a client will invoke a registered view directly (using the display file name), you must register the command using the same name for the view as the actual display file name, as shown in this example (MyView and MyView.jsp).

A view that is not registered in the table can only be invoked using the display file name. Therefore, if there is an unregistered view that uses the file MyUnregisteredView.jsp, the URL to access this view is as follows:

```
http://hostname.com/webapp/wcs/stores/servlet/MyUnregisteredView.jsp
```

The following example SQL statement creates an entry for *MyNewViewCommand* which is used by one particular store:

```
insert into VIEWREG (VIEWNAME, DEVICEFMT_ID, STOREENT_ID,
INTERFACENAME, CLASSNAME, PROPERTIES, DESCRIPTION,HTTPS, LASTUPDATE,
INTERNAL) values ('MyNewViewCommand', 'BROWSER', 5,
'com.ibm.commerce.command.ForwardViewCommand',
'com.ibm.commmerce.command.HttpForwardViewCommandImpl',
'docname=MyNewViewCommand.jsp', 'A test view command.', 0,
'0000-12-01', 0)
```

The following table provides another sample VIEWREG table with key information:

COMMAND - NAME	**DEVICE - FMT_ID**	**INTERFACE - NAME**	**CLASSNAME**	**PROPERTIES**
ProductDisplayView	BROWSER	Forward View Command	HttpForwardView CommandImpl	
InterestItemAddView	BROWSER	Redirect View Command	HttpRedirectView CommandImpl	docname =item.jsp
InterestItemDeleteView	BROWSER	Redirect View Command	HttpRedirectView CommandImpl	docname =item.jsp
GenericApplication Error	BROWSER	Redirect View Command	HttpRedirectView CommandImpl	docname =usererr.jsp
GenericSystemError	BROWSER	Redirect View Command	HttpRedirectView CommandImpl	docname =syserr.jsp

COMMAND - NAME	DEVICE - FMT_ID	INTERFACE - NAME	CLASSNAME	PROPERTIES
Logon	BROWSER	Forward View Command	HttpForwardView CommandImpl	docname= logon.jsp & storeDir=no

Note: Any spaces in the values for COMMANDNAME, INTERFACENAME, CLASSNAME and PROPERTIES are for display purposes only. Each value is actually one continuous string. The hyphens in the column names are also for display purposes.

The preceding table illustrates the following scenarios:

- The *ProductDisplayView* view name is returned to the Web controller by a controller command (ProductDisplay in this case). The Web controller determines the view command interface and class names using the ProductDisplayView view command name and its device identifier. A view command can have different implementation classes for different stores and device identifiers. The interface name, however, should remain the same, since it defines the view command type.
- The InterestItemAdd and InterestItemDelete commands return the *InterestItemAddView* and *InterestItemDeleteView* view names to the Web controller, respectively. Both commands require redirect views, therefore, the view command interface name for both views is the RedirectViewCommand. There is a common JSP template registered for both views. The Web controller fetches the properties (docname=item.jsp) and passes them to the view command (HttpRedirectViewCommandImpl).
- If a controller or task command throws an ECApplication exception for a bad user parameter, the following may occur:
  - If there is a view specified within the controller command that should be called in the case of an application exception, the entry for that view is retrieved from the VIEWREG table and processed accordingly.
  - If a view is not specified, the GenericApplicationError command is called and the JSP template registered in the database is displayed. Using the preceding table as an example, this would result in the display of the `usererr.jsp` template.
- If a controller or task command throws an ECSystem exception for a system exception, the following may occur:
  - If there is a view specified within the controller command that should be called in the case of a system exception, the entry for that view is retrieved from the VIEWREG table and processed accordingly.

  - If a view is not specified, the GenericSystemError command is called and the JSP template registered in the database is displayed. Using the preceding table as an example, this would result in the display of the `syserr.jsp` template.
- Browser clients can invoke the logon page by entering the logon URL. Since the storeDir property is set to “no”, store-specific information is not included in the path for the JSP template. Hence, the same logon page is displayed for customers at all stores.

Display design pattern

Display pages return a response to a client. Typically, display pages are implemented as JSP templates (the recommended method), however, they can be written directly as servlets.

In order to support multiple device types, a URL access to a view command should use the view name, not the name of the actual JSP file.

The main rationale behind this level of indirection is that the JSP template represents a view. The ability to select the appropriate view (for example, based on locale, device type, or other data in the request context) is highly desirable, especially since a single request often has multiple possible views. Consider the example of two shoppers requesting the home page of a store, one shopper using a typical Web browser and the other using a cellular phone. Clearly, the same home page should not be displayed to each shopper. It is the Web controller’s responsibility to accept the request, then based upon information in the command registration framework, determine the view that each shopper receives.

JSP templates and data beans

A data bean is a Java bean that is used within a JSP template to provide dynamic content. A data bean normally provides a simple representation of a WebSphere Commerce entity bean. The data bean encapsulates properties that can be retrieved from or set within the entity bean. As such, the data been simplifies the task of incorporating dynamic data into JSP templates.

A data bean has a *BeanInfo* class that defines the properties that can be used on the display page. The BeanInfo class also enables the use of data beans in multicultural sites by providing property names in all supported languages of WebSphere Commerce.

A data bean is activated by the following call:

```
com.ibm.commerce.beans.DataBeanManager.activate(DataBean, HTTPServletRequest)
```

WebSphere Studio Page Designer generates the preceding line of code automatically when a WebSphere Commerce data bean is inserted into a JSP template.

Store developers should consider properties of the store and multicultural enablement issues when developing JSP templates. For more information on multicultural enablement, refer to the WebSphere Commerce online help.

JSP templates and data beans security consideration

A particular coding practice for the use of JSP templates and data beans minimizes the chance for malicious users to access your database in an unauthorized manner. Insert, select, update and delete parts of SQL statements should be created at development time. Use parameter inserts to gather run-time input information.

An example of using a parameter insert to collect run-time input information follows:

```
select * from Order where owner =?
```

In contrast, you should avoid using input strings as a way to compose the SQL statement. An example of using an input string follows:

```
select * from Order where owner = "input_string"
```

Types of data beans

A data bean is a Java bean that is mainly used to provide dynamic data in JSP templates. There are two types of data beans: smart data beans and command data beans.

A smart data bean uses a *lazy fetch* method to retrieve its own data. This type of data bean can provide better performance in situations where not all data from the access bean is required, since it retrieves data only as required. Smart data beans that require access to the database should extend from the access bean for the corresponding entity bean and implement the com.ibm.commerce.SmartDataBean interface. For example, the ProductData data bean extends the ProductAccessBean access bean, which corresponds to the Product entity bean.

Some smart data beans do not require database access. For example, the PropertyResource smart data bean retrieves data from a resource bundle, rather than the database. When database access is not required, the smart data bean should extend the SmartDataBeanImpl class.

A command data bean relies on a command to retrieve its data and is a more lightweight data bean. The command retrieves all attributes for the data bean at once, regardless of whether the JSP template requires them. As a result, for JSP templates that use only a selection of attributes from the data bean, a command data bean may be costly in terms of performance time. For JSP templates that require most or all attributes, the command data bean is very convenient.

Command data beans can also extend from their corresponding access beans and implement the com.ibm.commerce.CommandDataBean interface.

Data bean interfaces

Data beans implement one or all of the following Java interfaces:

- com.ibm.commerce.SmartDataBean.
- com.ibm.commerce.CommandDataBean
- com.ibm.commerce.InputDataBean (optional)

Each Java interface describes the source of data from which a data bean is populated. By implementing multiple interfaces, the data bean can access data from a variety of sources. More information about each of the interfaces is provided below.

SmartDataBean interface: A data bean implementing the SmartDataBean interface can retrieve its own data, without an associated data bean command. A smart data bean usually extends from the access bean of a corresponding entity bean. When a smart data bean is activated, the data bean manager invokes the data bean's populate method. Using the populate method, the data bean can retrieve all attributes, except attributes from associated objects. For example, if the data bean extends from an access bean class for an entity bean, the data bean invokes the refreshCopyHelper method. All the attributes from the corresponding entity bean are populated into the smart data bean automatically. However, if the entity bean has associated objects, the attributes from those objects are not retrieved. The main advantages of using smart data beans are:

- Implementation is simple and there is no need to write a data bean command.
- When new fields are added to the entity bean, changes in the data bean are not required. After the entity bean has been modified, the access bean must be regenerated (using the tools in VisualAge for Java). As soon as the access been has been regenerated, all the new attributes are automatically available to the smart data bean.
- Entity beans often contain attributes representing associated objects. For performance reasons, the smart data bean does not automatically retrieve these attributes. Instead, it is preferable to delay retrieval of these attributes until they are required, as shown in the following diagram:

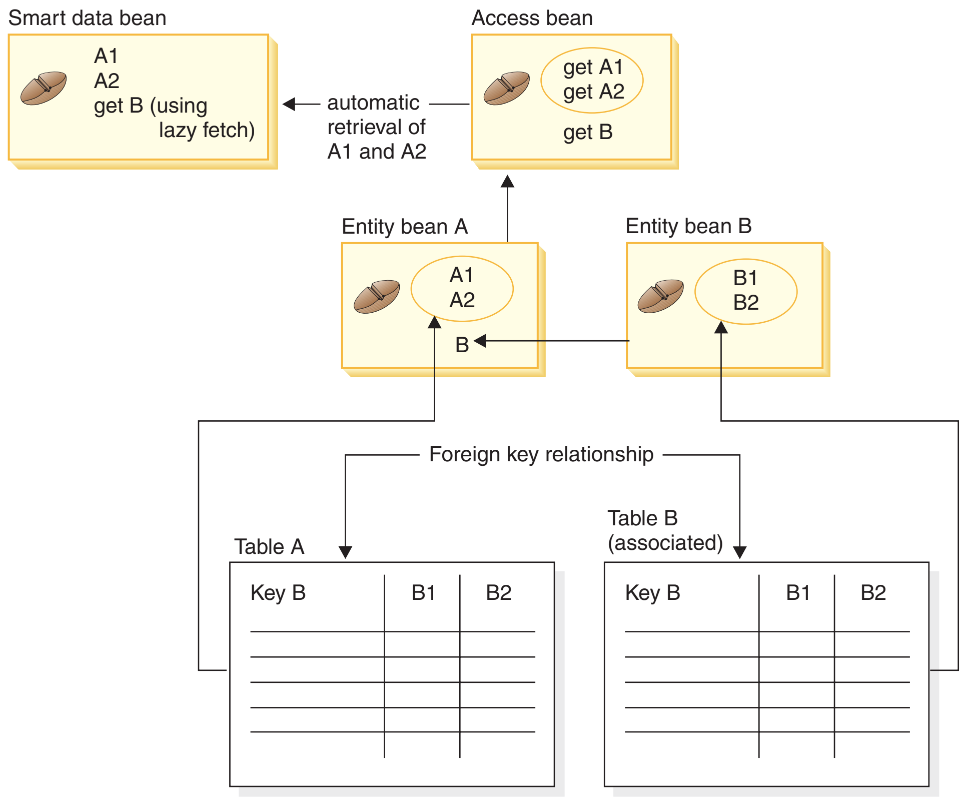


Figure 11.

For more information about implementing a lazy fetch retrieval, refer to "Lazy fetch data retrieval" on page 41.

CommandDataBean interface: A data bean implementing the CommandDataBean interface retrieves data from a data bean command. A data bean of this type is a lightweight object; it relies on a data bean command to populate its data. The data bean must implement the getCommandInterfaceName() method (as defined by the com.ibm.commerce.CommandDataBean interface) which returns the interface name of the data bean command.

InputDataBean interface: A data bean implementing the InputDataBean interface retrieves data from the URL parameters or attributes set by the view command.

Attributes defined in this interface can be used as primary key fields to fetch additional data. When a JSP template is invoked, the generated JSP servlet code populates all the attributes that match the URL parameters, and then activates the data bean by passing the data bean to the data bean manager. The data bean manager then invokes the data bean's setRequestProperties() method (as defined by the com.ibm.commerce.InputDataBean interface) to pass all the attributes set by the view command. It should be noted that WebSphere Studio generates the following code for each data bean that is inserted into pages using Page Designer:

```
com.ibm.commerce.beans.DataBeanManager.activate(DataBean, HTTPServletRequest);
```

BeanInfo class

A data bean is not complete without a BeanInfo class that implements the java.lang.Object.BeanInfo interface. The BeanInfo class is used to provide explicit information about the methods and properties of the data bean. It can be used to hide public run-time methods in the data bean implementation class from the Web designer, or to set the appropriate display string for each of the data bean's attributes.

For more information on implementing a BeanInfo class, refer to the JavaBeans specification from Sun Microsystems.

Data bean activation

Data beans can be activated using either the activate or silentActivate methods that are found in the com.ibm.commerce.beans.DataBeanManager class. The activate method is a full activation method in which the activation event is only successful if all attributes are available. If even one attribute is unavailable, an exception is thrown for the whole activation process.

The silentActivate method does not throw exceptions when individual attributes are unavailable.

Invoking controller commands from within a JSP template

Although invoking controller commands from with a JSP template is not consistent with separating logic from display, you may encounter a situation in which this is required. If so, the `ControllerCommandInvokerDataBean` can be used for this purpose.

Using this data bean, you can specify the interface name of the command to be invoked, or you can directly set the command name to be invoked. You can also set the request properties for the command.

When this data bean is activated by the data bean manager, the controller command is executed and the response properties are available to the JSP template.

Once the controller command has executed, you can execute the view.

Lazy fetch data retrieval

When a data bean is activated, it can be populated by a data bean command or by the data bean's populate() method. The attributes that are retrieved come from the data bean's corresponding entity bean. An entity bean may also have associated objects, which themselves, have a number of attributes.

If, upon activation, the attributes from all the associated objects were automatically retrieved, a performance problem may be encountered. Performance may degrade as the number of associated objects increase.

Consider a product data bean that contains a large number of cross-sell, up-sell or accessory products (associated objects). It is possible to populate all associated objects as soon as the product data bean is activated. However, populating in this manner may require multiple database queries. If not all attributes are required by the page, multiple database queries may be inefficient.

In general, not all attributes are required for a page, therefore, a better design pattern is to perform a lazy fetch as illustrated below:

```
getCrossSellProducts () {
   if (crossSellDataBeans == null)
     crossSellDataBeans= getCrossSellDataBeans();
   return crossSellDataBean;
    }
```

Setting JSP attributes - overview

The WebSphere Commerce program model promotes the MVC design pattern. As such, the presentation for the result of a URL request is separated from controller and task commands. These commands are device independent. They implement business logic and produce data to be returned to the client, without having information about the client. Conversely, a view command is device specific.

While the controller and task commands do not directly compose the view, they do pass information to the view. It is important to understand how information is passed to the view. The following diagram demonstrates how properties are passed between the Web controller, command registry, controller command, and view command:

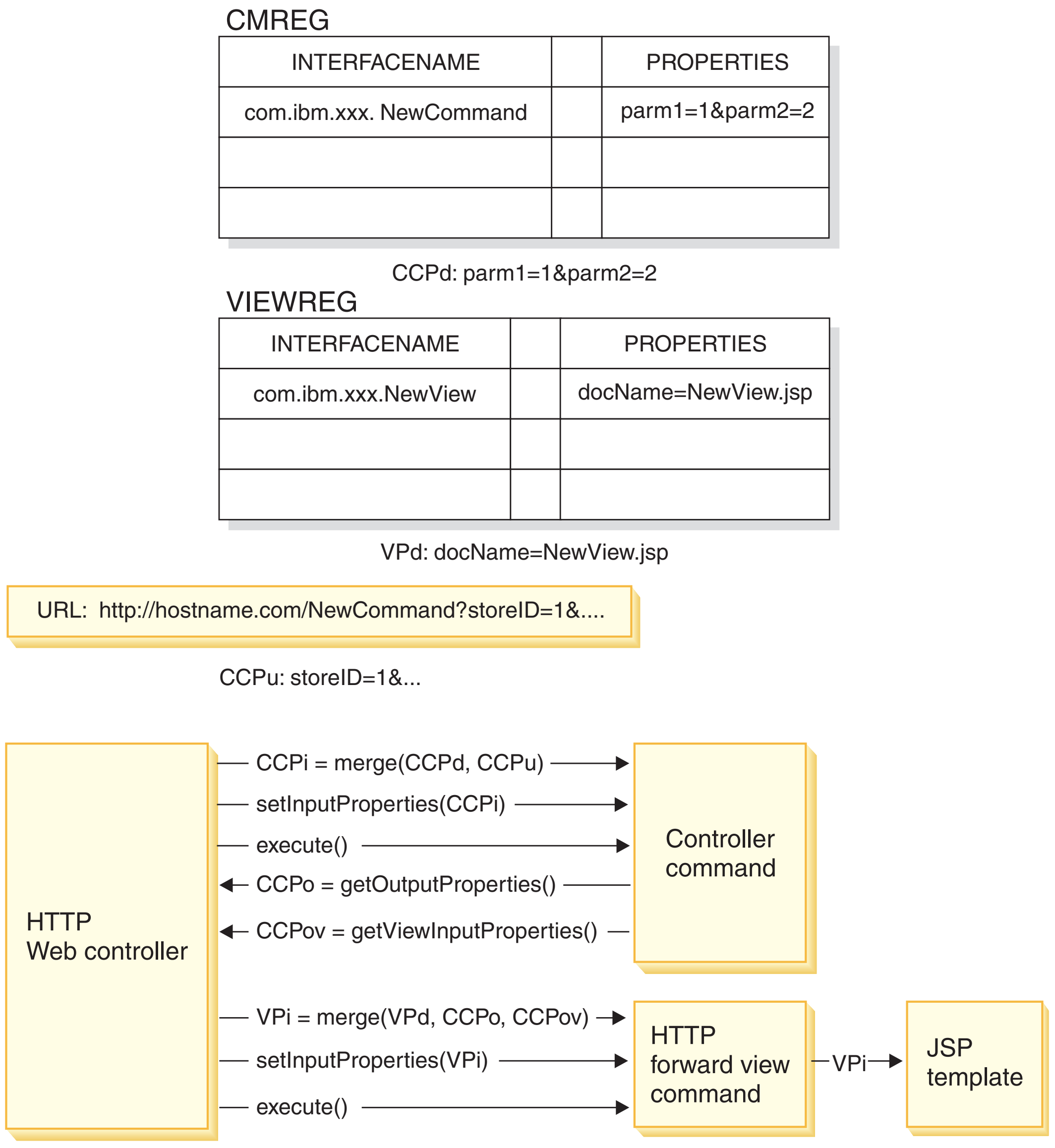


Figure 12.

The preceding diagram shows the following interactions:

- The Web controller merges the input properties from the URL parameters (CCPu) and the entry in the CMDREG table for the controller command (CCPd). This creates CCPi.
- The Web controller passes the merged properties (CCPi) to the controller command and executes the controller command.

- The controller command sets output properties, as CCPo. These are the output properties produced by the command itself. One of the output properties, viewCommandName, is set to the desired view command name. These properties are retrieved by the Web controller using a get method.
- The controller command sets another set of output properties, as CCPov. By default, these are set to the original merged input properties (CCPi). It is possible to customize these properties. For example, it may not be necessary to pass all input parameters to the view command.
- The Web controller merges the three sets of properties, CCPo, CCPov, and VPd (the properties that are registered in the VIEWREG table) into the input properties for the view command (VPi).
- The Web controller sets the merged properties, VPi, and executes the view command.
- The view command sets the attributes to the JSP template from the input properties.

When writing new commands, you do not have explicitly perform the merge of properties. The abstract command classes include a mergeProperties method. For more information about this method, refer to the "Reference" section of the WebSphere Commerce online help.

Required property settings

A controller command must set the following properties for each type of view command. If the properties are not set by the command, they must be defined in the VIEWREG table.

- If using the ForwardView command, set `docname = ` *`view_file_name`* where *`view_file_name`* is the name of the display template. For example, `docname = productDisplay.jsp`.
- If using the DirectView command, do one of the following:
  - Set `textDocument = ` *`xxx`* where *`xxx`* is a java.io.InputStream object that contains the document in text form
  - Set `rawDocument = ` *`yyy`* where *`yyy`* is a java.io.InputStream object that contains the document in binary form

  When using the DirectView command, it is optional to set `contentType = ` *`ttt`* where *`ttt`*is the document content type
- If using the RedirectView command, set `url = ` *`uuu`* where *`uuu`* is the redirect URL.

Chapter 3. Persistent object model

WebSphere Commerce deals with a large amount of persistent data. There are more than 520 tables defined in the current database schema. Even with this extensive schema, you may need to extend or customize the database schema for your particular business needs.

WebSphere Commerce uses entity beans that are based on the Enterprise JavaBeans (EJBs) component architecture as the persistent object layer. These entity beans represent WebSphere Commerce data in a manner that models concepts and objects in the commerce domain. This persistence layer provides an extensible framework.

VisualAge for Java, Enterprise Edition provides sophisticated EJB tooling and a unit test environment that supports development for this framework.

Implementation of WebSphere Commerce entity beans

WebSphere Commerce entity beans - overview

As mentioned previously, the persistence layer within the WebSphere Commerce architecture is implemented according to the EJB component architecture. The EJB architecture defines two types of enterprise beans: entity beans and session beans. Entity beans are further divided into container-managed persistence (CMP) beans and bean-managed persistence (BMP) beans.

Most of the WebSphere Commerce entity beans are CMP entity beans. A small number of stateless session beans are used to handle intensive database operations, such as performing a sum of all the rows in a particular column. One advantage of using CMP entity beans is that developers can utilize the EJB tools provided in VisualAge for Java, Enterprise Edition. These tools allow developers to define Java objects and their database table mappings. The tools automatically generate the required persisters for the entity beans. Persisters are Java objects that persist Java fields to the database and populate Java fields with data from the database.

VisualAge for Java provides two extensions to the current EJB specifications: EJB inheritance and association. EJB inheritance allows an enterprise bean to inherit properties, methods, and method-level control descriptor attributes from another enterprise bean that resides in the same group. An association is a relationship that exists between two CMP entity beans.

Some of the WebSphere Commerce entity beans exploit the EJB inheritance feature. The WebSphere Commerce entity beans do not use the associations feature provided by VisualAge for Java. When developing your own entity beans, it is recommended that you do not use VisualAge for Java's association feature. This recommendation is made in order to minimize complexity in the object model. Rather than using the association feature provided by VisualAge for Java, an object relationship between enterprise beans can be established by adding explicit getter methods in the enterprise beans.

WebSphere Commerce provides two sets of enterprise beans: private and public. Private enterprise beans are used by the WebSphere Commerce run-time environment and tools. You *must not* use or modify these beans.

Public enterprise beans, on the other hand, are used by commerce applications, and can be both used and extended. These public enterprise beans are organized into the following EJB groups:

- WCSActrlEJBGroup
- WCSApproval
- WCSAuction
- WCSCatalog
- WCSCommon
- WCSContract
- WCSCoupon
- WCSFulfillment
- WCSInventory
- WCSMessageExtensions
- WCSOrder
- WCSOrderManagement
- WCSOrderStatus
- WCSPayment
- WCSPVCDevices
- WCSTaxation
- WCSUserTraffic
- WCSUser
- WCSUTF

Some of the EJB groups listed above contain session beans. In order to simplify migration in the future, you should not modify a session bean class. If required, you can create a new session bean in a new EJB group. For more information on creating new session beans, refer to "Writing new session beans" on page 71.

Deployment descriptors for WebSphere Commerce enterprise beans

A deployment descriptor is a special class that is serialized and that contains run-time settings for an enterprise bean. The deployment descriptors for WebSphere Commerce enterprise beans are set in a particular way, and should not be modified.

When creating new enterprise beans (entity or session beans), set the deployment descriptors within the EJB Development Environment (of VisualAge for Java) by right-clicking the bean and selecting **Properties**. Deployment descriptors for new enterprise beans should follow the same convention as those for the WebSphere Commerce enterprise beans. In particular, ensure you set the attributes as follows:

Attribute	**Value**
Transaction Attribute	`TX_REQUIRED`
Isolation Level	`TRANSACTION_READ_COMMITTED`
Run-As Mode	`SYSTEM_IDENTITY` or `CLIENT_IDENTITY`
Reentrant	Ensure this is not selected.

An enterprise bean often contains methods that only read information from the database, but never perform database updates. These methods are known as *read-only* methods. All read-only methods should be explicitly marked as such (right-click the method and select **EJB Method Attributes > Read-only Method**). If read-only methods are not marked in this manner, the EJB container unnecessarily attempts to update the database at the end of a transaction and causes a transaction rollback error in the read-only transaction. This causes performance problems.

Isolation levels

The transaction isolation level used for the WebSphere Commerce enterprise beans inside of VisualAge for Java is `TRANSACTION_READ_COMMITTED`. Note that there is a difference between the implementation of this isolation level for the JDBC driver for DB2 and the JDBC driver for Oracle. As such the transaction isolation level that is used when deploying to the WebSphere Application Server environment varies depending on which database is being used.

The isolation level used for DB2 databases when operating outside of the WebSphere Test Environment is `TRANSACTION_REPEATABLE_READ`. The isolation level used for Oracle databases when operating outside of the WebSphere Test Environment is `TRANSACTION_READ_COMMITTED`.

If you are deploying to a DB2 database, you do not need to manually change the transaction isolation level, it is changed during the step in deployment when you issue the `modifyIsolationLevel` command.

The following table presents a mapping of the DB2 and Oracle transaction isolation levels to their corresponding JDBC transaction isolation level.

JDBC	DB2	Oracle
Read Uncommitted	Uncommitted Read	Read Uncommitted
Read Committed	Cursor Stability	(Not applicable)
Repeatable Read	Read Stability	Read Committed
Serializable	Repeatable Read	Serializable
None	(Not applicable)	(Not applicable)

For the Cursor Stability transaction isolation level in DB2, only rows which have been updated during the given transaction will be locked exclusively. If no column of a given row is updated, even though it is part of the result set which got returned from a SQL statement, the row lock of that particular row will be released once the cursor has moved off into another row. In some situations, such as updating inventory, this may not be a desired behavior. Therefore, by changing the transaction isolation level to Read Stability in DB2, with a slight tradeoff in concurrency, data integrity can be greatly improved.

In the case of Oracle where only the Read Committed transaction isolation level, or JDBC Repeatable Read equivalent, is available, the actual implementation is done in a way which achieves the behavior most similar to the Repeatable Read transaction isolation level in DB2.

Extending the WebSphere Commerce object model

The WebSphere Commerce object model can be extended in the following ways:

- Extend the WebSphere Commerce's public enterprise beans
- Write a new entity bean
- Write a new stateless session bean

Details about how to perform these extensions are contained in the following sections.

Object model extension methodologies

Application requirements may lead to you extend the existing WebSphere Commerce object model. One example of such a requirement is adding additional attributes to your application. This can be accomplished by one of the following ways:

Without modifying an existing WebSphere Commerce public entity bean
Create a new database table, then create a new entity bean for that table. Add fields and methods to the entity bean to manipulate the new attribute, as required. Generate deployed code and an access

bean for the new entity bean. When the application requires the new attribute, it instantiates an access bean object and uses its methods to retrieve, set or manipulate the attribute.

By modifying an existing WebSphere Commerce public entity bean
Create a new database table, and create a table join between the new table and the existing table that corresponds to the existing enterprise bean that you are modifying. Create new fields in the existing WebSphere Commerce public entity bean and map the fields to their corresponding columns in the new table, using a secondary table map. Add any methods required. Regenerate the deployed code and access bean for the existing entity bean. The new attributes are available when the application instantiates the access bean object.

There are trade-offs between these two approaches. In general, the trade-offs relate to performance and effort for code maintenance.

Extension example: Consider an example in which your application requires you to capture the type of home that a customer has. You create a table called USERRES that contains the customer's ID and type of residence, where residence type (resType) may be a freehold home, a condominium or an apartment. This type of information is demographic information, and as such, is related to the existing Commerce Suite USERDEMO table. In examining the WebSphere Commerce code repository, you find that the WCSUser EJB group contains a ″Demographics″ enterprise bean. This bean has the getters and setters for demographic information stored in the USERDEMO table.

To perform your customization, there are two options. You can either create a new entity bean that interacts with the USERRES table, or you can add a new field (plus appropriate getter and setter methods) to the Demographics bean.

Using the first approach (creating entirely new code), you create a new Userres entity bean and map its fields to the columns of the USERRES table. When the application requires the customer's residence type, it must instantiate a Userres access bean object and retrieve the data. If the application requires other demographic information at the same time, it must also instantiate a Demographics access bean object and retrieve any other required attributes. Any parts of application logic that attempt to retrieve a complete set of demographic information for a customer must be modified to instantiate the new access bean as well as the original one. The following diagram displays this approach to extending the object model:

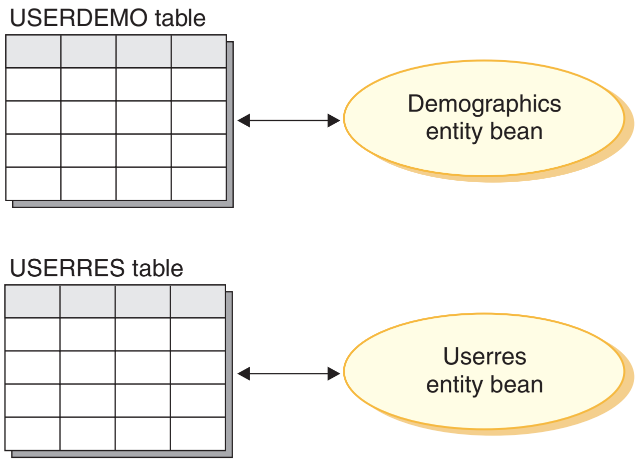


Figure 13.

From a display template perspective, a data bean must be able to access the new attribute, so that the information is available to JSP templates. In order to present a unified view to the Web developer creating the JSP templates, you should create a new data bean that extends the access bean for the original, existing entity bean. The data bean should also use delegation to populate the attributes from the new access bean. The following diagram displays this data bean implementation scenario:

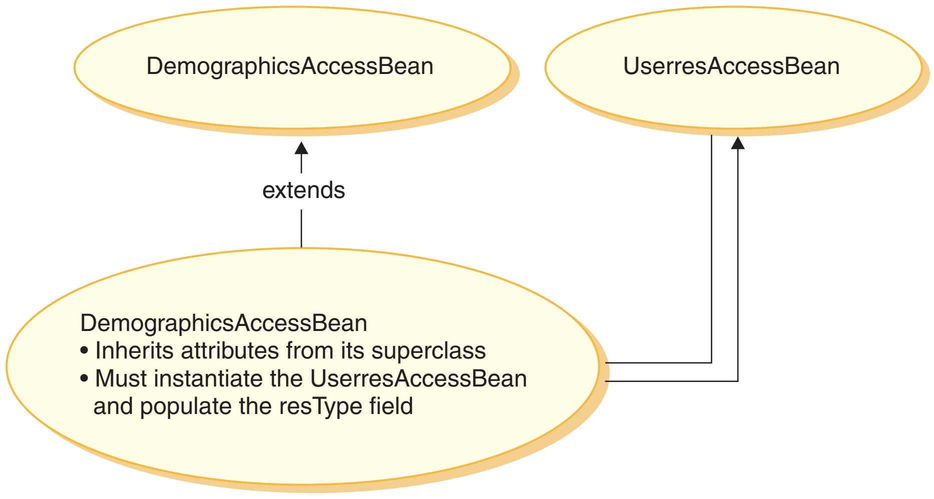

DemographicsAccessBean
UserresAccessBean
extends
DemographicsAccessBean
• Inherits attributes from its superclass
• Must instantiate the UserresAccessBean and populate the resType field

Figure 14.

Using the second approach (modifying existing code), you add a new field to the Demographics entity bean and create a secondary table map between the new field and the appropriate column in the USERRES table. When the application requires the customer's residence type, it instantiates a Demographics access bean object and retrieves the residence type. If the application requires any other demographic information about the customer, it is available in the same call to the bean. The following diagram displays this approach to for enterprise bean modification:

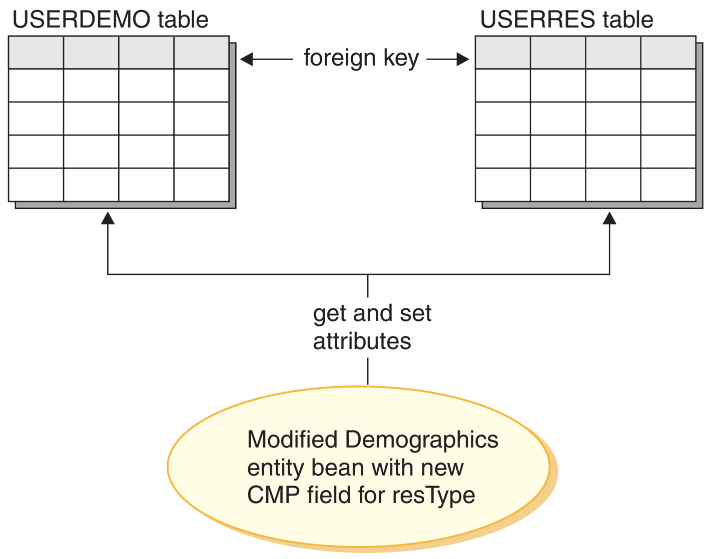

USERDEMO table
foreign key
USERRES table
get and set attributes
Modified Demographics entity bean with new CMP field for resType

Figure 15.

From a display template perspective, the new attribute (resType) is automatically available in the data bean, as soon as the DemographicsAccessBean is regenerated.

Note that when you are extending the object model, you must *not* add new columns to existing WebSphere Commerce database tables. You must create a new table for the new attribute. If you do attempt to add new columns to existing tables, the new attribute will be lost when you migrate to future releases of WebSphere Commerce.

Performance and code maintenance implications: The second approach has better run-time performance. This is a result of the fact that getting or setting the new attribute only requires the instantiation of a single entity bean and a single fetch is used to retrieve all required attributes.

Due to the fact that the second approach modifies existing WebSphere Commerce code, a migration issue arises when a new WebSphere Commerce code repository is released. You must merge your customized code with the new code, but when importing the new repository of code, the mapping information between the fields you added to the enterprise bean and the new table is not preserved. As a result, when migrating to a new release of the WebSphere Commerce code repository, the following steps must be performed:

1. Version your customized EJB code.
2. Import the new version of WebSphere Commerce code.
3. Using the tools in VisualAge for Java, compare the customized version of code to the new release of WebSphere Commerce code. Merge your customized code back into your workspace.
4. Manually remap any attributes you added to WebSphere Commerce public enterprise beans to the appropriate columns in your database.
5. Regenerate deployed code and access beans for the enterprise beans you modified in step 4.

In order to make this migration simpler, it is important to fully document your object model extensions at development time.

You may select to use a mix of the two approaches when making many extensions to the object model. You can use the first approach for areas of the system that are less susceptible to a degradation in performance and use the second approach were performance is an issue. In this manner, you can minimize effort for future migration, while still maintaining good system performance levels.

Recommended use of session beans

One of the strengths of WebSphere Commerce stems from its ability to take advantage of container-managed persistence (CMP) entity beans. CMP entity beans are distributed, persistent, transactional, server-side Java components that can be generated by the tooling provided within VisualAge for Java. In many cases, CMP entity beans are an extremely good choice for object persistence and they can be made to work at least as efficiently or even more efficiently than other object-to-relational mapping options. For these reasons, WebSphere Commerce has implemented core commerce objects using CMP entity beans.

There are, however, some situations in which it is recommended to use a session bean JDBC helper. These situations include the following:

- A case where a query returns a large result set. This is referred to as the *large result set* case.
- A case where a query retrieves data from several tables. This is referred to as the *aggregrate entity case.*
- A case where a SQL statement performs a database intensive operation. This is referred to as the *arbitrary SQL* case.

More details are provided in the following sections.

Note that if the session bean is being used as a JDBC wrapper to retrieve information from the database, it becomes more difficult to implement resource-level access control. When a session bean is used in this manner, the developer of the session bean must add the appropriate “`where`” clauses to the “`select`” statement in order to prevent unauthorized users from accessing resources.

Large result set case: There are cases where a query returns a large result set and the data retrieved are mainly for read or display purpose. In this case, it is better use a stateless session bean and within that session bean, create a finder method that performs the same functions as a finder method in an entity bean. That is, the finder method in the stateless session bean should do the following:
- Perform a SQL select statement
- For each row that is fetched, instantiate an access bean
- For each column retrieved, set the corresponding attributes in the access bean

When the access bean is returned, the command is unaware of whether the access bean was returned by a finder method in a session bean or from a finder method in an entity bean. As a result, using a finder method in a session bean does not cause any change to the programming model. Only the calling command is aware of whether it is invoking a finder method in a session bean or in an entity bean. It is transparent to all other parts of the programming model.

Aggregate entity case: In this case, one view combines parts of several objects and a single display page is populated with pieces of information that come from several database tables. For example, consider the concept of “My Account”. This may consist of information from table of customer information (for example, the customer name, age and customer ID) and information from an address table (for example, an address made up of a street and city).

It is possible to construct a simple SQL statement to retrieve all of the information from the various tables by performing a SQL join. This can be referred to as performing a “deep fetch”. The following is an example of a SQL select statement for the “My Account” example, where the CUSTOMER table is T1 and the ADDRESS table is T2:

```
select T1.NAME, T1.AGE, T2.STREET, T2.CITY
  from CUSTOMER T1, ADDRESS T2
  where (T1.ID=? and T1.ID=T2.ID)
```

The EJB tooling in VisualAge for Java does not support this notion of a deep fetch. Instead, it does a lazy fetch that results in a SQL select for each associated object. This is not the preferred method for retrieving this type of information.

In order to perform a deep fetch, it is recommended that you use a session bean. In that session bean, create a finder method to retrieve the required information. The finder method should do the following:

- Perform a SQL select statement for the deep fetch
- Instantiate an access bean for each row in the main table as well as for each associated object.
- For each column fetched and for each associated object fetched, set the corresponding attribute in the access bean.

Note that an access bean does not cache a getter method that throws an exception. In this case, you should create a simple wrapper class for the access bean using the following pattern:

```
public class CustomerAccessBeanCopy extends CustomerAccessBean {
   private AddressAccessBean address=null;

   /* The following method overrides the getAddress method in
      the CustomerAccessBean.
   */
   public AddressAccessBean getAddress() {
      if (address == null)
         address = super.getAddress();
      return address;
      }

    /* The following method sets the address to the copy. */

    public void _setAddress(AddressAccessBean aBean) {
       address = aBean;
      }
}
```

Continuing the CUSTOMER and ADDRESS example, the session bean finder method would instantiate a CustomerAccessBean for each row in the CUSTOMER table and an AddressAccessBean for each corresponding row in the ADDRESS table. Then, for each column in the ADDRESS table, it sets the attributes in the AddressAccessBean (street and city). For each column in the ADDRESS table, it sets the attributes in the CustomerAccessBean (name, age and address). This is shown in the following diagram.

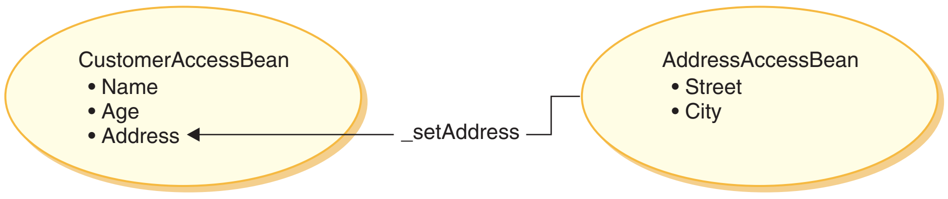


Figure 16.

Arbitrary SQL case: In this case, there is a set of arbitrary SQL statements that perform database intensive operations. For example, the operation to sum all the rows in a table would be considered a database intensive operation. It is possible that not all of the selected rows correspond to an entity bean in the persistent model.

An example that could result in the creation of an arbitrary SQL statement is a when a customer tries to browse through a very large set of data. For example, if the customer wanted to examine all of the fasteners in an online hardware store, or all of the dresses in an online clothing store. This creates a very large result set, but out of this result set, it is most likely that only a few fields from each row are required. That is, the customer may only initially be presented with a summary showing the item name, picture and price.

In this case, create a session bean helper method. This session bean helper method either performs a read or a write operation. When performing a read operation, it returns a read-only value object that is used for display purposes.

With proper data modelling, the number of cases of arbitrary SQL statements can usually be minimized.

Extending public entity beans

This section describes the design pattern of the WebSphere Commerce public entity beans. This design pattern enables you to make extensions such as adding new persistent fields, new business methods, or new finder methods.

The following diagram shows the implementation classes of the Catalog entity bean.

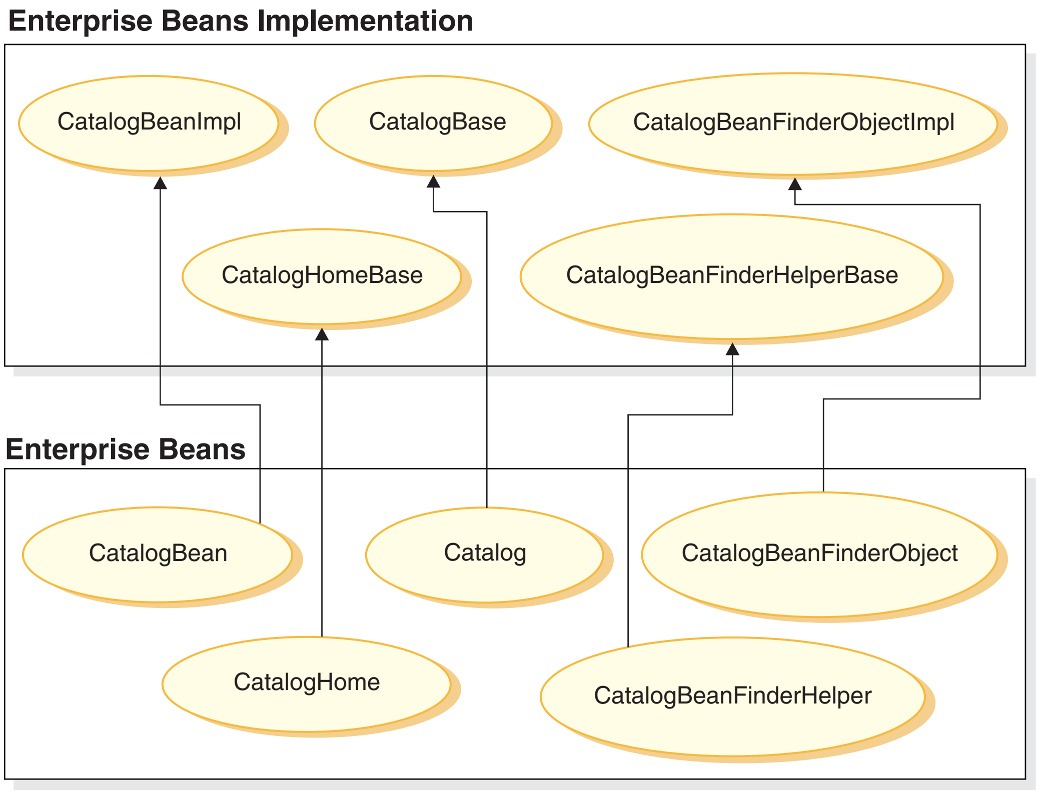


Figure 17.

The preceding diagram also applies to other entity beans because they are structured in a similar fashion and follow the same naming convention. To apply the diagram to another entity bean, substitute the entity bean's name for "Catalog". For example, the InterestItemBean class extends the InterestItemBeanImpl class and the InterestItem interface extends the InterestItemBase interface.

The diagram shows that the implementation class or interface for the public enterprise beans has been separated into two parts, using Java inheritance. The superclass or interface contains the WebSphere Commerce implementation code. All of these superclasses and interfaces are defined in separate packages and projects from the child classes and interfaces.

With the exception of the `*bean_name*BeanFinderHelperBase` and the `*bean_name*BeanFinderObjectImpl` classes, the WebSphere Commerce code repository contains binary code for all of these superclasses and interfaces. The source code for the `*bean_name*BeanFinderHelperBase` and the *bean_name*`BeanFinderObjectImpl` classes is included, so that you can see how the SQL statement for each finder is defined. You should not modify these classes, in any way.

To examine the source code for `CatalogBeanFinderHelperBase` and `CatalogBeanFinderObjectImpl`, open the `com.ibm.commerce.catalog.objsrc` package. Other packages use a similar naming convention.

Modifications can be made to the child classes and interfaces.

If you add new finder methods to the public enterprise beans, you must follow a particular naming convention for the methods. Name the new methods `findX`*a_description* where *a_description* is a description of your choice. Some examples of names are `findXByOwnerId` and `findXByOrderStatus`. Using this naming convention avoids the risk of name collision (duplicate names) with WebSphere Commerce finder methods.

One way to modify an existing WebSphere Commerce public entity bean is to add additional fields. In this case, after adding the new fields, you must examine each finder method in the bean. If the `where` clause portion of the finder methods contain any database aliases (for example, T1. or T2.), the aliases must be removed.

Creating a new CMP enterprise bean

When you have a new attribute that needs to be added to the WebSphere Commerce object model, you can create a new database table with a column for the required attribute. You must then also include this attribute in an enterprise bean, so that WebSphere Commerce commands can access the information.

One way to integrate the new attribute into the WebSphere Commerce object model is to create a new CMP enterprise bean. In this bean, you would then create a field that corresponds to the attribute in the new database table.

To create a new CMP enterprise bean, you must perform the following steps in VisualAge for Java:

1. Ensure that your workspace owner is set to WCS Developer.
2. Create a new EJB group for the bean.
3. Create the new CMP enterprise bean, using the Create Enterprise Bean SmartGuide tool.
4. For each column in the corresponding database table, add a new CMP field to the bean.
5. If required, create a pair of getter and setter methods for each of the CMP fields created.

6. If required, define the FinderHelper fields in the FinderHelper interface and add the new FinderHelper method.
7. Create a new ejbCreate method, if required, and promote the ejbCreate method to the home interface of the enterprise bean. This step is required if the new enterprise bean must create new entries in its corresponding database table.
8. Map the fields in the enterprise bean to the columns in the database table.
9. Generate the access bean and deployed code for the enterprise bean.

More detail on each of these steps is contained in the following sections.

Note: If your new enterprise bean is to be protected by the WebSphere Commerce access control system, refer to Chapter 4, "Access control" on page 85 for further details. Access control can be added after you create your bean.

Suppose that you have a new table called USERRES that specifies some information about the type of residence a user has. This table contains three columns: a USERID column, a HOME column that specifies the type of home and a ROOMS column that specifies the number of bedrooms in the residence.

Creating a new EJB group: When creating new entity beans, you must create them in an EJB group that is separate from the WebSphere Commerce EJB groups. When working with enterprise beans in VisualAge for Java, you must switch to the EJB tab in the workspace. Then, from the **EJB** menu, you can select **Add > EJB Group** to launch the Add EJB Group SmartGuide.

There are two important items that you must specify when creating the EJB group: the project in which EJB code is stored and the name of the EJB group.

The EJB project is viewable from the Projects tab in the workspace. When creating a new EJB group, you must specify a project that is separate from the WebSphere Commerce projects. For example, you might select to have your EJB group use the MyCustomEJB project. This project need not exist before you create the group, since VisualAge for Java can automatically create it for you. This project should only be used for EJB code; it should not be used for any command or data bean code. This separation of types of code is required for deployment purposes. By separating your own customized code from the WebSphere Commerce code, you will minimize the impact of migrating to future releases.

For the names of both the project and the EJB group, ensure that you follow any appropriate naming conventions for your application.

Creating the new CMP enterprise bean: To create your new CMP enterprise bean, you can use the Create Enterprise Bean SmartGuide tool. To launch the tool, right-click on the EJB group to which you will add the new bean and select **Add > Enterprise Bean**.

Select to create a new enterprise bean and specify a name for the bean. The WebSphere Commerce naming convention for enterprise beans is name the bean using the same name as the table to which the bean corresponds. For example, if your new database table is named USERRES, the enterprise bean should be named UserRes.

The Project field is automatically populated with the name of your project. You must specify a package name within the project in which code should be stored. An example of code to be stored in the package is the code for the access bean that you create for the enterprise bean. Again, when naming the package, ensure that you follow any appropriate naming conventions for your application. An example package name is `com.mycompany.mycustombeans`.

In the bean class, VisualAge for Java creates the private field called `EntityContext`. WebSphere Commerce provides its own entity context field in the ECEntityBean and your new entity bean should use that field, rather than the generated field in your own class. As such, you should remove the generated `EntityContext` from your new entity bean.

Your new enterprise bean must contain a `serialVersionUID` field. If you use the SmartGuide to create your new bean, VisualAge for Java generates this field for you. If you do not use the tool to create your bean, you must add this field.

In the Superclass field, you must specify the `com.ibm.commerce.base.objects.ECEntityBean` class. The following example of code demonstrates the functions provided by the superclass:

```
public class myEJB extends com.ibm.commerce.base.objects.ECEntityBean {
   public void ejbLoad()throws java.rmi.RemoteException {
      super.ejbLoad();--the super method will add EJB trace
       --your logic --
   }
   public void ejbStore()throws java.rmi.RemoteException {
      super.ejbStore();--the super method will add EJB trace
      --your logic --
   }
}
```

You should also remove any ejbCreate() and ejbPostCreate() methods that take no arguments. Leaving these methods in the entity bean can cause run-time errors.

For each column of your database table, you must create a new CMP field in the bean (you must click Next in the SmartGuide to see the Add CMP fields to the bean option). For each field, specify a field name and the data type for the field. For any column that either the primary key or part of the primary key, enable the Key Field checkbox. For all other columns, enable the Promote getter and setter methods to the remote interface checkbox.

Once all fields have been filled in, click Finish and VisualAge for Java creates the new CMP enterprise bean.

Creating new FinderHelper fields in the *`Bean_Name`*`FinderHelper` interface: The *`Bean_Name`*`FinderHelper` interface contains SQL search clauses that correspond to all FinderHelper methods other than the findByPrimaryKey method.

A new enterprise bean uses `findXBy`*`ArgName`* (where *`ArgName`* is the name of an argument) methods defined in the bean's FinderHelper interface along with FinderHelper methods to compose SQL queries. Use the "findXBy" naming convention for your field name to ensure that your field names are always unique from WebSphere Commerce field names.

To create the new FinderHelper fields in your bean, you must select the *`Bean_Name`*`_FinderHelper` interface and modify the source code to establish how select statements should be formed. An example follows:

```
public interface UserResFinderHelper {
 public static final String findXByHomeAndRoomsWhereClause = "(T1.HOME = ?
      and T1.ROOMS = ?)";
}
```

The home interface would then require a method called `findXByHomeAndRooms` that takes input parameters for each of HOME and ROOMS that populate the values represented by the ? characters. This type of query construction is referred to as a *parameter insert*.

If the input parameters were "detached" and "3", the generated SQL statement would be

```
select * from USERRES where HOME=detached and ROOMS=3
```

For security reasons, when creating FinderHelper methods for a new entity bean, you should use parameter inserts. The reason for this recommendation is that it protects the query from being altered by users. An alternative approach would be to use a construct similar to the following:

```
public static final String
     findXByOwnerIdWhereClause = " (T1.OWNERID = input_string) ";
```

where `input_string` is a string value passed in from a URL. This is not desirable, since a malicious user could enter a value such as "'123' OR 1=1" which changes the SQL statement. If a user can change the SQL statement, they may be able to make unauthorized access to data. Therefore, the recommended approach is to use parameter inserts.

If you cannot use a parameter insert and therefore, have to use an input string to compose the SQL statement, you must enforce parameter checking on the input string to ensure that the input parameter is not a malicious attempt to access data.

Creating new FinderHelper methods in the *Bean_Name*`Home` interface: For each FinderHelper field that you have specified in the *Bean_Name*`FinderHelper` interface, you must create a FinderHelper method in the home interface of the bean. The name of the FinderHelper methods must match the FinderHelper field name exactly, except "WhereClause" is dropped. That is, with the example field name of `findXByHomeAndRoomsWhereClause`, the corresponding method name is `findXByHomeAndRooms`.

To create the new FinderHelper methods, do the following:
1. Right-click the *Bean_Name*__Home__ interface, and select **Add > Method**. The Create Method SmartGuide opens.
2. Select **Create a new method** and click **Next**.
3. In the **Method Name** field, enter a name for the FinderHelper method. This name of the FinderHelper methods must match the FinderHelper field name exactly, except the "WhereClause" part is dropped. For example, enter `findXByHomeAndRooms`.
4. In the **Return type** field, enter one of the following:
   - If the FinderHelper method uses the primary key to query the database and the method should return a unique record, specify the EJB object as the return type. For example, enter `UserRes`.
   - If the FinderHelper method returns a result set instead of a unique record, specify the return type as `java.util.Enumeration`.
5. Click the **Add** button beside **What parameters should this method have?** to add the appropriate parameters. For example, you might add `argHome` of type `String` to hold the residence type and `argRooms` of type `byte` to hold the number of rooms.
6. When you have completed adding all parameters, click **Next**.
7. Click the **Add** button beside **What exceptions may this method throw?** and add the following exceptions:
   - `java.rmi.RemoteException`
   - `javax.ejb.FinderException`

Note: The preceding list of exceptions shows the minimum set of exceptions that your method should throw. Depending upon your own code, you may need to specify other exceptions.

8. Click **Finish**.

Creating a new ejbCreate method: When the enterprise bean is created, the ejbCreate method is automatically generated. This method is then promoted to the remote interface, so that it is available in the access bean. The default ejbCreate method only contains parameters that are either the primary key, or part of the primary key. This means, only those values get instantiated upon instantiation.

If your enterprise bean contains fields that are not part of the primary key and are non-nullable fields, you must create a new ejbCreate method in which you specifically instantiate those fields. By doing so, each time a new record is created, all non-nullable fields will be populated with the appropriate data.

To create a new ejbCreate method, do the following:

1. In the Types pane, expand the ***Bean_Name*Bean** class. For example, select **UserResBean**.
2. Click the existing ejbCreate method to view the source code. (Note, this may be ejbCreate(String), ejbCreate(String, int) or it may take some other input parameters, depending upon the primary key of your enterprise bean.)
3. You must modify the source code so that each CMP field is included as an input parameter to the method, and so each CMP field is instantiated with the appropriate value. In the UserRes example where the UserId is the primary key, the source code initially appears as:

   ```
   public void ejbCreate(int argUserId)
      throws javax.ejb.CreateException, java.rmi.RemoteExeption {
      _initLinks();
      userId = argUserId;
   }
   ```

   But, you may want to ensure that both the number of rooms and type of home are initialized. In this case, you would change the code to the following:

   ```
   public void ejbCreate(int argUserId, String argHome, byte Rooms)
      throws javax.ejb.CreateException, java.rmi.RemoteExeption {
      _initLinks();
      // All CMP fields should be initialized here
      userId = argUserId;
      home = argHome;
      rooms = argRooms;
   }
   ```

Note: If you want to use a system generated primary key, refer to "Primary keys" on page 75 for details.

4. Save the modified method. When you save code, VisualAge for Java creates a new ejbCreate method that takes the new parameters. The original ejbCreate method remains.
5. Delete the original ejbCreate method (the one with no arguments).
6. Right-click the new ejbCreate method and select **Add > EJB Home Interface**.
7. Create a corresponding ejbPostCreate method. (This method does not need to be added to the home interface.)

Mapping the database table to the new enterprise bean: Once you have created the new enterprise bean, you must create a mapping between the CMP fields in the bean and the columns in the database table. This mapping is called a schema. VisualAge for Java provides tooling to simplify this task.

To create the schema, do the following:

1. From the **EJB** menu, select **Open To > Database Schemas**. The Schema Browser opens.
2. From the **Schemas** menu, select **Import / Export Schema > Import Schema from Database**.
3. Enter a name for the new schema and click **OK**.
4. In the Database connection window, enter the information as follows:

Attribute	DB2 value	Oracle value
Connection Type	`COM.ibm.db2.jdbc.app.DB2Driver`	`Oracle.jdbc.driver.OracleDriver`
Data Source	`jdbc:db2:wc_database_name`	`jdbc:oracle:thin@`*`hostname`*`:`*`port`*`:`*`Oracle_SID`*
User Name	*`wc_db_user_name`*	*`wc_db_user_name`*
Password	*`wc_db_password`*	*`wc_db_password`*

with values replaced as follows:

- DB2 *`wc_database_name`* is the name of your WebSphere Commerce database
- Oracle *`hostname`* is the Oracle server host name.
- Oracle *`port`* is the port number of the Oracle database.
- Oracle *`Oracle_SID`* is the Oracle instance ID.
- *`wc_db_user_name`* is the user name for the database.
- *`wc_db_password`* is the database password.

5. From the **Qualifiers** list, select your database qualifier (this may be your database user name).
6. Click **Build Table List**.
7. Select *your_new_table* (from the generated list and click OK to generate the schema.
8. After the schema is generated, click *your_new_table* in the Schema pane.
9. Highlight, then double-click *your_new_table* in the Table pane.
   The Table Editor window opens.
10. Remove any information in the **Qualifier** field. This makes your enterprise bean portable to other machines.
11. From the **Schemas** menu, select **Save Schema**. Enter the appropriate project, package and class names.

Next you must create the schema map. The schema map is a mapping between database columns and fields in the enterprise bean.

To create the schema map, do the following:
1. From the **EJB** menu, select **Open To > Schema Maps**.
   The Datastore Map window opens.
2. Enter the name of the map. Refer to "Database schema object naming considerations" on page 78 for recommendations about naming the new map.
3. Select your EJB group and schema. Refer to "Database schema object naming considerations" on page 78 for recommendations about naming the new schema.
4. In the Datastore Maps pane, select your map.
5. In the Persistent classes pane, select your class.
6. From the **Table Maps** menu, select **New Table Map > Add Table Map with No Inheritance**.
7. From the **Table** drop-down list, select your table and click **OK**.
8. In the Table Maps panel, highlight then right-click your table map and select **Edit Property Maps**.
   The Property Map Editor opens.
9. For each of the class attributes (the CMP fields in the bean) you must specify the map type and the database column to which it should be mapped. For example, you would map the UserId class attribute using a map type of "Simple" to the USERID column in the database table.
10. After all fields have been mapped to their corresponding database column you must save the schema map. From the **Database Map** menu, select **Save Datastore Map**. Enter the appropriate project, package and class names for saving the map. Click **Finish**.

Creating the access bean and generating deployed code: An access bean acts as a wrapper for the enterprise bean that simplifies how other components interact with the enterprise bean. You must create an access bean for your new enterprise bean.

When generating deployed code, the tools in VisualAge for Java analyze the bean to ensure that the rules specified in the Sun Microsystems EJB specifications, as well as the rules specific to the EJB server, are met.

To create an access bean for your new enterprise bean, do the following:

1. In the Enterprise Beans pane, right-click your new enterprise bean and select **Add > Access Bean**. (You may have to expand the EJB group that contains your new bean first, in order to view the bean.)
   The Create Access Bean SmartGuide opens.
2. In the **EJB Group**, **Enterprise bean** and **Access bean name** fields, specify the appropriate EJB group, bean name, and access bean name.
3. Select **Copy Helper for an Entity Bean for the Access Bean Type** and click **Next**.
4. From the **Select home method for zero argument constructor** drop-down list, select **findByPrimaryKey**.
5. From the **Converter** drop-down list, select **WCSStringConverter** for the initial properties and click **Next**.
6. In the Select and Customize Bean Properties for Copy Helper window, select the **WCSStringConverter** for each field.
7. Click **Finish**.

To generate the deployed code, do the following:

1. In the Enterprise Beans pane, right-click your new enterprise bean and select **Generate Deployed Code**.

Note that the deployed code that is generated using this tool complies with the EJB 1.0 specification and is only used when running your enterprise bean within VisualAge for Java. At a later stage when you deploy your enterprise bean to a WebSphere Commerce application running within WebSphere Application Server V4.0, you are required to generate a JAR file containing deployed code that complies with the EJB 1.1 specifications. For more information about creating this EJB 1.1 Export JAR file, refer to “Information about EJB deployed code” on page 188 and “Generating deployed code” on page 346.

Using the test client to test the enterprise bean: VisualAge for Java provides a test client that can be used to test enterprise beans. To use the test client to test your new bean, do the following:

1. Start the EJB server that contains the enterprise bean you are testing.

2. Right-click the enterprise bean and select **Run Test Client**.
   The EJB Test Client and EJB Lookup windows open.
3. In the **JNDI Name** field, enter the JNDI name of the enterprise bean and click **Lookup**.
4. Right-click the **findByPrimaryKey** method that has the arguments filled in and select **Invoke**.

Coding Practices: The following enterprise bean coding practices should be observed:

- Do not use either the BLOB or CLOB datatype.
- No CMP field of the LONG datatype (also known as LONG VARCHAR) should be either the first or last member in the CMP field list for the enterprise bean. To verify the list, check the `EJSJDBCPersister._hydrate()` method and see if either the first or last element in the list if of type LONG VARCHAR. If the first field is of this type, do the following:
  1. Unset the first field. Call this `fieldA`.
  2. Unset another field that is *not* of the type LONG VARCHAR. Call this `fieldB`.
  3. Reset `fieldA`.
  4. Reset `fieldB`.
  5. Open the Schema Map Browser and edit the table map to reflect these changes. Save the map.
  6. Regenerate deployed code.
- Enterprise bean code should not reference anything outside of the enterprise bean packages. For example, you should not reference commands or data beans in the enterprise bean code
- To enable access control for your enterprise bean, add the com.ibm.commerce.security.Protectable interface and/or the com.ibm.commerce.security.Groupable interface to the enterprise bean's remote interface. After adding these interfaces, regenerate the bean's deployed code and access bean. You must also create an access helper class object in the objsrc package.

Creating a simple data bean

A data bean is a bean that is used in JSP templates to retrieve information from the enterprise bean. A simple data bean extends its corresponding access bean and implements the SmartDataBean interface. Most code for the data bean is automatically generated by VisualAge for Java.

To create a simple data bean, you must perform the following steps:

1. Create a project and package to store the data bean code.
2. Create a data bean that extends the corresponding access bean and implements the appropriate data bean interface.

3. Create the set methods for the data bean.
4. Create the get methods for the data bean.

Creating the project and package for data bean code: Creating a project and package creates a place in which your data bean code can be stored.

To create a new project, do the following:
1. Select the **Projects** tab.
2. From the **Selected** menu, select **Add > Project**.
   The Add Project SmartGuide opens.
3. Ensure that **Create a new project named** is selected and enter a name for your new project. For example, enter `My Data Beans`.
4. Click **Finish**.

To create a new package, do the following:
1. Right-click the project that you created for your data bean code and select **Add > Package**. For example, right-click **My Data Beans** and select **Add > Package**.
   The Add Package SmartGuide opens.
2. Ensure that **Create a new package named** is selected and enter a suitable name for your data bean package. For example, enter `com.mycompany.mydatabeans`.
3. Click **Finish**.

Creating a data bean: A data bean is a Java bean that is used within a JSP template to provide dynamic content to the page. It normally provides a simple representation (indirectly) of an entity bean by extending an access bean. The data bean encapsulates properties that can be retrieved from or set within the entity bean.

To create a data bean, do the following:
1. Right-click the package into which you will store the data bean and select **Add > Class**.
   The Create Class SmartGuide opens.
2. The project and package name fields are already populated.
3. Ensure that **Create a new class** is selected and click **Next**.
4. In the **Class Name** field, enter a name for your new data bean. For example, to create a data bean that extends the UserResAccessBean, enter `UserResDataBean`.
5. To specify the superclass, click **Browse**, then in the pattern field, enter the name of the corresponding access bean. For example, enter `UserResAccessBean` and click **OK**.
6. Click **Next**.

7. To specify the interfaces that the data bean should implement, click **Add**. In the Interface window, do the following:
   a. In the **Pattern** field, enter `com.ibm.commerce.beans.SmartDataBean` then click **Add**.
   b. In the **Pattern** field, enter `com.ibm.commerce.beans.InputDataBean` then click **Add**.
   c. Click **Close**.
8. Click **Finish**.

Adding required fields to the data bean: This section describes how to add required fields to your new data bean.

To add the iCommandContext field, do the following:
1. Right-click the new data bean (for example, UserResDataBean) and select Add > Field.
   The Create Field SmartGuide opens.
2. In the **Field name** field, enter `iCommandContext`.
3. Click **Browse** to add the field type and enter `com.ibm.commerce.command.CommandContext`. Click **OK**.
4. For the access modifiers, select **Protected**.
5. Click **Finish**.

To add the iRequestProperties field, do the following:
1. Right-click the new data bean (for example, UserResDataBean) and select Add > Field.
   The Create Field SmartGuide opens.
2. In the **Field name** field, enter `iRequestProperties`.
3. Click **Browse** to add the field type and enter `com.ibm.commerce.datatype.TypedProperty`. Click **OK**.
4. For the access modifiers, select **Protected**.
5. Click **Finish**.

Modifying the data bean's set methods: After you have created your data bean, you must modify code in some of the generated set methods.

To update the set methods, do the following:
1. Expand your new data bean to view its fields and methods.
2. Select the **`setCommandContext(CommandContext)`** method to view its source code.
   The Source pane displays source code as follows:

```
public void setCommandContext(com.ibm.commerce.comand.CommandContext arg1)
 {
 }
```

3. Modify the source code so the method appears as follows:

   ```
   public void setCommandContext(com.ibm.commerce.comand.CommandContext arg1)
   {
      iCommandContext = arg1;
   }
   ```

   Save your work (Ctrl+S).
4. Select the **setRequestProperties(TypedProperty)** method to view its source code.
   The Source pane displays the source code as follows:

   ```
   public void setRequestProperties(
    com.ibm.commerce.datatype.TypedProperty arg1)
      throws Exception
      {
      }
   ```
5. You may want to modify the source code to populate the primary key of the corresponding access bean. The recommended way to do this is to use the data bean manager to indirectly set this value. This indirect method is designed to ensure that a primary key value taken from the URL properties will not override the primary key, if it has previously been set. To have your setRequestProperties method follow this model, code it in a fashion that is similar to the following code snippet. Note that in the following example, the primary key is the user id. This may be different, depending upon the situation (as such, the following code may not immediately compile in your application).

   ```
   public void setRequestProperties(
      com.ibm.commerce.datatype.TypedProperty arg1)
      throws Exception
      {
         iRequestProperties = arg1;
         try {
             if (// check for nulls
                 getDataBeanKeyUserId() == null)
                {
                   super.setInitKey_UserId(aUserId);
                }
         } catch (com.ibm.commerce.exception.ParameterNotFoundException e)
               {
               }
      }
   ```

There are two other ways in which the primary key for the access bean can be set. It can be done externally from the data bean, for example in the JSP template. In this case, before activating the data bean in the JSP template, explicitly call the data bean's set method for the primary key. For example, the JSP could include code similar to the following (where db is the data bean object):

```
db.setInitKey_UserId(/*input parameter*/)
db.activate();
```

Alternatively, the primary key can be set in a direct way. That is, the JSP template only contains the db.activate method and then the data bean manager explicitly sets the primary key in the access bean. For example, the code for the setRequestProperties method of the data bean would appear similar to the following:

```
public void setRequestProperties(
   com.ibm.commerce.datatype.TypedProperty arg1)
   throws Exception
      {
         iRequestProperties = arg1;
         try
             {
                super.setInitKey_UserId(aUserId);
             }
       } catch (com.ibm.commerce.exception.ParameterNotFoundException e)
           {
           }
      }
```

Note that the recommended procedure for setting the primary key is the indirect method, shown in step 5.

Modifying the data bean's get methods: After you have created your data bean, you must modify code in some of the generated get methods.

To update the get methods, do the following:

1. Expand your new data bean to view its fields and methods.
2. Select the **getCommandContext()** method to view its source code. The Source pane displays source code as follows:
   ```
   public com.ibm.commerce.comand.CommandContext getCommandContext ()
   {
      return null;
   }
   ```
3. Modify the source code so the method appears as follows:
   ```
   public com.ibm.commerce.comand.CommandContext getCommandContext ()
   {
      return iCommandContext;
   }
   ```

   Save your work (Ctrl+S).
4. Select the **getRequestProperties()** method to view its source code. The Source pane displays the source code as follows:

```
public com.ibm.commerce.datatype.TypedProperty setRequestProperties()
{
   return null;
}
```

5. Modify the source code so that it appears as follows:

   ```
   public com.ibm.commerce.datatype.TypedProperty setRequestProperties()
   {
      return iRequestProperties;
   }
   ```

   Save your work (Ctrl+S).

Modify the populate() method: You must modify the populate method, by doing the following:

1. Expand your new data bean to view its fields and methods.
2. Select the **`populate()`** method to view its source code.
   The Source pane displays source code as follows:

   ```
   public void populate () throws Exception {}
   ```
3. Modify the source code so the method appears as follows:

   ```
   public void populate () throws Exception {
      super.refreshCopyHelper();
   }
   ```

   Save your work (Ctrl+S).

Writing new session beans

When creating new session beans, you must create the session bean in an EJB group that is separate from the WebSphere Commerce EJB groups. Store this new EJB group in a new project that is separate from the WebSphere Commerce projects. For example, you might create the `MyCustomBeans` EJB group within the `com.mycompany.mycustomcode` package. By separating your own customized code from the WebSphere Commerce code, you will minimize the impact of migrating to future releases.

Your new session bean should extend the com.ibm.commerce.base.helpers.BaseJDBCHelper class. The superclass provides methods that allow you to obtain a JDBC connection object from the data source object used by the commerce server, so that the session bean participates in the same transaction as the other entity beans. The following is an example of code to demonstrate the functions provided by the superclass:

```
public class mySessionBean extends com.ibm.commerce.base.helpers.BaseJDBCHelper
   implements SessionBean {

   public Object myMethod () throws javax.naming.NamingException,
      java.rmi.RemoteException, SQLException {

       ////////////////////////////////////////////////
```

```
      //  -- your logic, such as initialization --    //
      ////////////////////////////////////////////////

         try {
            // get a connection from the WebSphere Commerce data source
            makeConnection();
            PreparedStatement stmt = getPreparedStatement(
               "your sql string");
            ////////////////////////////////////////////////////////////
            // -- your logic such as set parameter into the prepared  //
            // statement --                                           //
            ////////////////////////////////////////////////////////////
            ResultSet rs = executeQuery(stmt, false);

            ////////////////////////////////////////////////
            // -- your logic to process the result set -- //
           ////////////////////////////////////////////////

               }
          finally {
             // return the connection to the WebSphere Commerce data source
            closeConnection();
          }

        ////////////////////////////////////////////////////
       // -- your logic to return the result ---          //
       /////////////////////////////////////////////////////

   }

}
```

In the preceding code example, the `executeQuery` method takes two input parameters. The first is a prepared statement and the second is a boolean flag related to a cache flush operation. Set this flag to `true` if you need the container to flush all entity objects for the current transaction from the cache before executing the query. This would be required if you have performed updates on some entity objects and you need the query to search through these updated objects. If the flag were set to `false` those entity object updates would not be written to the database until the end of the transaction.

You should limit the use of this flush operation and generally set the flag to `false`, except in those cases where it is really required. The flush operation is a resource intensive operation.

Object life cycles

The enterprise beans in the object model include both *independent* and *dependent* objects. An independent object has its own life cycle, controlled directly by the create or remove requests of the business logic invoking the object. A dependent object has a life cycle that is attached to another object, known as the *owner object* (which may also in turn be a dependent object, but further up the association hierarchy, an independent object exists). When the owner object is deleted, all dependent objects are also deleted. The actual deletes are controlled by cascading delete specifications within the database.

For example, given a user object that returns an address book object and a list of order objects, if the user object is deleted, its address book object is also deleted (since the book is owned by the user), and so are all the address objects within the book (since the addresses are owned by the book). However, the order objects are not deleted because the owner of orders is a store object, not the user object.

A specific design pattern is used for the creation of dependent objects. The create method of a dependent object must supply a reference to its owner object; therefore, the owner object must exist before the dependent object can be created.

Transactions

The Enterprise JavaBeans V1.0 architecture specifies three alternative commit-time options with respect to the instance state. They are described as options A, B, and C in the specification document. For complete details on these options, refer to Sun Microystem's Enterprise JavaBean's V1.0 specification document.

Although the WebSphere Application Server implements options A and C, option A assumes that the database is not shared.

In option C, the enterprise bean container does not cache a "ready" instance between transactions. As soon as a transaction has completed, the instance is returned to the pool of available instances. WebSphere Commerce uses option C because the database is shared across multiple WebSphere Commerce applications. In this implementation, the container loads persistent data for entity beans at the start of each transaction and the entity beans are only cached for the duration of the transaction. The container activates multiple instances of an entity bean, one for each transaction in which the entity is being accessed. Transaction synchronization is performed by the database.

The transaction attribute of each enterprise bean is set to `TX_REQUIRED`. Since the Web controller starts a transaction before executing a command that accesses an enterprise bean (through its corresponding access bean), the business methods of the enterprise bean are invoked within the context of this transaction.

Other considerations for entity beans

Find for Update

A situation in which multiple applications can access the same row in a database for the purpose of updating the row, is known as a *concurrent update*.

There are situations in which concurrent updates may be allowed, and other situations where they are definitely not desired.

If the database update is an overwrite, where the new value has no relation to the current value in the database, concurrent updates may be allowed. If concurrent update is allowed and multiple applications attempt to update the same row in a database, the last attempt is the one that gets updated in the database.

If the database update depends upon the current value in the database, a concurrent update is not desired. For example, if an application is updating product inventory, only one application should be allowed to update the inventory at a time.

Factors that affect whether or not concurrent update is permitted include database locks and enterprise bean isolation levels.

In order to prevent a second application from concurrently updating a row, the first application accessing the row must fetch the row using the “find for update” option. When the “for update” option is used, a *write lock* (also known as an exclusive lock) is applied to the row. With this write lock applied to the row, any application that attempts to access the row using the “find for update” is blocked.

If your application permits concurrent updates, it can just fetch the data, without locking the row.

Consider the OrderProcess scenario in which UpdateInventory needs to find all the products included in an order and update the inventory accordingly. Since the same products may be included in many other orders, *find for update* should be used, and it should be used as early as possible within a transaction scope to reduce the possibility of deadlocks. Therefore, the UpdateInventory algorithm may be represented by the following pseudo code:

```
UpdateInventory
 find all the order items in the order
 for each order item
  fetch its inventory using "find for update"
  ...
```

In the long-running Business-to-Business scenario, where an order may have many items, find for update should be used as early as possible. The logic may become the following:

```
find for update the inventory of all the products in an order
for each product
 if (total quantity ordered for that product < inventory)
        deduct quantity from inventory
 else
        error
```

Flush remote method

Since WebSphere Application Server does not write changes made on the entity beans to the database until the transaction commit time, the database may get temporarily out of synchronization with the data cached in the entity bean's container.

A flush remote method is provided (in the `com.ibm.commerce.base.helpers.BaseJDBCHelper` class) that writes all the committed changes made in all transactions (that is, it takes information from the enterprise bean cache) and updates the database. This remote method can be called by a command. Use this method, only when absolutely required, since it is expensive in terms of overhead resources, and therefore, has a negative impact on performance.

Consider a logon command that has the following piece of code:

```
UserAccessBean uab = ...;
 uab.setRegisteredTimestamp(currentTimestamp);
 uab.commitCopyHelper();
```

Before the transaction has been committed, the REGISTEREDSTAMP in the USER table will not have been updated with the current time stamp. The update only occurs at transaction commit time. The flush method has to be used so that any direct JDBC query (in the same transaction), for example, *select from user where registeredstamp ...* returns the user with the specified registration time stamp.

Securing enterprise beans

If you are using the WebSphere Application Server for securing enterprise beans, you must assign the methods of any new enterprise beans to the `WCSMethodGroup` security method group, using the WebSphere Application Server Administrator's Console. Perform this step when deploying the new enterprise beans. Additionally, if you modify existing WebSphere Commerce entity beans, you must assign the methods of all of the entity beans in the affected EJB group to the `WCSMethodGroup` security method group. For a description of the deployment process for customized code, refer to "Code deployment" on page 187.

Primary keys

A primary key is a unique key that is part of the definition of a table. It can be used to distinguish one record from others. All records must have a primary key. When you create a new record in a table, you may need to generate a unique primary key for the record.

In the WebSphere Commerce programming model, the persistence layer includes entity beans that interact with the database. As such, database records may be created when an entity bean is instantiated. Therefore, the ejbCreate method for the instantiation of an entity bean may need to include logic to generate a primary key for new records.

When an application requires information from the database, it indirectly uses the entity beans by instantiating the bean's corresponding access bean and then getting or setting various fields. An access bean is instantiated for a particular record in a database (for example, for a particular user profile) and it uses the primary key to select the correct information from the database.

The following sections describe how to create a unique primary key and how to select by primary key.

Creating primary keys: The `ejbCreate` method is used to instantiate new instances of an entity bean. This method is automatically generated but the generated method only includes logic to initialize primary keys to a static value.

You may need to ensure that the primary key is a new, unique value. In this case, you may have an ejbCreate method similar to the following code snippet:

```
Public void ejbCreate(int argMyOtherValue)
   throws javax.ejb.CreateException,
   java.rmi.RemoteException {
      //Initialize CMP fields
      MyKeyValue = com.ibm.commerce.key.ECKeyManager.
         singleton().getNextKey("table_name");
      MyOtherValue = argMyOtherValue;
}
```

In the preceding code snippet, the getNextKey method generates a unique integer for the primary key. The *`table_name`* input parameter for the method must be an exact match to the TABLENAME value that is defined in the KEYS table. Be certain to match the characters and case exactly.

In addition to including the preceding code in your ejbCreate method, you must also create an entry in the KEYS table. The following is an example SQL statement to make the entry in the KEYS table:

```
insert into KEYS (TABLENAME, COUNTER, KEYS_ID)
   values ("table_name", 0, 1)
```

Note that with the preceding SQL statement default values for the other columns in the KEYS table are accepted. The value for COUNTER indicates the value at which the count should start. The value for KEYS_ID should be any positive value.

If your primary key is defined as a long datatype (BIGINT for DB2 or NUMBER for Oracle), use the `getNextKeyAsLong` method.

Selecting by primary key: Within an access bean, you must select the appropriate database record by using the primary key. The following code snip demonstrates how to perform this select. It also includes additional logic, that is explained later.

```
UserProfileAccessBean abUserProfile = new UserProfileAccessBean();
abUserProfile.setInitKey_UserId(getUserId().toString());
abUserProfile.refreshCopyHelper();
```

The first line in the preceding code snippet instantiates a new UserProfileAccessBean that is called "abUserProfile". The second line sets the primary key in the access bean. The `setInitKey_`*`xxx`* (where *`xxx`* is the primary key field name) naming convention is used by VisualAge for Java to name the set methods for primary keys. When instantiating an access bean, you should ensure that all fields set by a setInitKey_xxx method are initialized before using the refreshCopyHelper method. The order in which the setInitKey_xxx methods are called is not important.

After all setInitKey_xxx methods have been called, you have initialized all required fields and can use the refreshCopyHelper method to retrieve information from the database.

If you update values in the local cache of the access bean, you must also include a commitCopyHelper call to update the database with the updated information. For example, if after retrieving data using the refreshCopyHelper method you update a customer's name (by setting the name value) you must then call `abUserProfile.commitCopyHelper()` to update the database with the new information.

Using entity beans

A program that uses enterprise beans must deal with the Java Naming and Directory Interface (JNDI) as well as the home and remote interfaces of enterprise beans. To simplify the programming model, an access bean for each enterprise bean is generated. When creating your own enterprise beans, use the tooling in VisualAge for Java to generate this access bean.

WebSphere Commerce commands interact with access beans rather than directly with the entity beans. As the diagram illustrates, using the access bean provides the following advantages:

- A simpler programming interface. The access bean behaves like a Java bean and hides all the enterprise bean specific programming interfaces, like JNDI, home and remote interfaces from the clients.
- At run time the access bean caches the enterprise bean home object because look ups to the home object are expensive, in terms of time and resource usage.
- The access bean implements a copyHelper object that reduces the number of calls to the enterprise bean when commands get and set enterprise bean attributes. Therefore, only a single call to the enterprise bean is required, when reading or writing multiple enterprise bean attributes.

The following diagram displays the interaction between commands, access beans, entity beans and the database.

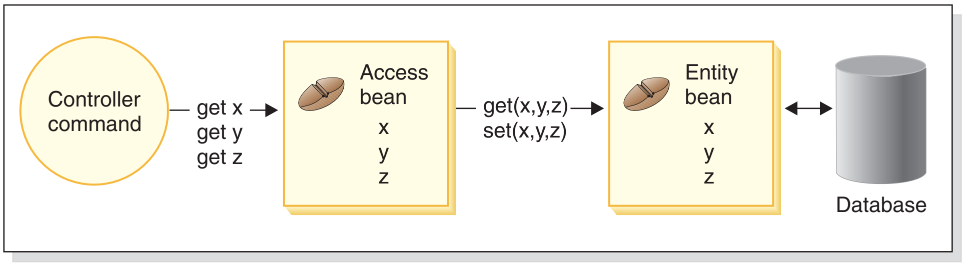


Figure 18.

Database considerations

As you customize your e-commerce application, you may create new database tables. When creating these tables, it is recommended that you follow a set of conventions, so that your tables are created in a manner consistent with the WebSphere Commerce tables.

Database schema object naming considerations

Subsequent sections provide guidance for the naming of database schema objects.

Naming conventions for tables and views

The following list provides guidance for the naming of new tables and views:

- In order to avoid name collision (duplicate names) with WebSphere Commerce tables and views in future releases, the first character in the table or view name should be `X`. For example, `XMYTABLE`.

- The table or view name should be no more than 10 characters in length. If the desired name exceeds this limit, shorten the length by removing vowels from the end of the name, until only 10 characters remain.
- The table or view name should not contain any special characters, such as "_", "+", "$", "%", or blank spaces.
- Do not use database reserved words as a table or view name.
- View names should end with `VW`.
- The table and view names should be singular nouns.

Naming conventions for columns

The following list provides guidance for the naming of columns in new tables:

- In order to avoid name collision (duplicate names) with columns in WebSphere Commerce tables in future releases, the first character in the column name should be `X`. For example, `XMYCOLUMN`.
- The column name should be no more than 18 characters in length. If the desired name exceeds this limit, shorten the length by removing vowels from the end of the name, until there are only 18 characters.
- Columns names (other than foreign keys) should not contain any special characters, such as "_", "+", "$", "%", or blank spaces.
- Do not use database reserved words as a column name.
- Combined words may be used as column names using the active voice combination. For example, `COMBINERESULT`.
- The generated primary key columns should be named as *`table`*`_id`. For example, the primary key for the USERS table is `USERS_ID`.
- The generated foreign key column names should not be changed.
- If reserving any columns for future customization, they should be named `field`*`x`* where *`x`* is a numeric digit starting from 1.

Naming conventions for indexes

The following list provides guidance for the naming of indexes in new tables:

- The index name should be no more than 18 characters in length.
- The index name should not contain blank spaces.
- The index name should not contain any database reserved words.
- A non-unique index should be named as `I_`*`tablex`* where *`table`* is the name of the table and *`x`* is a number, beginning at 1. For example, a non-unique index for the USERS table is `I_USERS1`.
- A unique index should be named as `UI_`*`tablex`* where *`table`* is the name of the table and *`x`* is a number, beginning at 1. For example, a unique index for the USERS table is `UI_USERS1`.
- The total size of the index should be no larger than 254 bytes.
- The index name must be unique across the whole database schema.

Naming conventions for primary keys

The following list provides guidance for the naming of primary keys for new tables:

- The primary key name should be no more than 18 characters in length.
- The primary key name should not contain blank spaces.
- The primary key name should not contain any database reserved words.
- The primary key should be named as `P_`*`table`* where *`table`* is the name of the table. For example, the primary key for the USERS table is `P_USERS`.
- The primary key name must be unique across the whole database schema.

Naming conventions for foreign keys

The following list provides guidance for the naming of foreign keys for new tables:

- The foreign key name should be no more than 18 characters in length.
- The foreign key name should not contain blank spaces.
- The foreign key name should not contain any database reserved words.
- The foreign key should be named as `F_`*`table`* where *`table`* is the name of the table. For example, the foreign key for the USERS table is `F_USERS1`.
- The foreign key name must be unique across the whole database schema.

Naming conventions for database triggers

The following list provides guidance for the naming of database triggers:

- The database trigger name should be no more than 18 characters in length.
- The database trigger name should not contain blank spaces.
- The database trigger name should not contain any database reserved words.
- The database trigger should be named as `T_`*`table`* where *`table`* is the name of the table. For example, the database trigger name for the USERS table is `T_USERS1`.
- The database trigger name must be unique across the whole database schema.

Database column datatype considerations

This section introduces column datatypes that can be used when creating new tables. The descriptions of the various datatypes use DB2 terminology. "Datatype differences between databases" on page 82 describes the differences if you are using a different database.

BIGINT

This is a 64-bit signed integer that has a range from -9223372036854775807 to 9223372036854775807. Contrast this to INTEGER, which is only half the size of BIGINT.

INTEGER

This is a 32-bit signed integer that has a range from -2147483647 to 2147483647. In general, INTEGER should be the default finite numeric datatype, instead of BIGINT. Unless there is a strong business reason for using BIGINT, for performance reasons it is better to use INTEGER as the numeric data type. A common user of the BIGINT datatype is a system generated key.

The use of either SMALLINT or SHORT datatypes is strongly discouraged because these data types are mapped to a non-object Java data type and these non-object data types will cause problems in some enterprise bean object instantiations.

TIMESTAMP
This is a is a seven-part value (year, month, day, hour, minute, second, and microsecond) that designates a date and time, except that the time includes a fractional specification of microseconds. The internal representation of a timestamp is a string of 10 bytes, each of which consists of 2 packed decimal digits. The first 4 bytes represent the date, the next 3 bytes the time, and the last 3 bytes the microseconds.

CHAR
This is a fixed-length character string of length INTEGER, which may range from 1 to 254 characters. If the length specification is omitted, a length of 1 character is assumed. Since CHAR is a fixed length database column, any unused trailing character spaces are changed into white spaces. Unless for performance reasons, it is not recommended to use CHAR datatype because CHAR is not flexible and length cannot be changed at a later time. As a rule of thumb, if your string column is less then 64 characters in length and is regularly retrieved or updated, use CHAR instead for better performance.

VARCHAR
This is a variable-length character string of maximum length integer, which may range from 1 to 32672. However, unlike CHAR where the column data is stored along with the table, VARCHAR is internally represented as a reference pointer inside a database page. Therefore, length of a VARCHAR column can be changed at any time after creation.

LONG VARCHAR
This is the variable-length character string that can be used if VARCHAR cannot be created within the same database page. LONG VARCHAR is very similar to VARCHAR except that it can span multiple database pages. Restrict the use of the LONG VARCHAR datatype to only those cases when it is absolutely required because LONG VARCHAR objects are typically expensive in terms of performance.

CLOB This another variable-length character string that can be used if the length of the column needs to exceed the 32KB limit of LONG VARCHAR. The length of a CLOB object can reach 1 GB without modifying the database configuration. Text data that is stored as CLOB is converted appropriately when moving among different systems.

BLOB This is a variable-length binary character string that stores unstructured data in the database. BLOB objects can store up to 4 GB of binary data. In general, you should avoid using BLOB as a column datatype, unless it is absolutely necessary. In terms of performance, a BLOB object is considered to be one of the most expensive objects in any database.

DECIMAL(20,5)
This datatype is specially defined to be used for most fixed decimal point numbers, such as currency units. For other floating point decimal numbers, FLOAT can be used instead.

Datatype differences between databases

The following table lists the datatypes used in the WebSphere Commerce database schema and shows the corresponding datatypes for different database implementations.

JDBC Object	Windows AIX Solaris Linux **DB2**	Windows AIX Solaris **Oracle®**	400 **DB2**
Hashtable	BLOB()	BLOB	BLOB()
Timestamp	TIMESTAMP	DATE	TIMESTAMP
Integer	INTEGER	INTEGER	INTEGER
BigDecimal	DECIMAL(,)	DECIMAL(,)	DECIMAL(,)
Long	BIGINT	NUMBER	BIGINT
Double	FLOAT	NUMBER	FLOAT
String	CHAR()	VARCHAR2()	GRAPHIC() CCSID 13488
byte[]	CHAR() for bit data	RAW()	CHAR() for bit data
String	VARCHAR()	VARCHAR2()	VARGRAPHIC() CCSID 13488

JDBC Object	Windows AIX Solaris Linux **DB2**	Windows AIX Solaris **Oracle®**	400 **DB2**
String	LONG VARCHAR	VARCHAR2() (See note following table for more details.)	VARGRAPHIC(4000) ALLOCATE() CCSID 13488
byte[]	LONG VARCHAR for bit data	LONG RAW	VARCHAR(8000) ALLOCATE() for bit data
String	CLOB()	CLOB()	DBCLOB() CCSID 13488

Note:

As a result of inconsistent rates of success for when the Oracle JDBC driver handles information that is of the LONG data type, it is recommended that you avoid using the LONG data type whenever possible. The most commonly reported error in this situation is the "Stream has already been closed" error.

If you must use this datatype, you can only have one column per database table that uses the LONG type. In addition, when constructing a select statement, do not put the LONG column as either the first or last element in the select. Another workaround for operations under a heavy load is to avoid mapping this particular column to a CMP field in an entity bean. Instead, use a session bean to perform retrieves and updates on this column.

Chapter 4. Access control

Understanding access control

The access control model of a WebSphere Commerce application has three primary concepts: users, actions and resources. Users are the people that use the system. Resources are the entities that are maintained in or by the application. For example, resources may be products, documents, or orders. User profiles that represent people are also resources. Actions are the activities that users can perform on the resources. Access control is the component of the e-commerce application that determines whether a given user can perform a given action on a given resource.

In a WebSphere Commerce application, there are two main levels of access control. The first level of access control is performed by the WebSphere Application Server. In this respect, WebSphere Commerce uses WebSphere Application Server to protect enterprise beans and servlets. The second level of access control is the fine-grained access control system of WebSphere Commerce.

The WebSphere Commerce access control framework uses access control policies to determine if a given user is permitted to perform a given action on a given resource. This access control framework provides fine-grained access control. It works in conjunction with, but does not replace the access control provided by the WebSphere Application Server.

Overview of resource protection in WebSphere Application Server

The following WebSphere Commerce resources are protected under access control by WebSphere Application Server:

- Entity beans
  These beans model objects in an e-commerce application. They are distributed objects that can be accessed by remote clients.
- JSP templates
  WebSphere Commerce uses JSP templates for display pages. Each JSP template can contain one or more data beans that retrieve data from entity beans. Clients can request JSP pages by composing a URL request.
- Controller and view commands
  Clients can request controller and view commands by composing URL requests. In addition, one display page may contain a link to another by using the JSP file name or the view name, as registered in the VIEWREG table.

The WebSphere Commerce Server is typically configured to use the following Web paths:

- `/webapp/wcs/stores/servlet/*`
  This is used for requests to the request servlet.
- `/webapp/wcs/stores/*.jsp`
  This is used for requests to the JSP servlet.

The following diagram shows the route that requests could potentially follow to access WebSphere Commerce resources, for the preceding Web path configuration.

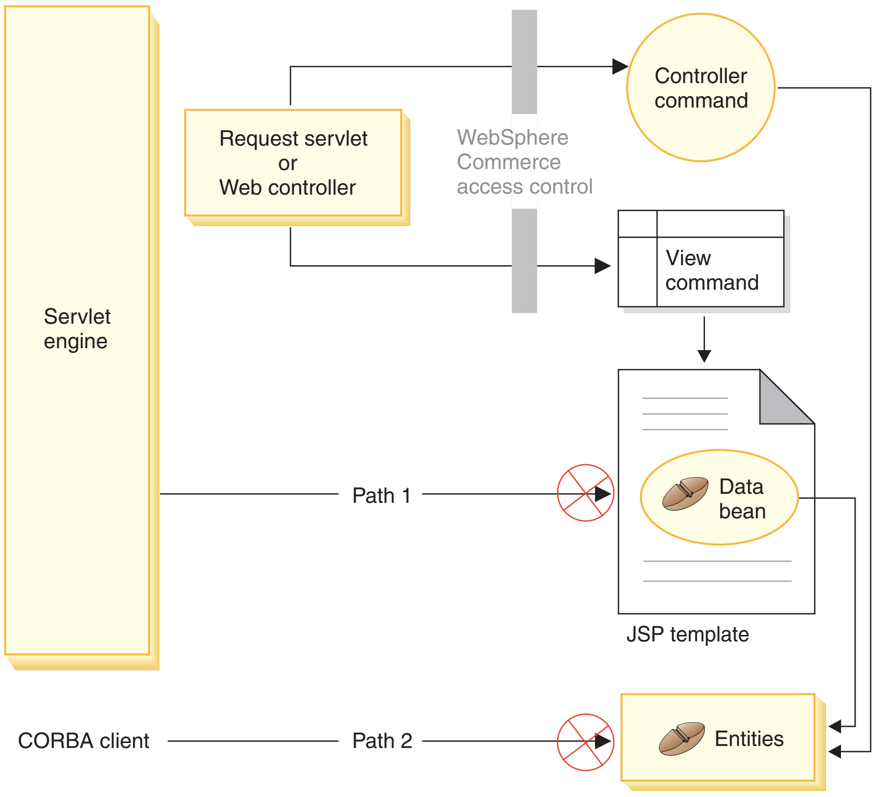


Figure 19.

All the legitimate requests should be directed to the request servlet, which then directs them to the Web controller. The Web controller implements access control for controller commands and views. The Web paths shown above do, however, make it possible for malicious users to directly access JSP templates (path 1) and entity beans (path 2). In order to prevent these malicious attacks from being successful, they must be rejected at run time.

Direct access to the JSP templates and entity beans can be prevented by using one of the following approaches:

WebSphere Application Server security

WebSphere Application Server provides a security feature. Using this approach, all enterprise bean methods and JSP templates are configured to be invoked by the System Identity only. To access these WebSphere Commerce resources, a URL request must be routed to the request servlet that sets the System Identity to the current thread, before passing it to the Web controller. The Web controller then ensures that the caller has the required authorization before passing the request to the corresponding controller command or view. Any attempts to directly access JSP templates and entity beans (that is, without using the Web controller) are rejected by the WebSphere Application Server security component.

For information about configuring WebSphere Application Server to secure WebSphere Commerce resources, refer to the *WebSphere Commerce Installation Guide*. For information about security within WebSphere Application Server, refer to the System Administration topic in the WebSphere Application Server documentation.

For information about configuring WebSphere Application Server security for methods in customized enterprise beans, refer to "Assembling new enterprise beans into an enterprise application" on page 357 and "Assembling modified enterprise beans into an enterprise application" on page 361.

Firewall protection

When a WebSphere Commerce Server runs behind the firewall, Internet clients are not able to directly access the entity beans. Using this approach, protection for JSP templates is provided by the data bean that is included in the page. The data bean is activated by the data bean manager. The data bean manager detects if the JSP template was forwarded by a view command. If it was not forwarded by a view command an exception is thrown and the request for the JSP template is rejected.

Introduction to WebSphere Commerce access control policies

The WebSphere Commerce access control model is based upon the enforcement of access control policies. Access control policies allow access control rules to be externalized from business logic code, thereby removing the need to hard code access control statements into code. For example, you do not need to include code similar to the following:

```
if (user.isAdministrator())
   then {}
```

Access control policies are enforced by the access control policy manager. In general, when a user attempts to access a protected resource, the access control policy manager first determines what access control policies are applicable for that protected resource, and then, based upon the applicable access control policies, it determines if the user is allowed to access the requested resources.

An access control policy is a 4-tuple policy that is stored in the ACPOLICY table. Each access control policy takes the following form:

```
AccessControlPolicy [UserGroup, ActionGroup, ResourceGroup, Relationship]
```

The elements in the 4-tuple access control policy specify that a user belonging to a specific user group is permitted to perform actions in the specified action group on resources belonging to the specified resource group, as long as the user satisfies the conditions specified in the relationship or relationship group, with respect to the resource in question. For example, `[AllUsers, UpdateDoc, doc, creator]` specifies that all users can update a document, if they are the creator of the document.

The user group is a specific type of member group that is defined in the MBRGRP database table. A user group must be associated with member group type of `-2`. The value of `-2` represents an access group and is defined in the MBRGRPTYPE table. The association between the user group and member group type is stored in the MBRGRPUSG table.

The membership of a user into a particular user group may be stated explicitly or implicitly. An explicit specification occurs if the MBRGRPMBR table states that the user belongs to a particular member group. An implicit specification occurs if the user satisfies a condition (for example, all users that fulfill the role of Product Manager) that is stated in the MBRGRPCOND table. There may also be combined conditions (for example, all users that fulfill the role of Product Manager and that have been in the role for at least 6 months) or explicit exclusions.

Most conditions to include a user in a user group are based upon the user fulfilling a particular role. For example, there could be an access control policy that allows all users that fulfill the Product Manager role, to perform catalog management operations. In this case, any user that has been assigned the Product Manager role in the MBRROLE table is then implicitly included in the user group.

For more details about the member group subsystem, refer to the WebSphere Commerce online help.

The ActionGroup element comes from the ACACTGRP table. An action group refers to an explicitly specified group of actions. The listing of actions is stored in the ACACTION table and the relationship of each action to its action group (or groups) is stored in the ACACTACTGP table. An example of an action group is the "OrderWriteCommands" action group. This action group includes the following actions that are used to update orders:

- com.ibm.commerce.order.commands.OrderDeleteCmd
- com.ibm.commerce.order.commands.OrderCancelCmd
- com.ibm.commerce.order.commands.OrderProfileUpateCmd
- com.ibm.commerce.order.commands.OrderUnlockCmd
- com.ibm.commerce.order.commands.OrderScheduleCmd
- com.ibm.commerce.order.commands.ScheduledOrderCancelCmd
- com.ibm.commerce.order.commands.ScheduledOrderProcessCmd
- com.ibm.commerce.order.commands.OrderItemAddCmd
- com.ibm.commerce.order.commands.OrderItemDeleteCmd
- com.ibm.commerce.order.commands.OrderItemUpdateCmd
- com.ibm.commerce.order.commands.PayResetPMCmd

A resource group is a mechanism to group together particular types of resources. Membership of a resource in a resource group can be specified in one of two ways:

- Using the conditions column in the ACRESGRP table
- Using the ACRESGPRES table

In most cases, it is sufficient to use the ACRESGPRES table for associating resources to resource groups. Using this method, resources are defined in the ACRESCGRY table using their Java class name. Then, these resources are associated with the appropriate resource groups (ACRESGRP table) using the ACRESGPRES association table. In cases where the Java class name alone is not sufficient to define the members of a resource group (for example, if you need to further restrict the objects of this class based on an attribute of the resource), the resource group can be defined entirely using the conditions column of the ACRESGRP table. Note that in order to perform this grouping of resources based on an attribute, the resource must also implement the Groupable interface.

The following diagram shows an example resource grouping specification. In this example resource group `10023` includes all the resources that are associated with it in the ACRESGPRES table. Resource group `10070` is defined using the conditions field column in the ACRESGRP table. This resource group includes instances of the Order remote interface, that also have status = "`Z`" (specifying a shared requisition list).

Note: Details about the XML information for the Conditions column of the ACRESGRP table are found in the *WebSphere Commerce Access Control Guide*.

ACRESGRP

AcResGrp_Id	GrpName	Conditions
10023	AccountRepresentatives CmdResourceGroup	null
10070	SharedRequisitionList ResourceGroup	<profile> <andListCondition> <simpleCondition> <variable name="Status"/> <operator name="="/> <value data="Z"/> </simpleCondition> <simpleCondition> <variable name="classname"/> <operator name="="/> <value data="com.ibm.commerce.order. objects.Order"/> </simpleCondition> </andListCondition> </profile>

ACRESGRPES

AcResGrp_Id	AcResCgry_Id
10023	10246
10023	10247
10023	10248
10023	10249
10023	10250

ACRESCGRY

AcResCgry_Id	ResClassname
10246	com.ibm.commerce.contract. commands.ContractCreateCmd
10247	com.ibm.commerce.contract. commands.ContractCreateCmd
10248	com.ibm.commerce.contract. commands.ContractCreateCmd
10249	com.ibm.commerce.contract. commands.ContractCreateCmd
10250	com.ibm.commerce.contract. commands.ContractCreateCmd

Figure 20.

The MEMBER_ID column of the ACACTGRP, ACRESGRP, and ACRELGRP tables should have a value of `-2001` (Root Organization).

The access control policy can optionally include either a Relationship or RelationshipGroup element as its fourth element.

If your access control policy uses a Relationship element, this comes from the ACRELATION table. If, on the other hand, it includes a RelationshipGroup element, that comes from the ACRELGRP table. Note that neither need be included, but if you include one, you cannot include the other. A RelationshipGroup specification from the ACRELGRP table takes precedence over the Relationship information from the ACRELATION table.

The ACRELATION table specifies the types of relationships that exist between users and resources. Some examples of types of relationships include creator, submitter, and owner. An example of the use of the relationship element is to use it to ensure that the creator of an order can always update the order.

The ACRELGRP table specifies the types of relationship groups that can be associated with particular resources. A relationship group is a grouping of one or more relationship chains. A relationship chain is a series of one more relationships. An example of a relationship group is to specify that a user must be the creator of the resource and also belong to the buying organizational entity that is referenced in the resource.

The relationship group (or relationship) specification is an optional part of the access control policy. It is commonly used if you have created your own commands and these commands are not restricted to certain roles. In these cases, you might want to enforce a relationship between the user and the resource. Typically, if commands are to be restricted to certain roles, it is accomplished through the UserGroup element of the access control policy rather than by using the Relationship element.

Another important concept related to access control policies is the concept of an access control policy *owner*. An access control policy owner is the organizational entity that owns the access control policy. Knowing the owner of an access control policy is important because an access control policy can only be applied to resources that are owned by the access control policy owner.

For each resource in question, the access control policy manager applies access control policies that are owned by the owning organizational entity or by its ancestor organizational entities in the member hierarchy, until either a policy is found that grants permission or until all policies have been checked and none grant permission.

Consider the following diagram showing a member hierarchy.

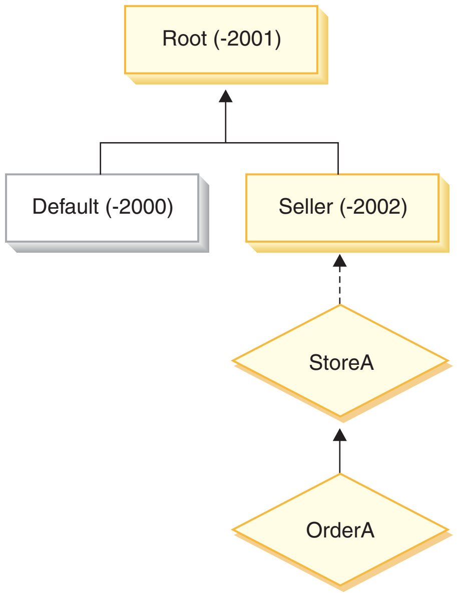


Figure 21.

For the resource “OrderA”, any access control policy that is owned by the Seller or Root organization can be applied. If the access control policy manager finds one policy owned by either of these organizations that grants the user permission (based upon the four elements in the access control policy) it immediately stops searching through the access control policies. However, if it does not find any access control policies owned by those organizations that grant the user permission to perform the action on the protected resources, then access is denied.

Relationship groups

A relationship group allows you to specify multiple relationships. A relationship can be directly between a user and the resource in question, or it can be a chain of relationships that indirectly relate the user to the resource.

Note: For the following sections related to relationship groups, it is important to recognize that the only organizations available in WebSphere Commerce Professional Edition are the RootOrganization, the DefaultOrganization, and the SellerOrganization. Examples that refer to other organizations only apply to WebSphere Commerce Business Edition.

Comparing relationships to relationship groups: Access control policies can specify that a user must fulfill a particular relationship with respect to the resource being accessed, or they can specify that a user must fulfill the conditions specified in a relationship group.

In most cases, specifying a relationship should satisfy the access control requirements for your application. If, however, the policy is such that you must specify a relationship that is not directly between the user and the resource, but that is actually a series of relationships between the user and the resource, you must then use a relationship group.

For example, if you must specify an association between a user and a buying organization where the relationship requires that the user is playing a particular role for that organization or that the user is a member of the buying organization, then you must use a relationship group and a chain of relationships.

If you merely need to enforce an association that is directly between the user and the resource in question, you can use a simple relationship. For example, this would be the case if you need to enforce that the user must be the creator of the resource.

If you combine multiple simple relationships, for example, the user must be the creator *or* the submitter, then this becomes a chain of relationships and you must use a relationship group. This combination of simple relationships may occur when using either WebSphere Commerce Professional Edition or WebSphere Commerce Business Edition.

General information about relationship groups: A relationship chain is a series of one more relationships. The length of a relationship chain is determined by the number of relationships that it contains. This can be determined by examining the number of `<parameter name="aName" value="aValue" />` elements in the XML representation of the relationship chain.

Only the last `<parameter name="Relationship" value="aValue" />` element must be handled by the fulfills() method of the resource. The rest are handled internally by the access control policy manager.

When a relationship chain has a length of 2, the first `<parameter name="aName" value="aValue" />` element is between a user and an organizational entity. The last `<parameter name="aName" value="aValue" />` element is between an organizational entity and the resource.

If you need to define relationship groups, you must do so by defining the relationship group information in an XML file. You can modify the `defaultAccessControlPolicies.xml` file, or create your own XML file. For more information about creating these XML-based information, refer to the *WebSphere Commerce Access Control Guide*.

The following sections show examples of different types of relationship groups.

Relationship groups composed of a single relationship chain: Business As part of an access control policy, you may be required to enforce that a user must belong to the organizational entity that is the BuyingOrganizationalEntity of the resource. This requires the creation of a relationship group that is composed of one relationship chain that has a length two. The relationship chain is said to be of length ″two″ because it consists of two separate relationships. The first relationship is between the user and its parent organizational entity. The user is the ″child″ in that relationship. For the second relationship, the access control policy manager checks if the parent organizational entity fulfills the BuyingOrganizationalEntity relationship with the resource. In other words, it returns ″true″ if it is the buying organizational entity of the resource.

The following XML snip is taken from the `defaultAccessControlPolicies.xml` file and shows how to define this type of relationship group:

```
<RelationGroup Name="MemberOf->BuyerOrganizationalEntity"
      OwnerID="RootOrganization">
   <RelationCondition><![CDATA[
      <profile>
         <openCondition name="RELATIONSHIP_CHAIN">
            <parameter name="HIERARCHY" value="child"/>
            <parameter name="RELATIONSHIP" value="BuyingOrganizationalEntity"/>
         </openCondition>
      </profile>
   ]]></RelationCondition>
</RelationGroup>
```

Business Another example would be to enforce that the user must have the role of Account Representative for the organizational entity that is the buying organizational entity of the resource in question. Again, this uses a relationship group that is composed of one relationship chain of length two. The first part of the chain will find all of the organizational entities for which the user has the Account Representative role. Then for this set of organizational entities, the access control policy manager checks if at least one of them fulfills the BuyingOrganizationalEntity relationship with the resource. In other words, it returns true if one of them is the buying organizational entity of the resource.

The following XML snip is taken from the `defaultAccessControlPolicies.xml` file and shows how to define this type of relationship group:

```
<RelationGroup Name="AccountRep->BuyerOrganizationalEntity"
      OwnerID="RootOrganization">
   <RelationCondition><![CDATA[
      <profile>
```

```
         <openCondition name="RELATIONSHIP_CHAIN">
            <parameter name="ROLE" value="Account Representative"/>
            <parameter name="RELATIONSHIP" value="BuyingOrganizationalEntity"/>
         </openCondition>
      </profile>
   ]]></RelationCondition>
</RelationGroup>
```

Relationship groups composed of multiple relationship chains: It is possible to compose a relationship group so that it contains multiple relationship chains. When doing so, you must specify whether the user must satisfy all of the relationship chains, meaning it is an *AND* scenario, or the user must satisfy at least one of the relationship chains, meaning it is an *OR* scenario.

Business To demonstrate this type of relationship, the following XML snip is used to enforce that a user must be the creator of the resource and the user must also belong to the BuyingOrganizationalEntity specified in the resource. The first chain, that specifies the user must be the creator of the resource is of length one. The second chain that specifies that the user must belong to the BuyingOrganizationalEntity specified in the resource is of length two.

```
<RelationGroup Name="Creator_And_MemberOf->BuyerOrganizationalEntity"
      OwnerID="RootOrganization">
 <RelationCondition><![CDATA[
  <profile>
   <andListCondition>
    <openCondition name="RELATIONSHIP_CHAIN">
     <parameter name="RELATIONSHIP" value="creator" />
    </openCondition>
    <openCondition name="RELATIONSHIP_CHAIN">
     <parameter name="HIERARCHY" value="child"/>
     <parameter name="RELATIONSHIP" value="BuyingOrganizationalEntity"/>
    </openCondition>
   </andListCondition>
  </profile>
 ]]></RelationCondition>
</RelationGroup>
```

If, instead of the *AND* scenario, you require the user to satisfy either of the two relationship chains, the `<andListCondition>` tag should be changed to the `<orListCondition>` tag.

Professional Business To demonstrate a relationship group that can be used in WebSphere Commerce Professional Edition (as well as WebSphere Commerce Business Edition), consider a relationship group that is used to enforce that the user must be either the creator or the submitter of the resource. This is shown in the following XML snip.

```
<RelationGroup Name="Creator_Or_Submitter"
     OwnerID="RootOrganization">
   <RelationCondition><![CDATA [
   <profile>
      <orListCondition>
         <openCondition name="RELATIONSHIP_CHAIN">
            <parameter name="RELATIONSHIP"value="creator"/>
         </openCondition>
         <openCondition name="RELATIONSHIP_CHAIN">
            <parameter name="RELATIONSHIP"value="submitter"/>
         </openCondition>
      </orListCondition>
   </profile>
   ]]></RelationCondition>
</RelationGroup>
```

Types of access control

There are two types of access control, both of which are policy-based: command-level access control and resource-level access control.

Command-level (also known as "role-based") access control uses a broad type of policy. You can specify that all users of a particular role can execute certain types of commands. For example, you can specify that users with the Account Representative role can execute any command in the AccountRepresentativesCmdResourceGroup resource group. Or, as depicted in the following diagram, another example policy is to specify that all store administrators can perform any action specified in the ExecuteCommandAction Group on any resource that is specified by the StoreAdminCmdResourceGrp.

Note: The XML information for the Conditions column of the MBRGRPCOND table is generated when you use the Administration Console to set up your access groups. For information about using the Administration Console to set up access groups, refer to the WebSphere Commerce online help.

ACPOLICY

PolicyName	Member_Id	MbrGrp_Id	AcActGrp_id	AcResGrp_Id	AcRelGrp_Id
StoreAdministrators ExecuteStoreAdmin CmdResourceGroup	-2001	-8	10052	10018	null

MBRGRP

MbrGrp_Id	MbrGrpName
-8	StoreAdministrators

MBRGRPCOND

MbrGrp_Id	Conditions
-8	`<profile> <simpleCondition> <variable name="role"/> <operator name="="/> <value data="Store Administrator"/> </simpleCondition> </profile>`

ACACTGRP

AcActGrp_Id	GroupName
10052	ExecuteCommandActionGroup

ACRESGRP

AcResGrp_Id	GrpName
10018	StoreAdminCmdResourceGroup

Figure 22.

A command-level access control policy always has the `ExecuteCommandActionGroup` as the action group for controller commands. For views, the resource group is always `ViewCommandResourceGroup`.

All controller commands must be protected by command-level access control. In addition, any view that can be called directly, or that can be launched by a redirect from another command (in contrast to being launched by forwarding to the view) must be protected by command-level access control.

Command-level access control does not consider the resource that the command would act upon. It merely determines if the user is allowed to execute the particular command. If the user is allowed to execute the command, a subsequent resource-level access control policy could be applied to determine if the user can access the resource in question.

Consider when a store administrator attempts to perform an administrative task. The first level of access control checking would be to determine if this user is allowed to execute the particular store administration command. Once it has been determined that the user is in fact permitted to do this (because store administrators are allowed to execute commands in the storeAdminCmds group), a resource-level access control policy may be invoked. This policy may state that store administrators are only permitted to perform administrative tasks for stores that are owned by the organization for which the user is a store administrator.

To summarize, in command-level access control the “resource” is the command itself and the “action” is merely to execute the command (in other words, to instantiate the command object). The access control check determines if the user is permitted to execute the command. By contrast, in resource-level access control the “resource” is any protectable resource that the command or bean accesses and the “action” is the command itself.

Access control interactions

This section presents the interaction diagram showing how access control works in the WebSphere Commerce access control policy framework.

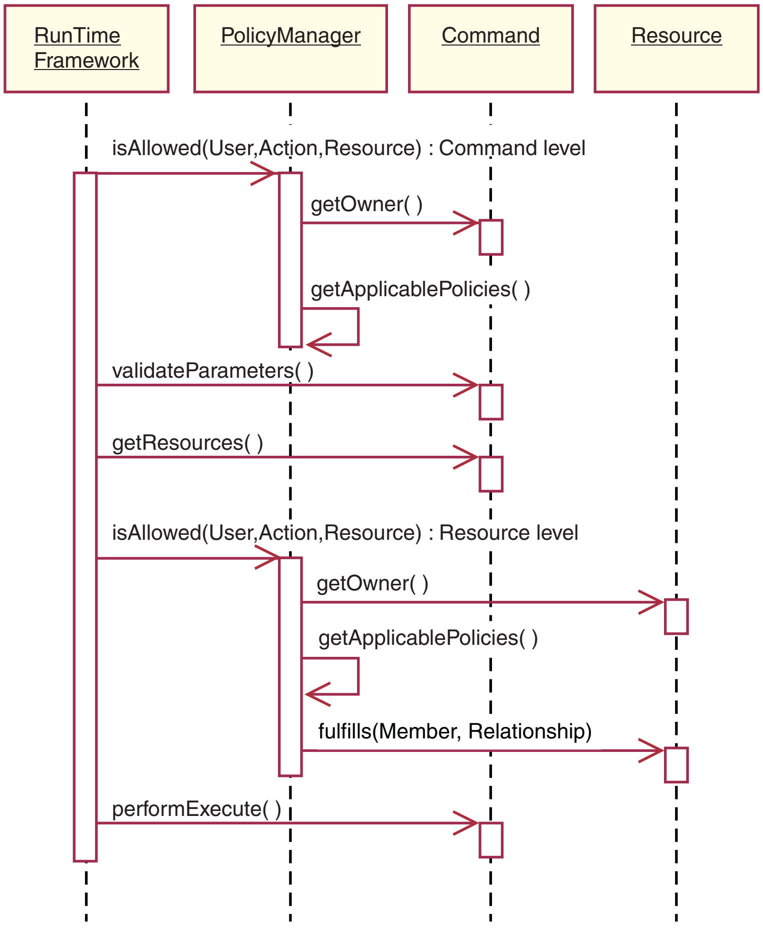


Figure 23.

The preceding diagram shows actions that are performed by the access control *policy manager*. The access control policy manager is the access control component that determines whether or not the current user is allowed to execute the specified action on the specified resource. It determines this by searching through the policies owned by the resource's owner and its ancestor organizations. If at least one policy grants access, then permission is granted.

The following list describes the actions from the preceding interaction diagram. They are ordered from the top of the diagram to the bottom.

1. `isAllowed()`
   The run-time components determine if the user has command level access for either the controller command or view.
2. `getOwner()`
   The access control policy manager determines the owner of the

command-level resource. The default implementation returns the member identifier (`memberId`) of the owner of the store (`storeId`) that is in the command context. If there is no store identifier in the command context, then the root organization (`-2001`) is returned.

3. `getApplicablePolicies()`
   The access control policy manager finds and processes the applicable policies, based on the specified user, action and resource.
4. `validateParameters()`
   Initial parameter checking and resolving.
5. `getResources()`
   Returns an access vector that is a vector of resource-action pairs.

   If nothing is returned, resource-level access control checking is not performed. If there are resources that should be protected an access vector (consisting of resource-action pairs) should be returned.

   Each *resource* is an instance of a protectable object (an object that implements the `com.ibm.commerce.security.Protectable` interface). In many cases, the resource is an access bean.

   An access bean may not implement the `com.ibm.commerce.security.Protectable` interface, however, the access control check can still occur as long as the corresponding enterprise bean is protected, according to the information included in "Implementing access control in enterprise beans" on page 103.

   The *action* is a string representing the operation to be performed on the resource. In most cases, the action is the interface name of the command.
6. `isAllowed()`
   The run-time components determine if the user has resource level access to all of the resource-action pairs specified by `getResources()`.
7. `getOwner()`
   The resource returns the `memberId` of its owner. This determines which policies apply. Only policies that are owned by the resource owner and its ancestor organizations apply.
8. `getApplicablePolicies()`
   The access control policy manager searches for applicable policies and then applies them. If at least one policy per resource-action pair that grants the user permission to access the resource is found, then access is granted, otherwise access it is denied.
9. `fulfills()`
   If an applicable policy has a relationship group specified, a check is done on the resource to see if the member satisfies the specified relationship or relationships, with respect to the resource.
10. `performExecute()`
    The business logic of the command.

Protectable interface

A key factor for having a resource protected by the WebSphere Commerce access control policies, is that the resource must implement the `com.ibm.commerce.security.Protectable` interface. This interface is most commonly used with enterprise beans and data beans, but only those particular beans that require protection need to implement the interface.

With the Protectable interface, a resource must provide two key methods: `getOwner()`, and `fulfills(Long member, String relationship)`.

Access control policies are owned by organizations or organizational entities. The `getOwner` method returns the memberId of the owner of the protectable resource. After the access control policy manager determines the owner of the resource, it also gets the memberId of each of the ancestors for the owner in the member hierarchy. All access control policies that belong to the owner from the original getOwner request as well as all access control policies that belong to any of the owner's ancestors are then applied.

Access control policies that apply to the specified owner, as well as access control policies that apply to any of the owner's higher level ancestors in the membership hierarchy, are applied.

The `fulfills` method only returns true if the given member satisfies the required relationship with respect to the resource. Typically the member is a single user, however it can also be an organization. It would be an organization if you are using a relationship group in the access control policy.

Groupable interface

The application of an access control policy is specific to a group of resources. Resource groupings can be made based upon attributes such as the class name, the state of an order or the storeId value.

If a resource is going to be grouped by an attribute other than its class name for the purpose of applying access control policies, it must implement the `com.ibm.commerce.grouping.Groupable` interface.

The following code snippet represents the Groupable interface:

```
Groupable interface {
 Object getGroupingAttributeValue (String attributeName, GroupContext context)
}
```

For example, to implement a policy that only applies to orders that are in the pending state (`status = P (pending)`), the remote interface of the Order entity bean implements the Groupable interface and the value for attributeName is set to `"status"`.

Usage of the Groupable interface is rare.

Finding more information about access control

For more information about the WebSphere Commerce access control model, refer to the *WebSphere Commerce Access Control Guide*. This guide provides a detailed overview of access control and describes how to use the Administration Console to create or modify policies, action groups, and resource groups.

Implementing access control

This section describes how to implement access control in customized code.

Identifying protectable resources

In general, enterprise beans and data beans are resources that you may want to protect. However, not all enterprise beans and data beans should be protected. Within the existing WebSphere Commerce application, resources that require protection already implement the protectable interface. The question of what to protect comes into play when you create new enterprise beans and data beans. Deciding which resources to protect depends upon your application.

If a command returns an enterprise bean in the `getResources` method, then the enterprise bean must be protected because the access control policy manager will call the getOwner method on the enterprise bean. The `fulfills` method will also be called if a relationship is specified in the corresponding resource-level access control policy.

If you were to implement the protectable interface (and therefore, put the resource under protection) for all of your own enterprise beans and data beans, your application could require many policies. As the number of policies increases, performance may degrade and policy management becomes more challenging.

A theoretical distinction is made between primary resources and dependent resource. A *primary resource* can exist upon its own. A *dependent resource* exists only when its related primary resource exists. For example, in the out-of-the-box WebSphere Commerce application code, the Order entity bean is a protectable resource, but the OrderItem entity bean is not. The reason for this is that the existence of an OrderItem depends upon an Order -- the Order is the primary resource and the OrderItem is a dependent resource. If a user should have access to an Order, it should also have access to the items in the order.

Similarly, the User entity bean is a protectable resource, but the Address entity bean is not. In this case, the existence of the address depends on the user, so anything that has access to the user, should also have access to the address.

Primary resources should be protected, but dependent resources often do not require protection. If a user is allowed to access a primary resource, it makes sense that, by default, the user should also be allowed to access its dependent resources.

Implementing access control in enterprise beans

If you create new enterprise beans that require protection by access control policies, you must do the following:

1. Create a new enterprise bean, ensuring that it extends from `com.ibm.commerce.base.objects.ECEntityBean`.
2. Ensure that the remote interface of the bean extends the `com.ibm.commerce.security.Protectable` interface.
3. If resources with which the bean interacts are grouped by an attribute other than the resource's Java class name, the remote interface of the bean must also extend the `com.ibm.commerce.grouping.Groupable` interface.
4. The enterprise bean class contains default implementations for the following methods:
   - `getOwner`
   - `fulfills`
   - `getGroupingAttributeValue`

   Override any methods that you need. At a minimum, you must override the `getOwner` method.
   The default implementations of these methods are shown in the following code snippets.

```
************************************************************************
public Long getOwner() throws Exception, java.rmi.RemoteException
{
   return null;
}
************************************************************************
************************************************************************
public boolean fulfills(Long member, String relationship)
   throws Exception, java.rmi.RemoteException
{
      return false;
}
************************************************************************
************************************************************************
public Object getGroupingAttributeValue(String attributeName,
   GroupingContext context) throws Exception, java.rmi.RemoteException
{
      return null;
}
************************************************************************
```

The following are sample implementations of these methods based on the OrderBean bean:

```
***************************************************************************
public Long getOwner() throws Exception, java.rmi.RemoteException
{
   com.ibm.commerce.common.objects.StoreEntityAccessBean storeEntAB = new
   com.ibm.commerce.common.objects.StoreEntityAccessBean();
   storeEntAB.setInitKey_storeEntityId(getStoreEntityId().toString());
   return storeEntAB.getMemberIdInEJBType();
}
***************************************************************************
***************************************************************************
public boolean fulfills(Long member, String relationship)
   throws Exception, java.rmi.RemoteException
   {
      if (relationship.equalsIgnoreCase("creator"))
      {
         return member.equals(getMemberId());
      }
      else if (relationship.equalsIgnoreCase (
         com.ibm.commerce.base.helpers.EJBConstants.
         SAME_ORGANIZATIONAL_ENTITY_AS_CREATOR_RELATION)) {
            com.ibm.commerce.user.objects.UserAccessBean creator = new
               com.ibm.commerce.user.objects.UserAccessBean();
            creator.setInitKey_MemberId(getMemberId().toString());
            com.ibm.commerce.user.objects.UserAccessBean ab = new
               com.ibm.commerce.user.objects.UserAccessBean();
            ab.setInitKey_MemberId(member.toString());
            if (ab.getParentMemberId().equals(creator.getParentMemberId()))
               return true;
          }
      return false;
}
***************************************************************************
***************************************************************************
public Object getGroupingAttributeValue(String attributeName,
   GroupingContext context) throws Exception
   {
      if (attributeName.equalsIgnoreCase("Status"))
         return getStatus();
   return null;
   }
***************************************************************************
```

5. Create (or recreate) the enterprise bean's access bean and generated code.

Implementing access control in data beans

If a data bean is to be protected, it can either be directly, or indirectly protected by access control policies. If a data bean is directly protected, then there exists an access control policy that applies to that particular data bean. If a data bean is indirectly protected, it delegates protection to another data bean, for which an access control policy exists.

If you create a new data bean that is to be directly protected by an access control policy, the data bean must do the following:

1. Implement the `com.ibm.commerce.security.Protectable` interface. As such, the bean must provide an implementation of the `getOwner()` and `fulfills(Long member, String relationship)` methods. These should be implemented on the remote interface of the bean.

   When a data bean implements the Protectable interface, the data bean manager calls the `isAllowed` method to determine if the user has the appropriate access control privileges, according to the current access control policy. The `isAllowed` method is described by the following code snippet:

   ```
   IsAllowed(Context, "Display", protectable_databean);
   ```
2. If resources that the bean interacts with are grouped by an attribute other than the resource's Java class name, the bean must implement the `com.ibm.commerce.grouping.Groupable` interface.
3. Implement the `com.ibm.commerce.security.Delegator` interface. This interface is described by the following code snippet:

   ```
   Interface Delegator {
    Protectable getDelegate();
   }
   ```

   Note: In order to be directly protected, the `getDelegate` method should return the data bean itself (that is, the data bean delegates to itself for the purpose of access control).

The distinction between which data beans should be protected directly versus which should be protected indirectly is similar to the distinction between primary and dependent resources. If the data bean object can exist on its own, it should be directly protected. If the existence of data bean depends upon the existence of another data bean, then it should delegate to the other data bean for protection.

An example of a data bean that would be directly protected is the Order data bean. An example of a data bean that would be indirectly protected is the OrderItem data bean.

If you create a new data bean that is to be indirectly protected by an access control policy, the data bean must do the following:

1. Implement the `com.ibm.commerce.security.Delegator` interface. This interface is described by the following code snippet:

```
Interface Delegator {
 Protectable getDelegate();
}
```

Note: The data bean returned by `getDelegate` must implement the Protectable interface.

If a data bean does not implement the Delegator interface, it is populated without the protection of access control policies.

Implementing access control in controller commands

When creating a new controller command, the implementation class for the new command should extend the `com.ibm.commerce.commands.ControllerCommandImpl` class and its interface should extend the `com.ibm.commerce.command.ControllerCommand` interface.

For command level policies for controller commands, the interface name of the command is specified as a resource. In order for a resource to be protected, it must implement the Protectable interface. According to the WebSphere Commerce programming model, this is accomplished by having the command's interface extend from `com.ibm.commerce.command.ControllerCommand` interface, and the command's implementation extend from `com.ibm.commerce.commands.ControllerCommandImpl`. The `ControllerCommand` interface extends `com.ibm.commerce.command.AccCommand` interface, which in turn extends `Protectable`. The `AccCommand` interface is the minimum interface that a command should implement in order to be protected by command level access control.

If the command accesses resources that should be protected, create a private instance variable of type `AccessVector` to hold the resources. Then override the `getResources` method since the default implementation of this method is to return a null value and therefore, no resource checking occurs.

In the new `getResources` method, you should return an array of resources or of resource-action pairs upon which the command can act. When an action is not explicitly specified, the action defaults to the interface name of the command being executed.

Additionally, it is recommended that the method determines if it must instantiate the resource or if it can use the existing instance variable holding the reference to the resource. Checking to see if the resource object already exists can help to improve system performance. You can then use the same `getResources` method, if required, in the `performExecute` method of the new controller command.

The following is an example of the `getResources` method:

```
private AccessVector resources = null;

public AccessVector getResources() throws ECException {

   if (resources == null) {
      OrderAccessBean orderAB = new OrderAccessBean();
      orderAB.setInitKey_orderId(getOrderId().toString());
      resouces = new AccessVector(orderAB);
      }
   return resources;
}
```

As an example, consider the OrderItemUpdate command. The `getResources` method of this command returns the Order and User protectable objects. Since the action is not specified, it defaults to the interface for the OrderItemUpdate command.

Multiple resources may be returned by the `getResources` method. When this occurs, a policy that gives the user access to all of the specified resources must be found if the action is to be carried out. If a user had access to two out of three resources, the action may not proceed (three out of three would be required).

If you need to perform additional parameter checking or resolving of parameters in the controller command, you can use the `validateParameters()` method. This is optional.

Additional resource level checking

It is not always possible to determine all of the resources that need to be protected, at the time the `getResources` method of the controller command is called.

If necessary, a task command can also implement a `getResources` method to return a list of resources, upon which the command can act.

Another way to invoke resource level checking is to make direct calls to the access control policy manager, using the `checkIsAllowed(Object resource, String action)` method. This method is available to any class that extends from the `com.ibm.commerce.command.AbstractECTargetableCommand` class. For example, the following classes extend from the `AbstractECTargetableCommand` class:

- `com.ibm.commerce.command.ControllerCommandImpl`
- `com.ibm.commerce.command.DataBeanCommandImpl`

The `checkIsAllowed` method is also available to classes that extend the `com.ibm.commerce.command.AbstractECCommand` class. For example, the following class extends from the `AbstractECCommand` class:

- `com.ibm.commerce.command.TaskCommandImpl`

The following shows the signature of the `checkIsAllowed` method:

```
void checkIsAllowed(Object resource, String action)
   throws ECException
```

This method throws an ECApplicationException if the current user is not allowed to perform the specified action on the specified resource. If access is granted, then the method simply returns.

Access control for “create” commands

Since the `getResources` method is called before the `performExecute` method in a command, a different approach must be taken for access control for resources that are not yet created. For example, if you have a `WidgetAddCmd`, the `getResources` method cannot return the resource that is about to be created. In this case, the `getResources` method should return the creator of the resources. For example, a command is created by a command factory, an order is created within a store, and a user is created within an organization.

Default implementations for command-level access control

For command-level access control, the default implementation of the getOwner() method returns the memberId of the store owner, if the storeId is specified. If the storeId is not specified, the memberId of the root organization is returned (`memberId = -2001`).

The default implementation of the `getResources()` method returns `null`.

The default implementation of the `validateParameters()` does nothing.

Implementing access control policies in views

Resource-level access control for views is performed by the data bean manager. The data bean manager is invoked in the following cases:

1. When the JSP template includes the `<useBean>` tag and the data bean is not in the attribute list.
2. When the JSP template includes the following activate method:
   ```
   DataBeanManager.activate(xyzDatabean, request);
   ```

Note: Any data bean that is to be protected (either directly or indirectly) must implement the Delegator interface. Any data bean that is to be directly protected will delegate to itself, and thus must also implement the Protectable interface. Data beans that are indirectly protected should delegate to a data bean that implements the Protectable interface.

While it is not recommended, a bypass of the access control checks occurs in the following cases:

1. If the JSP template makes direct calls to access beans, rather than using data beans.

2. If the JSP template invokes the data bean's populate() method directly.

If the results of a controller command are to be forwarded to a view (using the ForwardViewCommand), then command-level access control is not performed on the views. Furthermore, if the controller command puts the populated data beans (that are used in the view) on the attribute list of the response property and then forwards to a view, the JSP template can access the data without going through the data bean manager. This does require that the `<useBean>` tags are used in the JSP template. This can be a way to make a JSP template more efficient, since it can bypass any redundant resource-level access control checks on resources (data beans) to which the user has already been granted access via the controller command.

Chapter 5. Error handling and messages

Command error handling

WebSphere Commerce uses a well-defined command error handling framework that is simple to use in customized code. By design, the framework handles errors in a manner that supports multicultural stores. The following sections describe the types of exceptions that a command can throw, how the exceptions are handled, how message text is stored and used, how the exceptions are logged, and how to use the provided framework in your own commands.

Types of exceptions

A command can throw one of the following exceptions:

ECApplicationException
This exception is thrown if the error is related to the user. For example, when a user enters an invalid parameter, an ECApplicationException is thrown. When this exception is thrown, the Web controller does not retry the command, even if it is specified as a retriable command.

ECSystemException
This exception is thrown if a run-time exception or a WebSphere Commerce configuration error is detected. Examples of this type of exception include null-pointer exceptions and transaction rollback exceptions. When this type of exception is thrown, the Web controller retries the command if the command is retriable and the exception was caused by either a database deadlock or database rollback.

Both of the above listed exceptions are classes that extend from the ECException class, which is found in the com.ibm.commerce.exception package.

In order to throw one of these exceptions, the following information must be specified:

- Error view name
  The Web controller looks up this name in the VIEWREG table.
- ECMessage object
  This value corresponds to the message text contained within a properties file.
- Error parameters
  These name-value pairs are used to substitute information into the error message. For example, a message may contain a parameter to hold the

name of the method which threw the exception. This parameter is set when the exception is thrown, then when the error message is logged, the log file contains the actual method name.

- Error data
  These are optional attributes that can be made available to the JSP template through the error data bean.

Exception handling is tightly integrated with the logging system. When an exception is thrown, it is automatically logged.

Error message properties files

In order to simplify the maintenance of error messages and to support multilingual stores, the text for error messages is stored in properties files. WebSphere Commerce message text is stored in the `ecServerMessages_*XX_XX*.properties` file, where `_*XX_XX*` is the locale indicator (for example, `_en_US`).

The command context returns an identifier to indicate the language used by the client. When a message is required, the Web controller determines which properties file to use based upon the language identifier.

There are two types of messages defined in the `ecServerMessages*XX_XX*.properties` file: user messages and system messages. User messages are displayed to customers in their browsers. Both system and user messages are captured automatically in the message log.

When an error is thrown, one of the required parameters is a message object. For ECSystemExceptions, the message object must contain two keys, one for the system message and one for the user message. For ECApplicationExceptions, the message object contains the key for the user message (system messages are not used).

All system messages are predefined. You cannot create your own system messages. Therefore, when customized code throws an ECSystemException, it must specify a message key for one of the predefined system messages. Customized user messages can be created. New user messages must be stored in a separate properties file.

Exception handling flow

The following diagram shows the flow of information when an exception is caught. A description of each step follows.

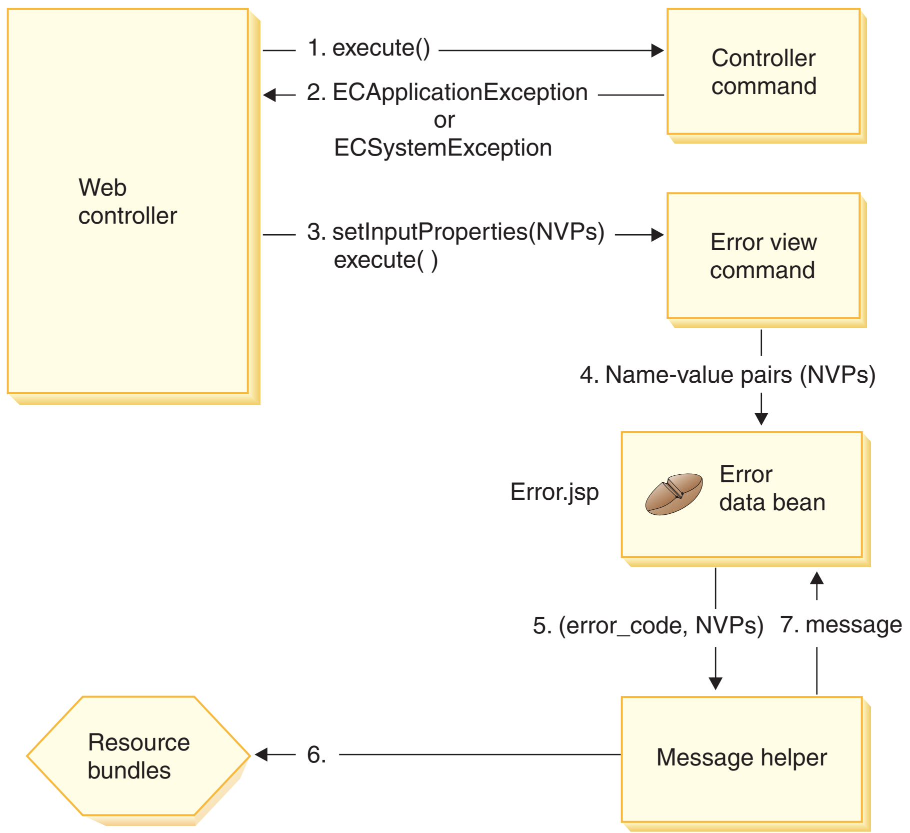


Figure 24.

1. The Web controller invokes a controller command.
2. The command throws an exception that is caught by the Web controller. This can be either an ECApplicationException, or an ECSystemException. The exception object contains the following information:
   - Error view name
   - ECMessage object
   - Error parameters
   - (optional) Error data
3. The Web controller determines the error view name from the VIEWREG table and invokes the specified error view command. When invoking the command, the Web controller composes a set of properties from the ECException object and sets it to the view command using the view command's setInputProperties method.
4. The view command invokes an error JSP template (Error.jsp in this case) and the name-value pairs are passed to the JSP template.

5. The ErrorDataBean passes the error parameters to the message helper object.
6. The message helper object gets the required message (using the message object and the error parameters) from the appropriate properties file.
7. The error data bean returns the message to the JSP template.

Exception handling in customized code

When creating new commands, it is important to include appropriate exception handling. You can take advantage of the error handling and messaging framework provided in WebSphere Commerce, by specifying the required information when catching an exception.

Writing your own exception handling logic, involves the following steps:

1. Catching the exceptions in your command that require special processing.
2. Constructing either an ECApplicationException or ECSystemException, based upon the type of exception caught.
3. If the ECApplicationException uses a new message, defining the message in a new properties file.

Catching and constructing exceptions

To illustrate the first two steps, the following code snippet shows an example of catching a system exception within a command:

```
try {
// your business logic
}
catch(FinderException e) {
     throw new ECSystemException (ECMessage._ERR_FINDER_EXCEPTION,
          className, methodName, new Object [] {e.toString()}, e);
}
```

The preceding _ERR_FINDER_EXCEPTION ECMessage object is defined as follows:

```
public static final ECMessage _ERR_FINDER_EXCEPTION =
new ECMessage (ECMessageSeverity.ERROR, ECMessageType.SYSTEM,
     ECMessageKey._ERR_FINDER_EXCEPTION);
```

The _ERR_FINDER_EXCEPTION message text is defined within the `ecServerMessages_xx_XX.properties` file (where `_xx_XX` is a locale indicator such as *_en_US*), as follows:

```
_ERR_FINDER_EXCEPTION  =
     The following Finder Exception occurred during processing: "{0}".
```

When catching a system exception, there is a predefined set of messages that can be used. These are described in the following table:

Message Object	Description
_ERR_FINDER_EXCEPTION	Thrown when an error is returned from an EJB finder method call.
_ERR_REMOTE_EXCEPTION	Thrown when an error is returned from an EJB remote method call.
_ERR_CREATE_EXCEPTION	Thrown when an error occurs creating an EJB instance.
_ERR_NAMING_EXCEPTION	Thrown when an error is returned from the name server.
_ERR_GENERIC	Thrown when an unexpected system error occurs. For example, a null pointer exception.

When catching an application exception, you can either use an existing message that is specified in the appropriate `ecServerMessages_xx_xx.properties` file, or create a new message that is stored in a new properties file. As specified previously, you *must not* modify any of the `ecServerMessages_xx_XX.properties` files.

The following code snippet shows an example of catching an application exception within a command:

```
try {
// your business logic

}
// catch some new type of application exception
catch(//your new exception)
 {
     throw new ECApplicationException (MyMessages._ERR_CUSTOMER_INVALID,
          className, methodName, errorTaskName, someNVPs);
}
```

The preceding _ERR_CUSTOMER_INVALID ECMessage object is defined as follows:

```
public static final ECMessage _ERR_CUSTOMER_INVALID =
     new ECMessage (ECMessageSeverity.ERROR, ECMessageType.USER,
     MyMessagesKey._ERR_CUSTOMER_INVALID, "ecCustomerMessages");
```

When constructing new user messages, you should assign them with a type of USER, as follows:

```
ECMessageType.USER
```

The text for the _ERR_CUSTOMER_INVALID message is contained in the `ecCustomerMessages.properties` file. This file must reside in a directory that is in the class path. The text is defined as follows:

```
_ERR_CUSTOMER_INVALID = Invalid ID "{0}"
```

Creating messages

If your command throws an ECApplicationException that uses a new message, you must create this new message. Creating a new message involves the following steps:

1. Creating a new class that contains the message keys.
2. Creating a new class that contains the ECMessage objects.
3. Creating a resource bundle.
4. Unit testing the message.

Details about each step are found in the following sections.

Creating a class for message keys

The first step in creating new user messages is to create a class that contains the new message keys. A message key is a unique indicator that is used by the logging service to locate the corresponding message text in a resource bundle. This new class must be created within your own package and stored in a project separate from the WebSphere Commerce projects.

Consider an example, called `MyNewMessages`, in which you create a new class, called `MyMessageKeys` that contains the `_ERR_CUSTOMER` and `_ERR_CUSTOMER_INVALID_ID` message keys and you put this class in the `com.mycompany.messages` package. In this case, the class definition appears as follows:

```
public class MyMessageKeys
{
    public static String _ERR_CUSTOMER="_ERR_CUSTOMER";
    public static String _ERR_CUSTOMER_INVALID_ID="_ERR_CUSTOMER_INVALID_ID";
}
```

Providing String wrappers for message keys allows the compiler to check their validity.

Creating a class for ECMessage objects

Within the same package that you created the class for your message keys, create another class that contains the ECMessage objects. The ECMessage class defines the structure of a message object. It is used to retrieve and persist locale-sensitive text messages.

The message object has the following attributes: severity, type, key, resource bundle and associated resource bundle. There are several constructor methods for this class. Refer to the “Reference” section of the WebSphere Commerce online help for complete details.

Following the MyNewMessages example, create a new class called `MyMessages` within the `com.mycompany.messages` package, as follows:

```
import com.ibm.commerce.ras.*;

public class MyMessages
    {
        static String myResourceBundle = "ecCustomerMessages";

        public static ECMessage _ERR_CUSTOMER = new ECMessage
             (ECMessageSeverity.ERROR,ECMessageType.USER,
                   MyMessageKeys._ERR_CUSTOMER, myResourceBundle);
        public static ECMessage _ERR_CUSTOMER_INVALID_ID = new ECMessage
             (ECMessageSeverity.ERROR, ECMessageType.USER,
                  MyMessageKeys._ERR_CUSTOMER_INVALID_ID,
                  myResourceBundle);

    }
```

In the preceding code snippet, the import statement is required for the creation of the ECMessage object. The object MyMessage._ERR_CUSTOMER is a user message of severity `ERROR`. The MyMessageKeys._ERR_CUSTOMER is used by the WebSphere Commerce logging service to find the message text contained in the *ecCustomerMessages* properties file.

Creating a user message resource bundle

You must create a new resource bundle, in which the message keys with corresponding message text are stored. This resource bundle can be implemented either as a Java object, or as a properties file. It is recommended that you use properties files, since they are easier to translate and maintain. Properties files are used for WebSphere Commerce messages.

To continue the MyNewMessages example, create a text file by the name of `ecCustomerMessages.properties`. If the messages are to be used by a single store servlet, place this file in the following directory:

drive`:\WebSphere\AppServer\installedApps\WC_EnterpriseApp_`*instanceName*`.ear\`
`wcstores.war\WEB-INF\classes`

If the messages are to be used by a single tools servlet, place this file in the following directory:

drive`:\WebSphere\AppServer\installedApps\WC_EnterpriseApp_`*instanceName*`.ear\`
`wctools.war\WEB-INF\classes`

If the messages are used globaly by any servlet in the enterprise application, place the file in the following directory:

drive`:\WebSphere\AppServer\installedApps\WC_EnterpriseApp_`*instanceName*`.ear\`
`properties`

Since the properties file contains pairs of message keys and the corresponding message text, the `ecCustomerMessages.properties` file contains the following lines:

```
_ERR_CUSTOMER_MESSAGE = The customer message "{0}".
_ERR_CUSTOMER_INVALID_ID = Invalid ID "{0}".
```

Unit testing a message

Once the classes containing the message keys, the classes containing ECMessage objects, and the resource bundle have been created, you should unit test your new messages.

In order to test the new messages described in the preceding sections, do the following:

1. Create a new class for testing purposes. In this case, create `MyTestingClass`.
2. Add the following import statement to the class
   ```
   import com.ibm.commerce.ras.*;
   ```
3. Add a main() method to the class.
4. Examine the following code snippet. Modify it according to your own requirements (for example, ensure that the directory path points to a valid configuration file on your system) and insert it into the main() method
   ```
   // the fileName String variable should point
   // to a valid WebSphere Commerce configuration file:
     String fileName = "E:\\WebSphere\\CommerceServer\\instances
         \\demo\\xml\\demo.xml";

      LogConfiguration config = LogConfiguration.getUniqueInstance ();
      config.initialize (fileName, "testClone");

      ECMessageLog.out (MyMessage._ERR_CUSTOMER,
           "MyTestingClass", "main", "Hello");
   ```
   The first three lines of code initialize the WebSphere Commerce logging service. The last line of code instructs ECMessageLog to print out the message MyMessage._ERR_CUSTOMER. `MyTestingClass` and `main` are part of the log trace format. The `Hello` string is placed into the {0} placeholder of the message defined in `ecCustomerMessages.properties` file.
5. Run the class. In the log file, the message trace appears similar to the following:
   ```
   ==============
    TimeStamp:      2000-11-29 16:41:42.5
    Thread ID:     <main>
    Class:          MyTestingClass
    Method:         main
    Severity:       1
    Message Text:   The customer message "Hello".
   ```

You can run a similar test to see the second message.

Execution flow tracing

WebSphere Commerce includes the ECTrace class that is used to trace the execution flow of components running in the WebSphere Commerce Server. The ECTrace class is part of the com.ibm.commerce.ras package.

When creating new business logic, you can insert a trace within your code to trace a method for debugging purposes. Information from the trace is captured in the trace log. You can specify an entry and exit point for the trace. In addition, you can specify that specific data be traced between those two points.

In order to use tracing, it must be enabled for the component in which you would like to run the trace. To enable tracing for a particular component, you can use either the Administration Console or the Configuration Manager.

When tracing customized code, you *must* use the EXTERN component. In the Configuration Manager, this is called *External*.

To set the entry point for a trace within your code, use the following syntax:

```
ECTrace.entry (ECTraceIdentifiers.COMPONENT_EXTERN, myClassName, myMethodName);
```

where *myClassName* is the string representation of the class that contains the traced method. Since this string can be used to trace file parsing, it should contain the fully qualified class name. If the method being traced is static, then an example declaration of myClassName is

```
String myClassName = "com.mycompany.agrouping.MyTracedClass";
```

If the method being traced is not static, then an example declaration of myClassName is

```
String myClassName = this.getClass().getName();
```

To set the trace point to trace data within a method, use the following syntax:

```
ECTrace.trace (ECTraceIdentifiers.COMPONENT_EXTERN, myClassName,
     myMethodName, myText);
```

where *myText* is the text to appear in the trace log.

To set the exit point for a trace within your code, use the following syntax:

```
ECTrace.exit (ECTraceIdentifiers.COMPONENT_EXTERN, myClassName,
     myMethodName);
```

If you need to trace the object returned from the method being traced, then set the exit point as follows:

```
ECTrace.exit (ECTraceIdentifiers.COMPONENT_EXTERN, myClassName,
     myMethodName, returnedObject);
```

where `returnedObject` represents the Java object returned by the method.

Consider an example in which you need to trace the performExecute method of a new controller command called MyNewControllerCmd. The following code snippet shows how to use the ECTrace methods within your performExecute method.

```
public void performExecute() throws ECException {
     ECTrace.entry(ECTraceIdentifiers.COMPONENT_EXTERN,
          this.getClass().getName(), "performExecute");

     super.performExecute();


/////////////////////////////////////////////
//   Some of your business logic           //
/////////////////////////////////////////////


     ECTrace.trace(ECTraceIdentifiers.COMPONENT_EXTERN,
          this.getClass().getName(), "performExecute",
          "My code is great!");

/////////////////////////////////////////////
//   Some more business logic              //
/////////////////////////////////////////////


     ECTrace.exit(ECTraceIdentifiers.COMPONENT_EXTERN,
          this.getClass().getName(), "performExecute");

}
```

When the preceding performExecute method is called, the trace log file captures the following information:

```
==============
TimeStamp:      2000-12-05 17:32:00.257
Thread ID:     <P=502832:O=0:CT>
Component:      EXTERN
Class:          com.mycompany.agrouping.MyNewControllerCmd
Method:         performExecute
Trace:          ENTRY POINT
==============
TimeStamp:     2000-12-05 17:32:00.257
Thread ID:    <P=502832:O=0:CT>
Component:     EXTERN
Class:         com.mycompany.agrouping.MyNewControllerCmd
Method:        performExecute
Trace:         My code is great!
==============
TimeStamp:     2000-12-05 17:32:00.258
Thread ID:    <P=502832:O=0:CT>
```

```
Component:     EXTERN
Class:         com.mycompany.agrouping.MyNewControllerCmd
Method:        performExecute
Trace:         EXIT POINT
```

It is recommended that tracing be used only on major functions. Tracing is not enabled for multiple languages, since it is intended to be used by Store Developers. This is in contrast to messages, which are enabled for multiple languages because system messages are used for administration purposes and user messages are displayed to customers.

JSP template error handling

Error handling for JSP templates can be performed in various ways:

- Error handling from within the page
  For JSP files that require more intricate error handling and recovery, the file can be written to directly handle errors from the data bean. The JSP file can either catch exceptions thrown by the data bean or it can check for error codes set within each data bean, depending on how the data bean was activated. The JSP file can then take an appropriate recovery action based on the error received. Note that a JSP file can use any combination of the following error handling scopes.
- Error JSP at the page level
  A JSP file can also specify its own default error JSP template from an exception occurring within itself through the JSP error tag. This enables a JSP program to specify its own handling of an error. A JSP file which does not specify a JSP error tag will have an error fall through to the application level JSP error template. In the page level error JSP, it must call the JSP helper class (com.ibm.server.JSPHelper) to rollback the current transaction.
- Error JSP at the application level
  An application under WebSphere can specify a default error JSP template when an exception from within any of its servlets or JSP files occur. The application level error JSP template can be used as a mall level or store level (for a single store model) error handler. In the application level error JSP template, a call must be made to the servlet helper class to roll back the current transaction. This is because the Web controller will not be in the execution path to roll back the transaction. Whenever possible, you should rely on the preceding two types of JSP error handling. Use the application level error handling strategy only when required.

Chapter 6. Command implementation

This section provides information about how to write new controller, task, and data bean commands. It also describes how to extend existing controller, task, and data bean commands.

Note: Business This chapter does not describe business policy commands. For information about business policy commands, refer to Chapter 7, "Trading agreements and business policies (Business Edition)" on page 147.

New commands - introduction

The WebSphere Commerce programming model defines four types of commands: controller, task, view and data bean commands. When creating new business logic for your e-commerce application, it is expected that you may need to create new controller, task and data bean commands. You should not need to create new view commands. More information on view commands is found later in this section.

New commands must implement their corresponding interface (which in turn should extend from an existing interface). To simplify command writing, WebSphere Commerce includes an abstract implementation class for each type of command. New commands should extend from these classes.

As an overview, the following table provides information about which implementation class a new command should extend from, and which interface it should implement:

Command type	**Example command name**	**Extends from**	**Implements example interface**
Controller command	MyControllerCmdImpl	com.ibm.commerce. command. ControllerCommandImpl	MyControllerCmd
Task command	MyTaskCmdImpl	com.ibm.commerce. command. TaskCommandImpl	MyTaskCmd
Data bean command	MyDataBeanCmdImpl	com.ibm.commerce. command. DataBeanCommandImpl	MyDataBean

Note: Any spaces in names of implementation classes are for presentation purposes only.

The following diagram illustrates the relationship between the interface and implementation class of a new controller command with the existing abstract implementation class and interface. The abstract class and interface are both found in the com.ibm.commerce.command package.

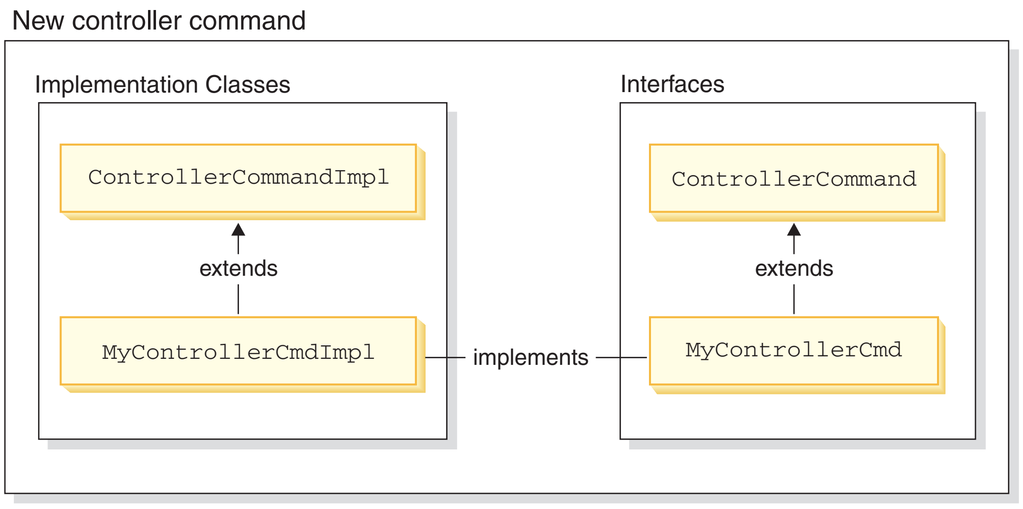


Figure 25.

The following diagram illustrates the relationship between the interface and implementation class of a new task command with the existing abstract implementation class and interface. The abstract class and interface are both found in the com.ibm.commerce.command package.

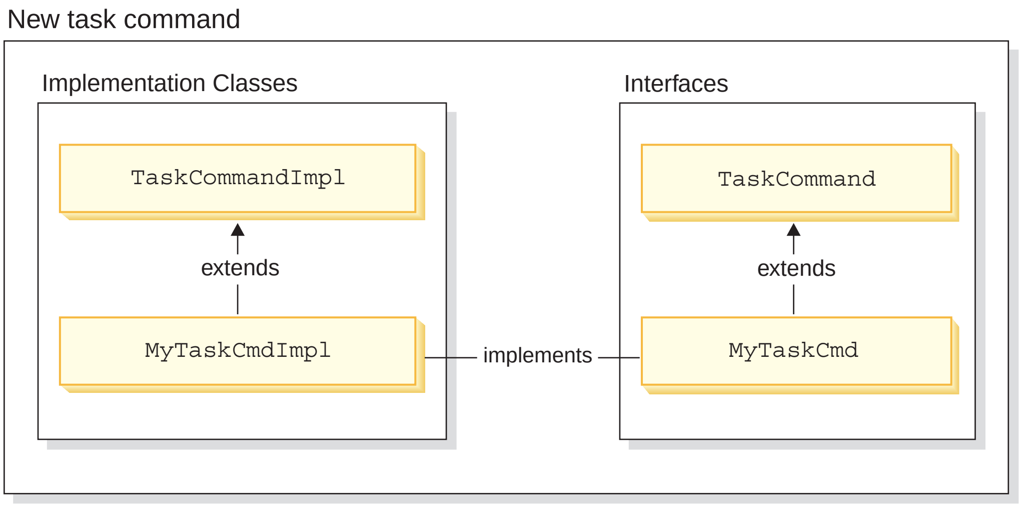


Figure 26.

The following diagram illustrates the relationship between the interface and implementation class of a new data bean command with the existing abstract implementation class and interface. The abstract class and interface are both found in the com.ibm.commerce.command package.

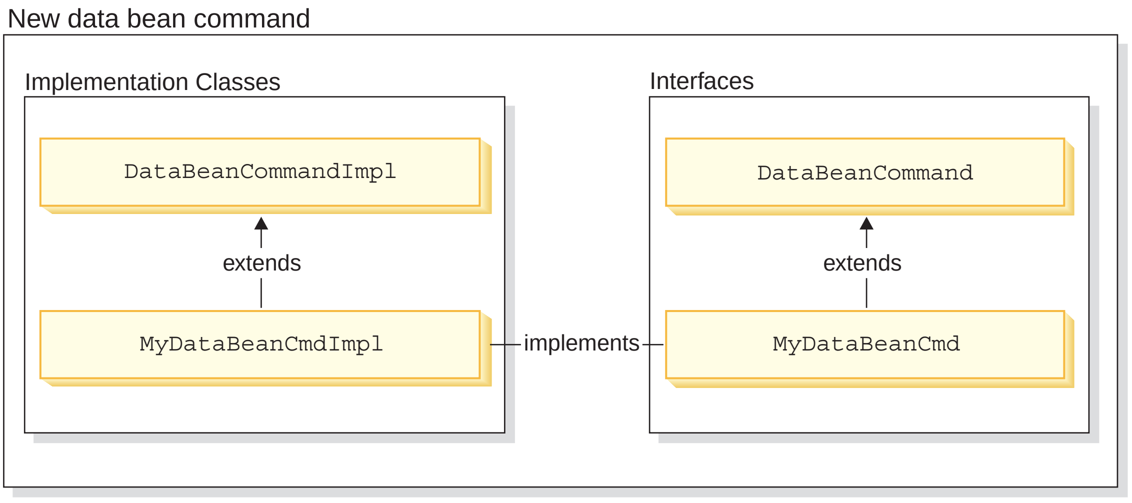


Figure 27.

A view command has two main functions: to format a response and to send the response to the client. A number of generic view commands are provided that send the response to clients, using different protocols. The formatting function is typically handled by the view command invoking a JSP template.

For example, the RedirectViewCommand view command directs the client to a URL to get the response (the response is then formatted by a specified JSP template). The ForwardViewCommand view command forwards the request to the JSP template for formatting and the page is displayed to the client.

Using this view command model, you can create new *views* (the response to the client) by creating new JSP templates. The JSP template should, however, be invoked by one of the existing view commands.

Packaging customized code

When creating customized code, you must follow a particular code organization structure. In general, customized code is maintained in packages and projects that are separate from those included with WebSphere Commerce.

When creating new commands, you must place them in a package named appropriately for your business requirements. That is, if the commands apply to a particular store, package them in a package that is unique to the store. If they apply to more than one store, package them accordingly. For example, you might have the following packages:

- `com.bigbusiness.storeA.commands`
- `com.bigbusiness.storeB.commands`
- `com.bigbusiness.commands`

The preceding packaging structure allows for differentiation between business logic at a store level. In addition, these packages should be stored in a project that is separate from the WebSphere Commerce projects. For example, the preceding packages may be placed in a project called `BigBusinessCustomCode`.

When creating new data beans, they must be kept in a package that is separate from command logic, however, this package can be kept within the project that stores the command packages. From the preceding example, you would then place the `com.bigbusiness.databeans` package within the `BigBusinessCustomCode` project.

When creating new entity beans, they must be stored in a unique project. Therefore, you might have the `BigBusinessCustomEntityBeans` project that contains the `com.bigbusiness.objects` package.

This packaging strategy is required for code deployment purposes.

Command context

Command context is a handle to the Web controller. Commands can obtain information from the Web controller using the command context. Examples of available information include the user's ID, the user object, the language identifier, and the store identifier.

When writing a command, you have access to the command context by calling the getCommandContext() method of the command's superclass. The command context is set to the controller command when the command is invoked by the Web controller. A controller command should propagate the command context to any task or controller commands that are invoked during processing. A command can get the following key information from the command context:

getUserId() and getUser()
Gets the current user ID or user object. The user ID for the current session is saved in a session context. The session context can be persisted in one of two ways: using the WebSphere Commerce cookie or a WebSphere Application Server persistent session object. The command context hides the complexity of session management from a command.

getStoreId(), getStore(), and getStore(storeId)
Gets the store associated with the current request. The Web controller returns the store ID in the URL. If the store ID is not specified in the URL, it can be retrieved from the session object that is saved from the previous request. The WebSphere Commerce run-time environment maintains a set of objects that are frequently accessed. For example, it maintains the set of store objects. A command should always get the store object from the command context to take advantage of the object cache in the Web controller. You can get the current store by calling the getStore() method or get a specific store object by calling the getStore(storeId) method from the command context.

getLanguageId()
Returns the language ID that should be used for the current request. The Web controller implements a Globalization Framework. The concept behind this framework is to determine a language that is preferred by the user and supported by the store. If the URL contains a language ID, the Web controller determines if this language is supported by the store, if so, this is the language ID that gets returned by getLanguageId() method. If no language ID were included in the URL, then the Web controller goes through a decision tree to determine if there is a language ID (that is supported by the store) in the current session object, or in the user's registered preferences, or lastly it will return the default language ID for the store.

getCurrency()
: Returns the currency to be used for the current request. Since currency is part of the Globalization Framework, logic behind this method is similar to the getLanguageId() method.

getCurrentTradingAgreements() and getTradingAgreement(tradingAgreementId)
: Returns the set of trading agreements that are used for the current session. This set may be all of the trading agreements to which the user is entitled, or it can be a subset that was defined by the ContractSetInSession command. A command should always get the trading agreement object from the command context to take advantage of the object cache in the Web controller. You can get the current trading agreement by calling the getCurrentTradingAgreements() method or get a specific trading agreement object by calling the getTradingAgreement(tradingAgreementId) method from the command context.

The command context should be used as a read-only object. You should not call its setter methods. The setter methods are reserved for use by the WebSphere Commerce run-time environment and they may be deprecated in future releases.

For complete details on the command context API (application programming interface), refer to the "Reference" section of the WebSphere Commerce online help.

New controller commands

As previously stated, a new controller command should extend from the abstract controller command class (com.ibm.commerce.command.ControllerCommandImpl). When writing a new controller command, you should override the following methods from the abstract class:

- isGeneric()
- isRetriable()
- setRequestProperties(com.ibm.commerce.datatype.TypedProperty reqParms)
- validateParameters()
- getResources()
- performExecute()

More information on each of the preceding methods is found in the following sections.

isGeneric method

In the standard WebSphere Commerce implementation there are multiple types of users. These include generic, guest, and registered users. Within the grouping of registered users there are customers and administrators.

The generic user has a common user ID that is used across the entire system. This common user ID supports general browsing on the site in a manner that minimizes system resource usage. It is more efficient to use this common user ID for general browsing, since the Web controller does not need to retrieve a user object for commands that can be invoked by the generic user.

The isGeneric method returns a boolean value which specifies whether or not the command can be invoked by the generic user. The isGeneric method of a controller command's superclass sets the value to `false` (meaning that the invoker must be either a registered user or a guest user). If your new controller command can be invoked by generic users, override this method to return `true`.

You should override this method to return `true` if your new command does not fetch or create resources associated with a user. An example of a command that can be invoked by a generic user is the ProductDisplay command. It is sensible to allow any user to be able to view products. An example of a command for which a user must be either a guest or registered user (and hence, isGeneric returns `false`) is the OrderItemAdd command.

When isGeneric returns a value of `true`, the Web controller does not create a new user object for the current session. As such, commands that can be invoked by the generic user run faster, since the Web controller does not need to retrieve a user object.

The syntax for using this method to enable generic users to invoke a command is as follows:

```
public boolean isGeneric()
{
    return true;
}
```

isRetriable method

The isRetriable method returns a boolean value which specifies whether or not the command can be retried on a transaction rollback exception. The isRetriable method of the new controller command's superclass returns a value of `false`. You should override this method and return a value of `true`, if your command can be retried on a transaction rollback exception.

An example of a command that should not be retried in the case of a transaction exception is the OrderProcess command. This command invokes the third party payment authorization process. It cannot be retried, since that authorization cannot be reversed. An example of a command that can be retried is the ProductDisplay command.

The syntax for enabling the command to be retried in the case of a transaction rollback exception is as follows:

```
public boolean isRetriable()
{
   return true;
}
```

setRequestProperties method

The setRequestProperties method is invoked by the Web controller to pass all input properties to the controller command. The controller command must parse the input properties and set each individual property explicitly within this method. This explicit setting of properties by the controller command itself promotes the concept of type safe properties.

The syntax for using this method is as follows:

```
public void setRequestProperties(
   com.ibm.commerce.datatype.TypedProperty reqParms)
{

     // parse the input properties and explicitly set each parameter

}
```

validateParameters method

The validateParameters method is used to do initial parameter checking and any necessary resolution of parameters. For example, it could be used to resolve `orderId=*`. This method is called before both the getResources and performExecute methods. Refer to “Access control interactions” on page 98 for more details about this sequence.

getResources method

This method is used to implement resource-level access control. It returns a vector of resource-action pairs upon which the command intends to act. If nothing is returned, no resource-level access control is performed. For more information about access control, refer to Chapter 4, “Access control” on page 85.

performExecute method

The performExecute method contains the business logic for your command. It should invoke the performExecute method of the command’s superclass before any new business logic is executed. At the end, it must return a view name.

The following shows example syntax for the performExecute method in a new controller command. In this case, the response uses a redirect view command, it could, however, use a forward view command or direct view command:

```
public void performExecute() throws ECException
{
     super.performExecute();

/////////////////////////////////////////////////
//  your business logic                        //
/////////////////////////////////////////////////

// Create a new TypedProperty for response properties.
TypedProperty rspProp = new TypedProperty();

// set response properties
rspProp.put(ECConstants.EC_VIEWTASKNAME, "MyView");
/////////////////////////////////////////////////////
//  The following line is optional.  The VIEWREG   //
//  table can specify the redirect URL.            //
/////////////////////////////////////////////////////

rspProp.put(ECConstants.EC_REDIRECTURL, MyURL);

////////////////////////////////////////////////////////////////////////////
//   If you are using a forward view, you can set the                     //
//   response properties as follows:                                      //
//   TypedProperty rspProp = new TypedProperty();                         //
//   rspProp.put(ECConstants.EC_VIEWTASKNAME, "MyView");                  //
//   rspProp.put(ECConstants.EC_DOCPATHNAME, "MyJSP.jsp");                //
//                                                                        //
//   Again, it is optional to explicitly set the name of the JSP template.//
//   The VIEWREG table can specify the JSP template.                      //
////////////////////////////////////////////////////////////////////////////

setResponseProperties(rspProp);
}
```

If you specify the redirect URL within the performExecute method and an entry exists in the VIEWREG table, the value specified in the code takes precedence over the value in the VIEWREG table. The same order of precedence holds true for specification of a JSP template within code.

Long-running controller commands

If a controller command takes a long time to execute, you can split the command into two commands. The first command, which is executed as the result of a URL request, simply adds the second command to the Scheduler, so that it runs as a background job. This is illustrated in the following diagram:

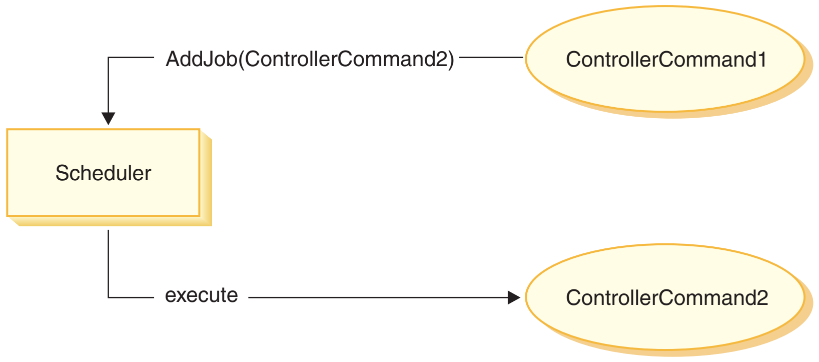


Figure 28.

The flow shown in the preceding diagram is as follows:

1. ControllerCommand1 is executed as a result of a URL request.
2. ControllerCommand1 adds a job to the Scheduler. The job is ControllerCommand2. ControllerCommand1 returns a view, immediately after adding the job to the Scheduler.
3. The Scheduler executes ControllerCommand2 as a background job.

In this scenario, the client typically polls the result from ControllerCommand2. ControllerCommand2 should write the job state to the database.

Formatting of input properties to view commands

When a controller command completes, it returns the name of a view that should be executed. This view may require that several input properties get passed to it. There can be three sources for these input parameters, as described in the following list:

- Default properties that are stored in the PROPERTIES column of the CMDREG table
- Default properties from the PROPERTIES column of the VIEWREG table
- Input properties from the URL

For more information about how these properties are merged and set in the attributes for the JSP template, refer to “Setting JSP attributes - overview” on page 41. This section describes how the input properties to a view command may be formatted.

For redirect view commands, two topics are examined:

- Flattening a query string to support URL redirection
- Dealing with a limit on the length of the redirect URL

For forward view commands, the topic of enumeration of input parameters and setting them as attributes in the HttpServletRequestObject is examined.

Flattening input parameters into a query string for HttpRedirectView

All input parameters that are passed to a redirect view command are flattened into a query string for URL redirection. For example, suppose that the input to the redirect view command contains the following properties:

```
URL = "MyView?p1=v1&p2=v2";
ip1 = "iv1";   // input to orginal controller command
ip2 = "iv2" ;  // input to original controller command
op1 = "ov1";
op2 = "ov2";
```

Based upon the preceding input parameters, the final URL is

```
MyView?p1=v1&p2=v2&ip1=iv1&ip2=iv2&op1=ov1&op2=ov2
```

Note that if the command is to use SSL, then the parameters are encrypted and the final URL appears as

```
MyView?krypto=encrypted_value_of"p1=v1&p2=v2&ip1=iv1&ip2=iv2&op1=ov1&op2=ov2"
```

Handling a limited length redirect URL

By default, all input parameters to the controller command are propagated to the redirect view command. If there is a limit on the number of characters in the redirect URL, this may cause a problem. An example of when the length may be limited is if the client is using the Internet Explorer browser. For this browser, the URL cannot exceed 2083 bytes. If the URL does exceed this limit, the URL gets truncated. As such, you can encounter a problem if there are a large number of input parameters, or if you are using encryption, because an encryption string is typically two to three times longer than an unencrypted string.

There are two approaches for handling a limited length redirect URL:

1. Override the getViewInputProperties method in the controller command to return only the sets of parameters that are required to be passed to the redirect view command.
2. Use a specified special character in the URL parameters to indicate which parameters can be removed from the input parameter string.

To demonstrate each of the preceding approaches, consider the following set of input parameters to the controller command:

```
URL="MyView";
// All of the following are inputs to the original controller command.
ip1="ipv1";
ip2="ipv2";
ip3="ipv3";
iq1="iqv1";
```

```
iq2="iqv2";
ir1="ipr1";
ir2="ipr2";
is="isv";
```

If you are overriding the getViewInputProperties method, the new method can be written so that only the following parameters are passed to the view command:

```
ir2="ipr2";
is="isv";
```

Using the second approach, the view command can be invoked using special parameters to indicate that certain input parameters should be removed. For example, you can achieve the same result by specifying the following as the URL parameter:

```
URL="MyView?ip*=&iq*=&ir1="
```

This URL parameter instructs the WebSphere Commerce run-time framework of the following:

- The `ip*=` specification means that all parameters whose names start with `ip` should be removed.
- The `iq*=` specification means that all parameters whose names start with `iq` should be removed.
- The `ir1=` specification means that the `ir1` parameter should be removed.

Setting attributes in the HttpServletRequest object for HttpForwardView

The default HttpForwardViewCommandImpl enumerates all of the parameters passed to the command and sets them as attributes in the HttpServletRequest object.

For example, suppose that the requestProperties object passed to the forward view command contains the following parameters:

```
p1="pv1";
p2="pv2";
p3=pv3;   // pv3 is an object
```

Then the following attributes are passed to the JSP template using the request.setAttribute() method.

```
request.setAttribute("p1", "pv1");
request.setAttribute("p2", "pv2");
request.setAttribute("p1", pv1);
request.setAttribute("RequestProperties", requestProperties);
request.setAttribute("CommandContext", commandContext);
```

where `requestProperties` is the TypedProperty object that is passed to the command, `commandContext` is the command context object that is passed to the command, and `p1`, `p2`, and `p3` are parameters defined in the requestProperties object.

Database commits and rollbacks for controller commands

Throughout the execution of a controller command, data is often created or updated. In many cases, the database must be updated with the new information at the end of the transaction. The transaction is managed by the Web controller.

The Web controller marks the beginning of the transaction before calling the controller command. When the execution of the controller command is complete, the controller command returns a view name to the Web controller. The Web controller is responsible for marking the end of the transaction. The actual point at which the transaction ends (before or after invoking the view) is dependent upon the type of view used.

There are three types of view commands:
- Forward view command
- Redirect view command
- Direct view command

The Web controller determines the view command to be used for the view, by looking up the view name in the VIEWREG table.

If the entry in the VIEWREG table specifies the use of the ForwardViewCommand, then the Web controller forwards the results of the controller command to the corresponding ForwardViewCommand implementation class (also specified in the VIEWREG). The view command executes within the context of the current transaction. In this case, the database commit or rollback does not occur until the view command completes.

If the entry in the VIEWREG table specifies the use of the RedirectViewCommand, then the Web controller forwards the results of the controller command to the corresponding RedirectViewCommand implementation class. The view command then operates outside of the scope of the current transaction and the database commit or rollback occurs before the redirected view command is called.

If the entry in the VIEWREG table specifies the use of the DirectViewCommand, then the Web controller forwards the results of the controller command to the corresponding DirectViewCommand implementation class. The view command executes within the context of the current transaction. In this case, the database commit or rollback does not occur until the view command completes. (Note that ForwardViewCommand and DirectViewCommand are similar. The ForwardViewCommand forwards the results to a JSP template. In contrast, the DirectViewCommand receives the results as input stream and passes them on as an output stream. It uses either the getRawDocument method that treats the data as bytes, or the getTextDocument that treats the data as text.)

In the cases where the view command executes under the same transaction scope as the controller command, an error in the view command causes a rollback of the entire transaction. This may or may not be the desired outcome, depending upon your business logic.

Example of transaction scope with a controller command

To illustrate the differences in transaction scope for a controller command, depending upon the type of view command used, consider the following examples.

Case 1: Executing the view within the scope of the controller command transaction

Suppose that you have created a new controller command called `YourControllerCmdA`. The command's performExecute method would then include the following:

```
.
.
// Create a new TypedProperty object for output.
TypedProperty rspProp = new TypedProperty();

//////////////////////
// Business logic   //
//////////////////////

// Return the view
rspProp.put(ECConstants.EC_VIEWTASKNAME, "YourView");
SetResponseProperties(rspProp);
```

In the preceding code snippet, the controller command returns "YourView" as the view. YourView is registered in the VIEWREG table. The following is an example insert statement to register YourView.

```
insert into VIEWREG (ViewName, DeviceFmt_id, storeEnt_id, interfacename,
classname, properties)

values ('YourView',  -1, XX,'com.ibm.commerce.command.ForwardViewCommand',
'com.ibm.commerce.command.HttpForwardViewCommandImpl','docname=YourView.jsp');
```

where XX is the store identifier. Since the view uses the `com.ibm.commerce.command.HttpForwardViewCommandImpl` implementation class, the Web controller uses the generic forward view command.

Based upon the preceding command registration, the Web controller launches the `YourView.jsp` file within the scope of the controller command transaction. If an error occurs in `YourView.jsp`, the transaction fails and a database rollback occurs. As a result, the entire controller command fails.

Case 2: Executing the view outside of the scope of the controller command transaction

Suppose that you would prefer to have information committed to the database, even in the case when an error may occur in the view. In order to have the view execute outside the scope of the controller command's transaction, the view must be executed as a redirect.

To execute the view as a redirect, the performExecute method of the controller command returns the view in the following manner:

```
.
.
// Create a new TypedProperty object for output.
TypedProperty rspProp = new TypedProperty();

//////////////////////
// Business logic   //
//////////////////////

// Return the view
rspProp.put(ECConstants.EC_VIEWTASKNAME, EC_GENERIC_REDIRECTVIEW);
rspProp.put(EC_Constants.EC_REDIRECTURL, "YourView2");
```

The following example SQL statement supports the redirect strategy:

```
insert into VIEWREG (ViewName, DeviceFmt_id, storeEnt_id, interfacename,
classname, properties)

values ('YourView2',  -1, XX,'com.ibm.commerce.command.ForwardViewCommand',
'com.ibm.commerce.command.HttpForwardViewCommandImpl','docname=YourView2.jsp');
```

where `XX` is the store identifier.

Since the command passes the `EC_GENERIC_REDIRECTVIEW` value as a response property parameter, the Web controller uses the generic redirect view command. The generic redirect view is registered in the VIEWREG table with the following information:

- ViewName = RedirectView
- DeviceFmt_Id = -1
- InterfaceName = com.ibm.commerce.command.RedirectViewCommand
- ClassName = com.ibm.commerce.command.HttpRedirectViewCommandImpl

The Web controller invokes the generic redirect view command, which takes the redirect URL as an input property. The response is the redirected to the redirect URL. After the redirect occurs, the YourView2 is invoked. This is implemented as a generic forward view.

New task commands

A new task command should extend from the abstract task command class (com.ibm.commerce.command.TaskCommandImpl) and implement an interface that extends the com.ibm.commerce.TaskCommand interface. As shown in the diagram on page 125, the new task command should be defined as follows:

```
public class MyTaskCmdImpl extends com.ibm.commerce.command.TaskCommandImpl
     implements MyTaskCmd {

}
```

All the input and output properties for the task command must be defined in the command interface, for example MyTaskCmd. The caller programs to the task command interface, rather than the task command implementation class. This enables you to have multiple implementations of the task command (one for each store), without the caller being concerned about which implementation class to call.

All the methods defined in the interface must be implemented in the implementation class. Since the command context should be set by the caller (a controller command), the task command does not need to set the command context. The task command can, however, obtain information from the Web controller by using the command context.

In addition to implementing the methods defined in the task command interface, you should override the performExecute method from the com.ibm.commerce.command.TaskCommandImpl class.

The performExecute method contains the business logic for the particular unit of work that the task command performs. It should invoke the performExecute method of the task command’s superclass, before performing any business logic. The following code snippet shows an example performExecute method for a task command.

```
public void performExecute() throws ECException
{
     super.performExecute();

     // Include your business logic here.
     // Set output properties so the controller command
     // can retrieve the result from this task command.
}
```

The run-time framework calls the getResources method of the controller command to determine which protectable resources the command will access. It may be the case that a task command is executed during the scope of a controller command and it attempts to access resources that were not returned by the getResources method of the controller command. If this is the case, the task command itself can implement a getResources method to ensure that access control is provided for protectable resources.

Note that by default, getResources returns `null` for a task command and resource-level access control checking is not performed. Therefore, you must override this if the task command accesses protectable resources.

Customization of existing commands

This section describes the various ways in which you can customize existing controller, task and data bean commands.

Customizing existing controller commands

A controller command encapsulates the business logic for a business process. Individual units of work within the business process may be performed by task commands. As such, there are several ways in which a controller command can be customized, some of which involve customizing task commands.

When customizing a controller command, you can accomplish the following:

- Add additional processing and logic to an existing controller command. This can be added before existing business logic, after existing logic, or both before and after.
- Replace one or more task commands. This allows you to modify how a particular step in the business process is performed.
- Replace the view called by the controller command.

The following sections provide details on how to make the preceding modifications.

Adding new business logic to a controller command

Suppose there is an existing WebSphere Commerce controller command, called `ExistingControllerCmd`. Following the WebSphere Commerce naming conventions, this controller command would have an interface class named `ExistingControllerCmd` and an implementation class named `ExistingControllerCmdImpl`. Now assume that a business requirement arises and you must add new business logic to this existing command. One portion of the logic must be executed before the existing command logic and another portion must be executed after the existing command logic.

The first step in adding the new business logic is to create a new implementation class that extends the original implementation class. In this example, you would create a new `ModifiedControllerCmdImpl` class that extends the `ExistingControllerCmdImpl` class. The new implementation class should implement the original interface (`ExistingControllerCmd`).

In the new implementation class you must create a new `performExecute` method to override the `performExecute` of the existing command. Within the new `performExecute` method, there are two ways in which you can insert your new business logic: you can either include the code directly in the controller command, or you can create a new task command to perform the new business logic. If you create a new task command then you must instantiate the new task command object from within the controller command.

The following code snippet demonstrates how to add new business logic to the beginning and end of an existing controller command by including the logic directly in the controller command:

```
public class ModifiedControllerCmdImpl extends ExistingControllerCmdImpl
   implements ExistingControllerCmd
   {
      public void performExecute ()
         throws com.ibm.commerce.exception.ECException
         {

            /* Insert new business logic that must be
               executed before the original command.
            */

            // Execute the original command logic.
            super.performExecute();

            /* Insert new business logic that must be
               executed after the original command.
            */
          }
    }
```

The following code snippet demonstrates how to add new business logic to the beginning of an existing controller command by instantiating a new task command from within the controller command. In addition, you would also create the new task command interface and implementation class and register the task command in the command registry.

```
// Import the package with the CommandFactory
import com.ibm.commerce.command.*;

public class ModifiedControllerCmdImpl extends ExistingControllerCmdImpl
```

```
   implements ExistingControllerCmd
   {

      public void performExecute ()
         throws com.ibm.commerce.exception.ECException
         {
            MyNewTaskCmd cmd = null;
            cmd = (MyNewTaskCmd) CommandFactory.createCommand(
               "com.mycompany.mycommands.MyNewTaskCommand",
               getStoreId());

            /*
            Set task command's input parameters, call its
            execute method and retrieve output
            parameters, as required.
            */

            super.performExecute();
         }
   }
```

Regardless of whether you include the new business logic in the controller command, or create a task command to perform the logic, you must also update the CMDREG table in the WebSphere Commerce command registry to associate the new controller command implementation class with the existing controller command interface. The following SQL statement shows an example update:

```
update CMDREG
set CLASSNAME='ModifiedControllerCmdImpl'
where INTERFACENAME='ExistingControllerCmd'
```

Replacing task commands called by a controller command

A controller command often calls several task commands that perform individual tasks. Collectively, these tasks make up the business process represented by the controller command. You may need to change the way in which a particular step in the process is performed, rather than adding new business logic to the beginning or end of the controller command. In this case, you must replace the instantiation of the task command that you wish to override, with the instantiation of a new task command that performs the task in your desired manner.

As a result of the design of the WebSphere Commerce programming model, you do not need to create a new controller command implementation class to replace the task command. The controller command instantiates the task command by calling the command factory’s `createCommand` method. The command factory uses the task command’s interface name and then determines the correct implementation class, based upon the command registry. As such, to replace the task command that gets instantiated, you must create a new task command implementation class and then update the command registry so that the original task command interface name is associated with the new task command implementation class. Refer to “Customizing existing task commands” on page 143 for more information.

Replacing the view called by a controller command

To replace the view that is called by a controller command, you create a new implementation class for the controller command. For example, create a new `ModifiedControllerCmdImpl` that extends `ExistingControllerCmdImpl` and implements the `ExistingControllerCmd` interface.

Within the `ModifiedControllerCmdImpl` class, override the `performExecute` method. In the new `performExecute` method, call `super.performExecute` to ensure that all command processing occurs. After the command logic is executed, you can use the response properties to override the view called. The following code snippet displays how to override the view when the view is executed as a redirect:

```
// Import the packages containing TypedProperty, and ECConstants.
import com.ibm.commerce.datatype.*;
import com.ibm.commerce.server.*;

public class ModifiedControllerCmdImplImpl extends ExistingControllerCmdImpl
   implements ExistingControllerCmd
   {
      public void performExecute ()
         throws com.ibm.commerce.exception.ECException
         {

            // Execute the original command logic.
            super.performExecute();

            // Create a new TypedProperty for response properties.
            TypedProperty rspProp = new TypedProperty();

           // set response properties
           rspProp.put(ECConstants.EC_VIEWTASKNAME, "MyView");
           /////////////////////////////////////////////////////
          //  The following line is optional.  The VIEWREG   //
          //  table can specify the redirect URL.            //
          /////////////////////////////////////////////////////

          rspProp.put(ECConstants.EC_REDIRECTURL, MyURL);

          setResponseProperties(rspProp);

         }
   }
```

The following code snippet displays how to override the view when the view is executed as a forward view:

```
// Import the packages containing TypedProperty, and ECConstants.
import com.ibm.commerce.datatype.*;
import com.ibm.commerce.server.*;
```

```
public class ModifiedControllerCmdImplImpl extends ExistingControllerCmdImpl
   implements ExistingControllerCmd
   {
      public void performExecute ()
         throws com.ibm.commerce.exception.ECException
         {

            // Execute the original command logic.
            super.performExecute();

            // Create a new TypedProperty for response properties.
            TypedProperty rspProp = new TypedProperty();

           // set response properties
           rspProp.put(ECConstants.EC_VIEWTASKNAME, "MyView");

           ////////////////////////////////////////////////
          //  It is optional to explicitly set the name   //
          //  of the JSP template. The VIEWREG table can  //
          //  specify the JSP template.                   //
          /////////////////////////////////////////////////

          rspProp.put(ECConstants.EC_DOCPATHNAME, "MyJSP.jsp");

          setResponseProperties(rspProp);

         }
    }
```

To determine which view is used by an existing controller command, refer to the Reference section of the WebSphere Commerce online help.

Customizing existing task commands

There are two standard ways to modify existing WebSphere Commerce task commands. With these methods of modification, you can accomplish the following:

- Add additional processing and logic to an existing task command. This can be added before existing business logic, after existing logic, or both before and after.
- Completely replace the existing business logic with your own business logic.

To accomplish the above modifications, you actually create a new task command implementation class. More detail is provided in the following sections.

Adding new business logic to a task command

Suppose there is an existing WebSphere Commerce task command, called `ExistingTaskCmd`. Following the WebSphere Commerce naming conventions, this task command would have an interface class named `ExistingTaskCmd` and an implementation class named ExistingTaskCmdImpl. Now assume that a business requirement arises and you must add new business logic to this existing command. One portion of the logic must be executed before the existing command logic and another portion must be executed after the existing command logic.

The first step in adding the new business logic is to create a new implementation class that extends the original implementation class. In this example, you would create a new ModifiedTaskCmdImpl class that extends the ExistingTaskCmdImpl class. The new implementation class should implement the original interface (ExistingTaskCmd).

Within the new command, you override the existing performExecute method and include the new logic before and after calling the super.performExecute method.

The following pseudo-code demonstrates how to add new business logic to an existing task command:

```
public class ModifiedTaskCmdImpl extends ExistingTaskCmdImpl
     implements ExistingTaskCmd {


         /* Insert new business logic that must be
            executed before the original command.
         */

         // Execute the original command logic.
         super.performExecute();

         /* Insert new business logic that must be
            executed after the original command.
         */
}
```

You must also update the CMDREG table to associate the new implementation class with the existing interface. The following SQL statement shows an example update:

```
update CMDREG
set CLASSNAME='ModifiedTaskCmdImpl'
where INTERFACENAME='ExistingTaskCmd'
```

Replacing business logic of an existing task command

To replace the business logic of an existing task command, you must create a new implementation class for the task command. This new implementation class must extend from the existing task command but it should not implement the existing interface. Additionally, in the new implementation class, do not call the performExecute method of the superclass.

While extending from the exact command that you are replacing may seem counterintuitive, the reason for taking this approach is related to support for future versions of WebSphere Commerce. This approach shields your code from changes that may be made to command interfaces in future versions of WebSphere Commerce.

As an example, suppose you wanted to replace the business logic of the `OrderNotifyCmdImpl` task command. In this case, you would create a new task command called `CustomizedOrderNotifyCmdImpl`. This command extends `OrderNotifyCmdImpl`. In the new `CustomizedOrderNotifyCmdImpl`, you create the new business logic, but do not call the `performExecute` method from the superclass. If a future version of WebSphere Commerce then introduces a new method, called `newMethod` in the interface, the corresponding version of the `OrderNotifyCmdImpl` command will include a default implementation of the `newMethod` method. Then, since your new command extends from `OrderNotifyCmdImpl`, the compiler will find the default implementation of this new method in the `OrderNotifyCmdImpl` command and your new command is shielded from the interface change.

Refer to the Reference section of the WebSphere Commerce online help to ensure that the new implementation class provides the same external behavior as the existing class.

Data bean customization

A data bean normally extends an access bean. The access bean, which can be generated by VisualAge for Java, provides a simple way to access information from an entity bean. When modifications are made to an entity bean (for example, adding a new field, a new business method or a new finder), the update is reflected in the access bean as soon as the access bean is regenerated. Since the data bean extends the access bean, it automatically inherits the new attributes. As a result of this relationship, no coding is required to enable the data bean to use new attributes from the entity bean.

If you need to add new attributes to a data bean that are not derived from an entity bean, you can extend the existing data bean using Java inheritance. For example, if you want to add a new field to the OrderDataBean, define MyOrderDataBean as follows:

```
public class MyOrderDataBean extends OrderDataBean
{
     public String myNewField () {
     //  implement the new field here
     }

}
```

The new data bean must also have a BeanInfo class. A sample of the declaration for this class follows:

```
public class MyOrderDataBeanInfo extends java.beans.SimpleBeanInfo
{

}
```

VisualAge for Java provides a tool that allows you to generate this BeanInfo class.

Chapter 7. Trading agreements and business policies (Business Edition)

This chapter only applies to WebSphere Commerce Business Edition.

Introduction

One of the key elements of B2B (business-to-business) commerce is relationship management. A trading agreement is used to manage a business relationship between a buyer and a seller organization. The trading agreement model used by WebSphere Commerce Business Edition supports various types of trading agreements, such as Contract and RFQ (request for quote).

The main element of a trading agreement is a set of terms and conditions. Each term and condition defines a specific business rule to be used during trading. Using WebSphere Commerce Business Edition, a set of terms and conditions can be negotiated using an RFQ online process, or negotiated offline and then captured using the business relationship management interfaces in the WebSphere Commerce Accelerator.

There are several ways to model a term and condition:

- A term and condition that selects one of the predefined business policies, such as a Price list and a Return policy. Or it can select a business policy that you have created. One term and condition object can also refer to multiple business policy objects.
- A term and condition that applies a specific adjustment to the business policy, such as an adjustment to the standard pricing.
- A term and condition that defines a set of parameters that govern a business process. For example, it could specify that a particular fulfillment center is to be used by a specific contract.

A contract is made up of a set of terms and conditions. This is shown in the following diagram.

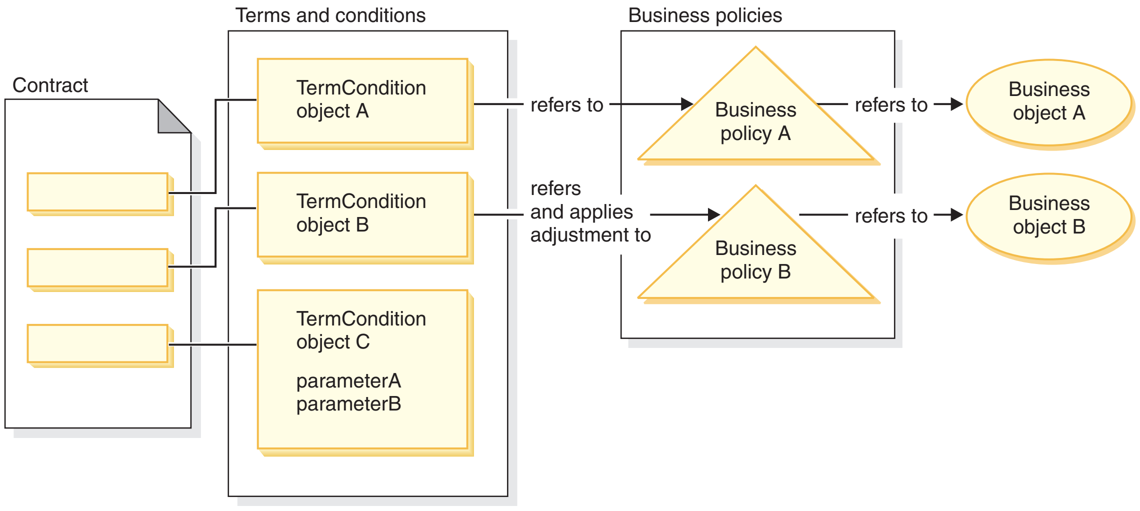


Figure 29.

From the preceding diagram, note the following:

- The term "adjustment" refers to a modification to the business policy. As an example, it can be used to apply a discount to the result of a business policy such that a 10% discount gets applied to the standard price. It can also be used to influence the business policy with a set of parameters.
- As an example, the TermCondition object A may represent a shipping term and condition object. In this case, the business policy A may represent a shipping mode business policy and the business object A represents the shipping mode "A3" of shipping carrier XYZ.
- An another example, the TermCondition object B may represent a price term and condition object that applies a 50% discount of the price defined by business policy B. In this case, business policy B is a price policy and business object B is a trading position container that defines the trading position for the master category.

This chapter provides guidelines for programmers on how to create new business policies and new terms and conditions.

The ToolTech sample store demonstrates a shipping term and condition object and a price term and condition object in its business flow. For more information about the contract data that supports these examples, refer to "ToolTech sample contract data" on page 150.

Business policy objects and commands

A business policy object contains the following information:

- Policy ID
  This is the primary key for the business policy object.
- Policy type
  This defines the business policy type. `Price` and `ProductSet` are examples of policy types.
- Policy name
  Each business policy must have a unique name.
- Store entity
  The store or store group in which the business policy is deployed.
- Properties
  A set of default properties that can be passed to the business policy command. The commands associated with the business policy object are stored in the BusinessPolicyCmd table.
- Effective period
  The period for which the business policy object is effective.
- Business policy command
  Zero or more business policy commands that implement the business policy. A business policy command is typically invoked by a business process that can be either a task command or a controller command. For example, the `getContractPrice()` command gets the price term and condition. This price term and condition refers to a particular price policy command and this price policy command is used used to calculate the price.

Multiple business policy commands can be associated with a single business policy object. Each business policy command must implement the same interface defined by the business policy type object. The structure of a new business policy command is depicted in the following diagram:

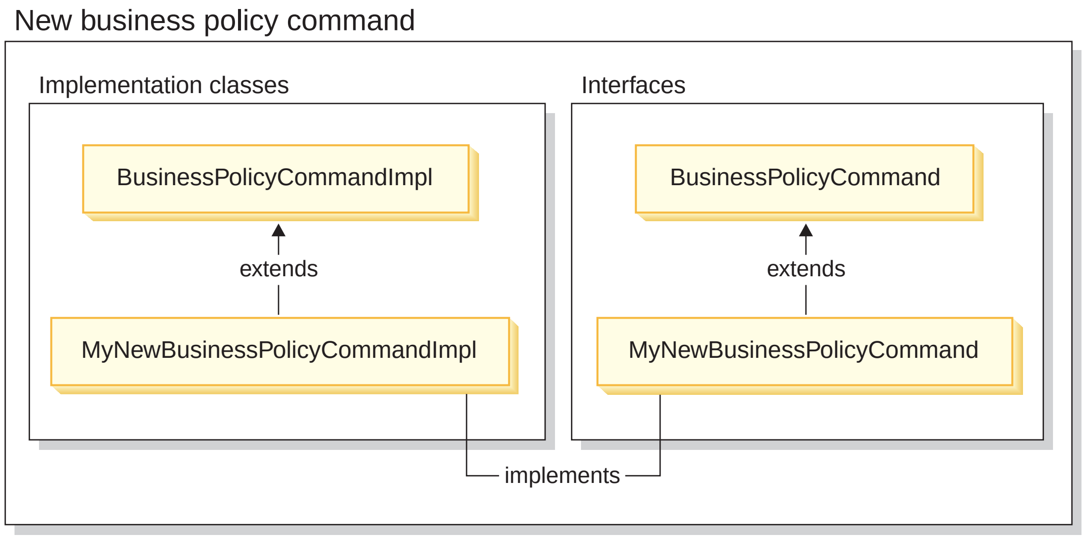

New business policy command
Implementation classes
BusinessPolicyCommandImpl
extends
MyNewBusinessPolicyCommandImpl
Interfaces
BusinessPolicyCommand
extends
MyNewBusinessPolicyCommand
implements

Figure 30.

As shown in the preceding diagram, in order to create a new business policy command, you create a new implementation class that extends the WebSphere Commerce BusinessPolicyCmdImpl implementation class. You also create a new interface that extends the BusinessPolicyCmd interface.

ToolTech sample contract data

This section provides an introduction to some of the contract data that is used in the ToolTech sample store.

The sample data in the following sections is organized by database table. Only the relevant rows and columns are displayed. Also note that when the sample is installed any unique identifiers (such as CONTRACT_ID) may have different values than what is shown here.

CONTRACT table sample data

The following table shows relevant sample data from the ToolTech CONTRACT database table. Note that for display purposes, the database column headings are showin the first column and the row of sample data from the table is shown in the second column.

Column name	Sample data
CONTRACT_ID	10007
MAJORVERSION	1
MINORVERSION	0

Column name	Sample data
NAME	ToolTechContractNumber 4567
MEMBER_ID	-2001
ORIGIN	0
STATE	3
USAGE	1
MARKFORDELETE	0

TERMCOND table sample data

The following table shows relevant sample data from the ToolTech TERMCOND database table. Note that for display purposes, the database column headings are shown the first column and the rows of sample data from the table are shown in the second and third columns.

Column name	Sample data row 1	Sample data row 2
TERMCOND_ID	10025	10030
TCSUBTYPE_ID	PriceTCPriceListWith SelectiveAdjustment	ShippingTCShippingMode
TRADING_ID	10007	10007
STRINGFILED1	ProductSet2	
INTEGERFIELD2	10002	
INTEGERFIELD3	1	
BIGINTFIELD1	10051	
FLOATFIELD1	-50.0	
SEQUENCE	1	6

POLICYTC table sample data

The following table shows relevant sample data from the ToolTech POLICYTC database table. This table establishishes the relationship between a policy and a terms and conditions object.

	Column name	
	POLICY_ID	TERMCOND_ID
Sample data row 1	10053	10025
Sample data row 2	10056	10030

POLICY table sample data

The following table shows relevant sample data from the ToolTech POLICY database table.

Column name	Sample data row 1	Sample data row 2
POLICY_ID	10053	10056
POLICYNAME	MasterCatalogPriceList	A3
POLICYTYPE_ID	Price	ShippingMode
STOREENT_ID	10051	10051
PROPERTIES	name=ToolTech& member_id=-2001	shippingMode=A3
STARTTIME	null	null
ENDTIME	null	null

TRADEPOSCN table sample data

The following table shows relevant sample data from the ToolTech TRADEPOSCN database table.

	Column name			
	READEPOSCN_ID	MEMBER_ID	NAME	TYPE
Sample data row	10051	-2001	ToolTech	S

SHIPMODE table sample data

The following table shows relevant sample data from the ToolTech SHIPMODE database table.

	Column name			
	SHIPMODE_ID	STOREENTITY_ID	CODE	CARRIER
Sample data row	10053	10051	A3	XYZ Carrier

Extending the existing contract model

A contract can be made up of one or more terms and conditions objects, each of which refers to a policy. As such, the subsequent sections describe the steps necessary for creating a new business policy and integrating this into your business flow.

As a brief overview, the following are the high-level steps for performing this task:

1. Creating a new business policy.
   The following tasks are related to creating a new business policy command:
   a. Create a new business policy type (if required).
      Several business policy types are provided, but if the standard types do not suit your business requirements, create a new business policy type.
   b. Creating a new business policy command.
   c. Register the new business policy and business policy command.
2. Relate a terms and conditions object to the new business policy
   This can be done by relating an existing terms and conditions object to the new business policy, or by creating a new terms and condition object. If you create a new terms and conditions object, then you must perform the following steps:
   a. Register the new term and condition in the database
   b. Register the new term and condition in the contract DTD (document type definition)
   c. Creating a new CMP enterprise bean for the term and condition
   d. Updating the WebSphere Commerce Accelerator to reflect the new term and condition
3. Invoking the new business policy during the business flow.

Creating a new business policy

Creating a new business policy typically involves registering a unique business policy in the database, as well as creating a new business policy command.

Creating a new business policy command involves the following high-level steps:
1. Creating a new business policy type (if required).
2. Writing the new business policy command.
3. Registering the new business policy and business policy command in the database.

Each of the preceding steps is described in more detail in the subsequent sections.

Creating a new business policy type

This section describes how to create a new business policy type. A business policy type indicates the realm of the transaction to which a policy applies. Examples of business policy types include:
- Price

- ProductSet
- ShippingMode
- ShippingCharge
- Payment
- ReturnCharge
- ReturnApproval
- ReturnPayment
- InvoiceFormat

If the existing business policy types do not satisfy your business requirements, you should create a new business policy type. Creating the new business policy type consists of defining and registering the business policy type.

When defining and registering a new policy type, you must update the following database tables:
- POLICYTYPE
- PLCYTYCMIF
- PLCYTYPDSC

The POLICYTYPE table specifies the type of business policy that you are creating. It contains a single column, POLICYTYPE_ID, that is the primary key. An example value is `Price`. If you create a new business policy type, ensure that you specify a unique POLICYTYPE_ID.

The PLCYTYCMIF table is the business policy type to command interface relationship specification table. That is, for each business policy type, it specifies the Java command interface for the business policy object. While there can be zero or more business policy commands that implement a business policy, each of the business policy commands must implement the interface specified here.

The PLCYTYPDSC table specifies a description of the business policy type. It includes a language identifier of the description and the description of the business policy type.

To create a new business policy type, create an entry in each of these tables for the new business policy type. The following SQL statements provides an example:

```
insert into POLICYTYPE (POLICYTYPE_ID) values ('MyNewPolicyType');
insert into PLCYTYCMIF (POLICYTYPE_ID, BUSINESSCMDIF)
   values ('MyNewPolicyType',
   'com.mycompany.mybusinesspolicycommands.MyNewPolicy');
```

```
insert into  PLCYTYPDSC (POLICYTYPE_ID, LANGUAGE_ID, DESCRIPTION)
   values ('MyNewPolicyType', -1,
   'My new policy type for example purposes.');
```

As the final step of creating the new business policy type, you may code one or more new business policy type interfaces. These interfaces are then implemented by any business policy command that falls under the realm of this business policy type. For example, in the ToolTech sample store, Price is defined as a business policy type. As such, there are the `com.ibm.commerce.price.commands.ResolvePriceListsCmd` and `com.ibm.commerce.price.commands.RetrievePricesCmd` interfaces that are implemented by all price-related business policy commands.

If you will not have a business policy command that performs operations on the new business policy type, then you are not required to create a new interface. This is rare, and in most cases when creating a new business policy type, you must create a new business policy type interface as well.

When you create a business policy type interface, the new interface must extend the `com.ibm.commerce.command.BusinessPolicyCommand` interface.

Writing the new business policy command

To create a new business policy command, you must create a new command that implements the interface of the business policy type to which the command relates. The new command must also extend `com.ibm.commerce.command.BusinessPolicyCommandImpl` implementation class. This is very similar to creating a new controller or task command.

There are two different approaches by which you can pass input properties to a business policy command. The first way is to have default input properties specified in the PROPERTIES column of the POLICY table. For more information about this table, refer to the following section.

The second approach is to create a new field in the command for each of the input properties. For each field, create a new pair of getter and setter methods.

Setting requestProperties in business policy commands

There are two ways in which requestProperties are set in a business policy command object. The first way uses the PROPERTIES column of the POLICY table to set the default properties. This is accomplished by the setRequestProperties method. The second way to set properties is to have the command (controller or task) that calls the business policy command explicitly set other required properties.

When creating a new business policy command, you should override the default setRequestProperties method to include the logic to explicitly set each of the parameters that are included in the requestProperties object.

Consider an example of a new business policy command that has an interface name of MyNewBusinessPolicyCmd and implementation class name of MyNewBusinessPolicyCmdImpl.

Assume that the entry in the POLICY table for this new business policy command includes the following values in the PROPERTIES column:

- defaultProperty1=apple
- defaultProperty2=orange
- defaultProperty3=banana

The interface for this new business policy command is defined as follows:

```
public interface MyNewBusinessPolicyCmd extends
   com.ibm.commerce.command.BusinessPolicyCmd {
      java.lang.String defaultCommandClassName =
         'com.mycompany.mycommands.MyNewBusinessPolicyCmdImpl';
      public void setProperty1();
      public void setProperty2();
}
```

The implementation class for this new business policy command is defined as follows:

```
public class MyNewBusinessPolicyCmdImpl extends
   com.ibm.commerce.command.BusinessPolicyCmdImpl
   implements com.mycompany.mycommands.MyNewBusinessPolicyCmd {
      // Establish default properties that are stored in the POLICY table

      private java.lang.String defaultProperty1;
      private java.lang.String defaultProperty2;
      private java.lang.String defaultProperty3;

      // Begin to establish properties that must be set
     // by the calling command.

      //  *** property1   ***
      private java.lang.String property1;
      public java.lang.String getProperty1() {
         return property1;
         }
      public void setProperty1(java.lang.String newProperty1) {
         property1 = newProperty1;
      }

      //  *** property2   ***
      private java.lang.String property2;
      public java.lang.String getProperty2() {
         return property2;
```

```
            }
         public void setProperty1(java.lang.String newProperty2) {
            property2 = newProperty2;
            }

         // End establishing properties that must be set
         // by the calling command.

         /* Upon instantiation the business policy command sets all
            default properties from the POLICY table into the
            requestProperties object.  The calling command
            is responsible for setting any other required properties.
         */

         public void setRequestProperties(com.ibm.commerce.datatype.TypedProperty
            requestProperties) {
            //  Get the default properties defined in the POLICY table
            setDefaultProperty1(requestProperties.get("defaultProperty1"));
            setDefaultProperty2(requestProperties.get("defaultProperty2"));
            setDefaultProperty3(requestProperties.get("defaultProperty3"));
            }

}
```

The command that calls the new business policy command could be defined in a manner similar to the following:

```
public class MyCallerCommandImpl
   extends com.ibm.commerce.command.TaskCommandImpl
   implements com.mycompany.mycommands.MyCallerCommand {

   /* Include all elements and processing required for the
      task command.
   */

   // Determine the policy ID and setPolicyId

   // Call the business policy command.

   cmd = (MyNewBusinessPolicyCmd) CommandFactory
           createPolicyCommand(policyId);

   // Set required properties

   cmd.setProperty1("Fruit salad");
   cmd.setProperty2("Favorite food");

 cmd.execute();
}
```

Registering the new business policy and business policy command

After you have created the new business policy command, you must register both the business policy and the business policy command in the database.

Business policies are registered in the POLICY table. This table contains the following columns:

- POLICY_ID
  The primary key. This is the policy identifier.
- POLICYNAME
  A unique policy name.
- POLICYTYPE_ID
  The policy type identifier. This is the foreign key to the POLICYTYPE table.
- STOREENT_ID
  The store or store group to which the policy applies.
- PROPERTIES
  Default properties that can be set to the business policy command. Specified as name-value pairs, for example, `parm1=val1&parm2=val2`.
- STARTDATE
  The starting date (specified as a timestamp) of the policy. If NULL, the starting date is immediate.
- ENDDATE
  The ending date (specified as a timestamp) of the policy. If NULL, there is no end date.

Once the new policy is registered in the POLICY table, you must register a relationship between the policy and the business policy command that implements the business policy. The POLICYCMD table is used for this purpose. The POLICYCMD table contains the following columns:

- POLICY_ID
  Foreign key reference to the POLICY table.
- BUSINESSCMDCLASS
  The business policy command that implements the policy.
- PROPERTIES
  Default properties that can be set to the business policy command. Specified as name-value pairs, for example, `parm1=val1&parm2=val2`.

Relating a terms and conditions object to a new business policy

In the WebSphere Commerce contracts and policies framework, terms and conditions (also referred to as *terms*) provide a way to describe an agreement between a buyer and a seller. Terms and conditions can be used in various types of trading agreements, such as contract and RFQ (request for quotation). Terms and conditions objects usually refer to business policies with an optional adjustment. For example, a price terms and conditions object is created by choosing one of the price policy objects. In the price term, an account manager can make adjustments to the store standard price, such as:

- A percentage discount over the standard price list

- A percentage discount on a specified set of the products

Each of the adjustments is specified as a term and condition.

When you create a new business policy, there must be at least one terms and conditions object that refers to this business policy, if the policy is to be used in a contract. You can either relate an existing term and condition object to the new business policy (this is done by capturing the relationship between the existing terms and conditions object and the new business policy in the `B2BTrading.dtd` file), or you can create a new terms and conditions object that is related to the new business policy.

Creating new terms and conditions

Within the WebSphere Commerce architecture, new terms and conditions objects are created by performing the following steps:

1. Updating the database schema to include the new term and condition.
2. Updating the `B2BTrading.dtd` file to reflect the new term and condition.
3. Creating a new enterprise bean for the term and condition.
4. Updating WebSphere Commerce Accelerator to reflect the new term and condition, or using the contract load command to create a new contract using the new term and condition.

In the following sections, the example of `MyTC` is the new term and condition object.

Registering the new term and condition in the database

When you are creating a new terms and condition object, you must update the database schema to include this object. The database tables that must be updated are TCTYPE and TCSUBTYPE.

The following SQL statement shows an example of how to update the schema:

```
insert into TCTYPE (TCTYPE_ID) values ('MyTC');
insert into TCSUBTYPE (TCSUBTYPE_ID, TCTYPE_ID, ACCESSBEANNAME,
   DEPLOYCOMMAND)
values ('MySubTC, 'MyTC ',
   'com.ibm.commerce.contract.objects.MySubTCAccessBean',
   'packagename.MySubTCDeployCmd');
```

Register the new term and condition in the contract document type definition

The `B2BTrading.dtd` is the document type definition (DTD) file that specifies the various terms and conditions that can be used within business policies. To make the new term and condition available in contracts, you must update this file to include the new term and condition.

When you have created a new term and condition, you must add the new term and condition to the TermCondition definition and create a new element that describes the term and condition.

To update the `B2BTrading.dtd` file, do the following:

1. Navigate to the following directory:
   - Windows *drive*`:\WebSphere\CommerceServer\xml\trading`
   - AIX `/usr/WebSphere/CommerceServer/xml/trading`
   - Solaris `/opt/WebSphere/CommerceServer/xml/trading`
   - Linux `/opt/WebSphere/CommerceServer/xml/trading`
   - 400 `/QIBM/ProdData/WebCommerce/xml/trading`
2. Open the `B2BTrading.dtd` file.
3. Update the TermCondition definition with the new term and condition. For example, the update is shown in bold in the following TermCondition definition:

   ```
   <!ELEMENT TermCondition (TermConditionDescription?,Participant*,
   CreateTime?,UpdateTime?,(PriceTC|ProductSetTC|ShippingTC|FulfillmentTC|
   PaymentTC|ReturnTC|InvoiceTC|RightToBuyTC|ObligationToBuyTC|
   PurchaseOrderTC|OrderApprovalTC|DisplayCustomizationTC|
   OrderTC|MyTC))>
   ```

   Note that the line breaks are for display purposes only.
4. Now add the new element to the `B2BTrading.dtd` file. For example, the following shows the update to add the MyTC element, which refers to a business policy and has two required attributes.

   ```
   <!ELEMENT MyTC (MySubTC)>
   <!ELEMENT MySubTC (PolicyReference)>
   <!ATTLIST MySubTC
      attr1 CDATA #REQUIRED
      attr2 CDATA #REQUIRED
   >
   ```
5. Save the file.

Creating a new CMP enterprise bean for the term and condition

You must create a new CMP enterprise bean for the term and condition object. The bean is created for the term and condition subtype.

Note that typically when creating new enterprise beans, you would place the beans into your own EJB group, rather than including them into one of the EJB groups that contain WebSphere Commerce entity beans. In this case however, since all new entity beans for terms and conditions must inherit from the WebSphere Commerce TermCondition bean, you must place your new term and condition beans into the WCS Contract EJB group.

To create the new CMP enterprise bean for the term and condition object, do the following in VisualAge for Java:

1. Use a wizard to create the new enterprise bean by doing the following:
   a. In the VisualAge for Java Workbench, select the EJB tab.
   b. Highlight, then right-click on the **WCS Contract** EJB group and select **Add > Enterprise Bean with Inheritance**.
      The Create Enterprise Bean with Inheritance SmartGuide opens.
   c. In the SmartGuide, enter information appropriate for your bean. For example, the following table shows example values.

Attribute	**Value**
Bean name	`MySubTC`
Inherit from	`TermCondition`
Package	`com.ibm.commerce.contract.objects`
Bean class	`MySubTCBean`
Remote interface	`MySubTC`
Home interface	`MySubTCHome`

   d. Click **Add** for adding CMP fields to the bean and create new fields for the bean, as required. For this example, two new CMP fields are created using the following information:

Attibute	**Value**
Field name	`attr1`
Field type	`String`
Access with getter and setter methods	enable
Promote getter and setter methods to remote interface	enable

Attibute	**Value**
Field name	`attr2`
Field type	`Integer`
Access with getter and setter methods	enable
Promote getter and setter methods to remote interface	enable

   e. Click **Finish**.
2. The next step is to map the fields from the new bean to columns in the TERMCOND table. To create this mapping information, do the following:

a. From the **EJB** menu, select **Open To > Schema Maps**.
The Map Browser opens.

b. In the Datastore Maps panel of the maps browser, double-click **WCS Contract**.

c. In the Persistent Classes panel, double-click **TermCondition** then select **MySubTC**.

d. From the Table Maps menu, select **New Table Map > Add Single Inheritance Table Map**.
The Single Inheritance Table Map Editor opens.

e. In the **Discriminator value** field, enter the TCSUBTYPE_ID value. For example, in this case enter `'MySubTC'` (include the quotes) and click **OK**.

f. Ensure that **MySubTC** is still selected in the Persistent Classes panel. In the Table Maps panel, highlight and right-click the TERMCOND table. Select Edit Property Maps.
The Property Maps Editor opens.

g. In the Property Maps Editor, set the attributes as follows:

Class Attribute	**Map Type**	**Table Column**
attr1	Simple	STRINGFIELD2
attr2	Simple	INTEGERFIELD1

and click **OK**.

h. From the Datastore Maps menu, select Save Datastore Map. Enter the following information when saving the map:

Attribute	**Value**
Project	`IBM WCS Enterprise Beans`
Package	`WCSContract EJB Reserved`
Class Name	`WCSContractMap`

Click **Finish** and then close the Map Browser.

3. In the new enterprise bean (that is, in MySubTCBean) create a new `ejbCreate(java.lang.Long argTradingId, org.w3c.dom.Element argElement)` method, as follows:

```
public void ejbCreate(java.lang.Long argTradingId,
   org.w3c.dom.Element argElement)
   throws javax.ejb.CreateException, javax.ejb.FinderException,
   java.rmi.RemoteException, javax.naming.NamingException,
   javax.ejb.RemoveException {
    _initLinks();
```

```
     super.ejbCreate (argTradingId, argElement);
     this.attr1= null;
     this.attr2= null;
}
```

4. Create a new `ejbPostCreate(java.lang.Long argTradingId, org.w3c.dom.Element argElement)` methos, as follows:

```
public void ejbPostCreate(java.lang.Long argTradingId,
   org.w3c.dom.Element argElement)
   throws javax.ejb.CreateException, javax.ejb.FinderException,
   java.rmi.RemoteException, javax.naming.NamingException,
   javax.ejb.RemoveException
   {
    parseXMLElement(argElement);
   }
```

5. Override the `parseXMLElement(org.w3c.dom.Element argElement)` method in MySubTCBean, as follows:

```
public void parseXMLElement(org.w3c.dom.Element argElement) throws
   javax.ejb.CreateException,
   javax.ejb.FinderException,
   java.rmi.RemoteException,
   javax.naming.NamingException,
   javax.ejb.RemoveException
   {
      super.parseXMLElement(argElement);

      if (argElement == null)
         return;

      String nodeName = argElement.getNodeName();
      if (nodeName.equals("TCCopy"))
         return;

      // get element "MyTC"
      Element eMyTC = ContractUtil.getElementByTag(argElement,"MyTC");

      // get element "MySubTC" from element "MyTC"
      Element eMySubTC = ContractUtil.getElementByTag(eMyTC ,"MySubTC");
      this.attr1 = eMySubTC .getAttribute("attr1").trim();
      this.attr2 = new Integer (eMySubTC .getAttribute("attr2").trim());

      // get element "PolicyReference" from "MySubTC"
      Element ePolicyReference = ContractUtil.getElementByTag(eMySubTC,
         "PolicyReference");
      parseElementPolicyReference(ePolicyReference);

}
```

6. Override the `createNewVersion(Long argNewTradingId)` method in MySubTCBean, as follows:

```
public Long createNewVersion(Long argNewTradingId) throws
   javax.ejb.CreateException,
   javax.ejb.FinderException,
   java.rmi.RemoteException,
   javax.naming.NamingException,
   javax.ejb.RemoveException,
   org.xml.sax.SAXException,
   java.io.IOException

   {

   // Contract a seqElement since tcSequence can not be null
   Element seqElement = ContractUtil.getSeqElementFromTCSequence(
      this.tcSequence);
   MySubTCAccessBean newTC = new MySubTCAccessBean(argNewTradingId,
     seqElement);

   Long newTCId = newTC.getReferenceNumberInEJBType();
   newTC.setInitKey_referenceNumber(newTCId.toString());
   newTC.setMandatoryFlag(this.mandatoryFlag);
   newTC.setChangeableFlag(this.changeableFlag);
   // set columns for this specific TC
   newTC.setAttr1(this.attr1);
   newTC.setAttr2(this.attr2);
   newTC.commitCopyHelper();

   return newTCId;
}
```

7. Override the `getXMLString()` method in MySubTCBean, as follows:

```
public String getXMLString() throws
   javax.ejb.CreateException,
   javax.ejb.FinderException,
   java.rmi.RemoteException,
   javax.naming.NamingException
   {
      String xmlTC = "      <MyTC>"  +
                    "%XML_POLICYREFERENCE%" +
                    "         <MySubTC attr1=\"" + this.attr1 +
                    "\"  attr2=\"" + this.attr2.toString() + "\"/>"
                    "         '>" +
                    "      <MYTC></MYTC>" ;
      String xmlPolicy = getXMLStringForElementPolicyReference(
         "ProductSet") ;
      xmlTC = ContractUtil.replace( xmlTC, "%XML_POLICYREFERENCE%",
         xmlPolicy );

      return xmlTC;
}
```

8. Override the `markForDelete()` method in MySubTCBean, as follows:

```
public void markForDelete() throws
   javax.ejb.CreateException,
   javax.ejb.FinderException,
   java.rmi.RemoteException,
   javax.naming.NamingException
   {
      // code: remove entries from associated tables which
      // cannot be deleted though delete cascade
}
```

9. Ensure that the `ejbCreate` method has been added to the home interface and that all other modified methods have been added to the remote interface.
10. The next step is to create an access bean for the MySubTC entity bean by doing the following:
    a. Right-click the **MySubTC** entity bean and select **Add > Access Bean**. The Create Access Bean SmartGuide opens.
    b. Ensure the following information is entered:

Table 1.

Attribute	**Value**
EJB Group	`WCSContract`
Enterprise Bean	`MySubTC`
Access Bean Name	`MySubTCAccessBean`
Access Bean Type	`Copy Helper for an Entity Bean`

    and click **Next**.
    c. From the **Select home method for zero argument constructor** drop-down list, select **findByPrimaryKey(TermConditionKey)**
    d. For **initKey_referenceNumber** (in the Initial Properties column), set **Converter** to `com.ibm.commerce.base.objects.WCStringConverter` and click **Next**.
    e. For all new fields that have been added, ensure that **CopyHelper** is selected and set the converter value for each to `com.ibm.commerce.base.objects.WCStringConverter`. Click **Finish**. After the code generation is complete, you can view the new code by switching to the **Projects** tab, expanding the **IBM WCS Enterprise Beans** project and then expanding the **com.ibm.commerce.contract.objects** package.
11. Back on the EJB tab, right-click the **MySubTC** enterprise bean and select **Generate Deployed Code**.
12. You must also regenerate the deployed code for the parent bean (the TermCondition bean) and all of the sibling beans (all of the other beans in the WCS Contract EJB group that contain “TC” in their names). Note that if you had added a new field or modified the remote interface of the existing TermCondition bean, then you would have to regenerate the access beans for itself and all of its child beans as well.
    To regenerate the deployed code, do the following:

a. Highlight the TermCondition bean and all other beans that contain "TC" in their names (for example, DisplayCustomizationTC, FulfillmentTC, and InvoiceTC are a few of the sibling beans).

b. With all of these beans highlighted, right-click and select **Generate Deployed Code**.

13. The next step is to override methods that are in the `ValidateContractCmd` task command. In this command, there are three methods that you may want to override to support the new term and condition object. They are:

 - `validateTCType()`
 This method checks what type of term can be in a contract. For example, the InvoiceTC belongs to account and therefore, it cannot appear in a contract.
 - `validateTCOccurrence()`
 This method checks the occurrence of the terms. For example, in the default implementation of this method, a contract has to have at least one PriceTC.
 - `otherValidateCheck()`
 The default implementation of this method is empty. You can add any additional validation that does not fall into the first two methods.

14. If the term and condition must be deployed, you must create a new deployment command and register this command in the database. If required, do the following:

 a. In this example, the new deployment command interface is called `MySubTCDeployedCmd` and the implementation class is called `MySubTCDeployedCmdImpl`. In addition, the command is packaged in the `packagename` package. To register this command, issue the following SQL command:

```
insert into CMDREG (STOREENT_ID, INTERFACENAME, CLASSNAME, TARGET)
values (0, 'packagename.MySubTCDeployCmd',
'packagename.MySubTCDeployCmdImpl', 'Local');
```

 b. In the `packagename` package, create the new `MySubTCDeployedCmd` interface. This interface must extend the `com.ibm.commerce.contract.commands.DeployTCCmd` command interface. The following describes the new command interface:

```
public interface MySubTCDeployCmd extends
   com.ibm.commerce.contract.commands.DeployTCCmd
   {
       // customized code
   }
```

 There is a protected parameter `abTC` and a method called `getTargetStoreId()` in `DeployTCCmd`. The value of `abTC` is `MySubTCAccessBean` and the `getTargetStoreId()` method returns the identifier of the store to which the contract is being deployed.

c. In the same package, create the `MySubTCDeployCmdImpl` implementation class. This implementation class must extend `com.ibm.commerce.contract.commands.DeployTCCmdImpl`. The following describes the new command implementation class:

```
public class MySubTCDeployCmdImpl
   extends com.ibm.commerce.contract.commands.DeployTCCmdImpl
   implements MySubTCDeployCmd
   {
     // customer code
   }
```

Updating the WebSphere Commerce Accelerator to use a new term and condition

Once you have created new terms and conditions, you can update the WebSphere Commerce Accelerator so that it can be used to create new contracts that include those new terms and conditions. Updating the WebSphere Commerce Accelerator for this purpose includes the following steps:

1. Creating a new JavaScript file for the new terms and conditions. For the purpose of the example in this section, this file is referred to as `Extensions.js`.
2. Creating a new JSP template that includes an HTML section in which a user can enter required information for the new terms and conditions. For the purpose of the example in this section, this file is referred to as `ContractMyTC.jsp`.
3. Creating a new data bean for the new terms and conditions. For the purpose of the example in this section, this file is referred to as `MyTCDataBean`.
4. Registering the new view in the VIEWREG table.
5. Updating the `ContractRB_`*`locale`*`.properties` file to include the new resources.
6. Editing the `ContractNotebook.xml` file to include the new page.

Each of these steps is described in more detail in the following sections.

Creating the new JavaScript file: The first step to updating the WebSphere Commerce Accelerator to use new terms and conditions is to create a new JavaScript file for them. For reference, you can refer to the following sample file:

- Windows *`drive`*`:\WebSphere\CommerceServer\samples\contract\Extensions.js`
- Windows *`drive`*`:\WebSphere\CommerceServerDev\samples\contract\Extensions.js`
- AIX `/usr/WebSphere/CommerceServer/samples/contract/Extensions.js`

- Solaris /opt/WebSphere/CommerceServer/samples/contract/Extensions.js
- Linux /opt/WebSphere/CommerceServer/samples/contract/Extensions.js
- 400 /QIBM/ProdData/WebCommerce/samples/contract/Extensions.js

In order to use this sample file, copy it to the following directory:

- Windows *drive*:\WebSphere\AppServer\installedApps\ WC_Enterprise_App_*instanceName*.ear\wctools.war\ javascript\tools\contract
- AIX /usr/WebSphere/AppServer/installedApps/ WC_Enterprise_App_*instanceName*.ear/wctools.war/ javascript/tools/contract
- Solaris /opt/WebSphere/AppServer/installedApps/ WC_Enterprise_App_*instanceName*.ear/wctools.war/ javascript/tools/contract
- Linux /opt/WebSphere/AppServer/installedApps/ WC_Enterprise_App_*instanceName*.ear/wctools.war/ javascript/tools/contract
- 400 /QIBM/UserData/WebASAdv4/*WAS_AdminInstanceName*/ installedApps/WC_Enterprise_App_*instanceName*.ear/ wctools.war/javascript/tools/contract

In this new file, you must create a JavaScript object to store the data for the new term and condition. This is shown in the following code snip:

```
function ContractMyTCModel() {

   this.tcReferenceNumber = "";
   this.policyReferenceNumber = "";

   this.attr1 = "";
   this.attr2 = "";

   this.policyList = new Array();
   this.selectedPolicyIndex = "0";
}
```

You should also create a new JavaScript object to submit the new term and condition. This must be done in a manner consistent with the extensions that you made to the B2BTrading.dtd file. This is shown in the following code snip:

```
function submitMyTC(termsAndConditions) {

   var tcModel = get("ContractMyTCModel");
   if (tcModel != null) {

      var myTC = new Object();
      myTC.MyTC = new Object();
      myTC.MyTC.MySubTC = new Object();
      myTC.MyTC.MySubTC.attr1 = tcModel.attr1;
      myTC.MyTC.MySubTC.attr2 = tcModel.attr2;

      myTC.MyTC.PolicyReference = new Object();
      myTC.MyTC.PolicyReference.policyName =
         tcModel.policyList[tcModel.selectedPolicyIndex].policyName;
      myTC.MyTC.PolicyReference.policyType = "ProductSet";
      myTC.MyTC.PolicyReference.storeIdentity =
         tcModel.policyList[tcModel.selectedPolicyIndex].storeIdentity;
      myTC.MyTC.PolicyReference.Member =
         tcModel.policyList[tcModel.selectedPolicyIndex].member;

      if (tcModel.tcReferenceNumber != "") {
         // Change the term and condition
         myTC.action = "update";
         myTC.referenceNumber = tcModel.tcReferenceNumber;
      }
      else {
         // Create a new term and condition
         myTC.action = "new";
      }

      termsAndConditions[termsAndConditions.length] = myTC;
   }

   return true;
}
```

Creating the new JSP template: The next step is to create a new JSP template that includes an HTML section in which the user can enter information required by the new term and condition. For reference, you can refer to the following sample file:

- Windows *drive*:\WebSphere\CommerceServer\samples\contract\ContractMyTC.jsp
- Windows *drive*:\WebSphere\CommerceServerDev\samples\contract\ContractMyTC.jsp
- AIX /usr/WebSphere/CommerceServer/samples/contract/ContractMyTC.jsp
- Solaris /opt/WebSphere/CommerceServer/samples/contract/ContractMyTC.jsp
- Linux /opt/WebSphere/CommerceServer/samples/contract/ContractMyTC.jsp

- 400 `/QIBM/ProdData/WebCommerce/samples/contract/ContractMyTC.jsp`

In order to use this sample file, copy it to the following directory:

- Windows *drive*`:\WebSphere\AppServer\installedApps\WC_Enterprise_App_`*instanceName*`.ear\wctools.war\tools\contract`
- AIX `/usr/WebSphere/AppServer/installedApps/WC_Enterprise_App_`*instanceName*`.ear/wctools.war/tools/contract`
- Solaris `/opt/WebSphere/AppServer/installedApps/WC_Enterprise_App_`*instanceName*`.ear/wctools.war/tools/contract`
- Linux `/opt/WebSphere/AppServer/installedApps/WC_Enterprise_App_`*instanceName*`.ear/wctools.war/tools/contract`
- 400 `/QIBM/UserData/WebASAdv4/`*WAS_AdminInstanceName*`/installedApps/WC_Enterprise_App_`*instanceName*`.ear/wctools.war/tools/contract`

The following code snip shows an example HTML section of a JSP template that can be used for MyTC.

```
<!--
///////////////////////////////////////
// HTML SECTION
///////////////////////////////////////
-->

<BODY onLoad="onLoad()" class="content">

   <H1>
   <%= contractsRB.get("MyTCHeading") %>
   </H1>

<FORM NAME="MyTCForm">

   <%= contractsRB.get("MyTCAttr1Label") %>
   <BR>
   <INPUT type=text name=Attr1 value="" size=10 maxlength=10>
   <BR>

   <%= contractsRB.get("MyTCAttr2Label") %>
   <BR>
   <INPUT type=text name=Attr2 value="" size=10 maxlength=10>
   <BR>

   <%= contractsRB.get("MyTCPolicyLabel") %>
   <BR>
   <SELECT NAME="PolicyList" SIZE="1">
   </SELECT>

</FORM>
```

Creating the new data bean: In this step, you create a new data bean that loads the necessary data from the MySubTC access bean. The relevant sections of code are shown in the following code snip:

```
public class MyTCDataBean extends MySubTCAccessBean
   implements SmartDataBean, Delegator {
      private java.lang.Long contractId;
      private boolean hasMyTC = false;
      private CommandContext iCommandContext;


      /**
       * MyTCDataBean default constructor.
       */
      public MyTCDataBean() {
      }
      /**
       * MyTCDataBean constructor.
       */
      public MyTCDataBean(Long newContractId) {
       contractId = newContractId;
      }


      /*
       * populate the attributes from TermConditionAccessBean
       */
      public void populate() throws Exception {

            Enumeration myTCEnum = new TermConditionAccessBean().
                     findByTradingAndTCSubType(contractId, "MySubTC");
                  if (myTCEnum != null) {
                        // assume a contract only has one MyTC for this example

                        setEJBRef(((TermConditionAccessBean)
                              myTCEnum.nextElement()).getEJBRef());
                        refreshCopyHelper();
                        hasMyTC = true;
                  }

}
```

Registering the new view in the VIEWREG table: You must register your newly created view in the VIEWREG table. The following is an example SQL statement to register the new view.

```
insert into VIEWREG(VIEWNAME,DEVICEFMT_ID,STOREENT_ID, INTERFACENAME,
   CLASSNAME, PROPERTIES, HTTPS, INTERNAL)
values ('ContractMyTCPanelView', -1, 0,
   'com.ibm.commerce.tools.command.ToolsForwardViewCommand',
   'com.ibm.commerce.tools.command.ToolsForwardViewCommandImpl',
   'docname=tools/contract/ContractMyTC.jsp', 1, 1)
```

Updating the `ContractRB_*locale*.properties` file: You must update the following properties file with information specific to the new term and condition:

- Windows *drive*:\WebSphere\AppServer\installedApps\WC_Enterprise_App_*instanceName*.ear\properties\com\ibm\commerce\tools\contract\properties\ContractRB_*locale*.properties
- AIX /usr/WebSphere/AppServer/installedApps/WC_Enterprise_App_*instanceName*.ear/properties/com/ibm/commerce/tools/contract/properties/ContractRB_*locale*.properties
- Solaris /opt/WebSphere/AppServer/installedApps/WC_Enterprise_App_*instanceName*.ear/properties/com/ibm/commerce/tools/contract/properties/ContractRB_*locale*.properties
- Linux /opt/WebSphere/AppServer/installedApps/WC_Enterprise_App_*instanceName*.ear/properties/com/ibm/commerce/tools/contract/properties/ContractRB_*locale*.properties
- 400 /QIBM/UserData/WebASAdv4/*WAS_AdminInstanceName*/installedApps/WC_Enterprise_App_*instanceName*.ear/properties/com/ibm/commerce/tools/contract/properties/ContractRB_*locale*.properties

The following is an example of the information that you would add to the file.

```
MyTCHeading=My TC
attr1Empty=Attribute One must be entered.
attr2Empty=Attribute Two must be entered.
attr1TooLong=Attribute One is too long.
attr2TooLong=Attribute Two is too long.
MyTCAttr1Label=Attribute One (required)
MyTCAttr2Label=Attribute Two (required)
MyTCPolicyLabel=Policy
```

Editing the `ContractNotebook.xml` file: The last step for including new terms and conditions in the WebSphere Commerce Accelerator is to update the following file to include the new page.

- Windows *drive*:\WebSphere\CommerceServer\xml\tools\contract\ContractNotebook.xml

- AIX `/usr/WebSphere/CommerceServer/xml/tools/contract/ContractNotebook.xml`
- Solaris `/opt/WebSphere/CommerceServer/xml/tools/contract/ContractNotebook.xml`
- Linux `/opt/WebSphere/CommerceServer/xml/tools/contract/ContractNotebook.xml`
- 400 `/QIBM/UserData/WebCommerce/instances/`*`instanceName`*`/xml/tools/contract/ContractNotebook.xml`

The following is an example snip of code that is used to include the new page in this example.

```
<panel name="MyTCHeading"
    url="ContractMyTCPanelView"
    parameters="contractId,accountId"
    helpKey="MC.contract.MyTCPanel.Help" />
```

Importing the new contract using the new term and condition

As an alternative to updating the WebSphere Commerce tools to use a new term and condition , you can use the contract import command (refer to the WebSphere Commerce online help for information about this command) to import a new contract that includes this new term and condition. After importing, the relevant section in the `Contract.xml` file appears as follows:

```
<TermCondition>
   <MyTC>
      <MySubTC attr1="adc" attr2="123" />
         <PolicyReference policyName = "Product Set 1"
                             policyType = "ProductSet"
                             storeIdentity = "StoreGroup1" >
            <Member>
               <User distinguishName = "uid=wcsadmin,o=Root Organization"/>
            </Member>
         </PolicyReference>
      </MyTC>
 </TermCondition>
```

Invoking the new business policy

Once you have created a new business policy and this business policy has been associated with at least one terms and conditions object, you can must update your application logic to invoke the new business policy commands.

Business policy commands are invoked from within controller and task commands.

The command factory is used to invoke business policy commands. There are two `create` methods that can be used to invoke business policy commands.

The first is used to invoke a business policy command when there is only one business policy command associated with the business policy. This is shown in the following code snippet:

```
CommandFactory createBusinessPolicyCommand(Long policyId);
```

The second method is used to invoke a business policy command when there is more than one business policy command associated with the business policy. This is shown in the following code snippet:

```
CommandFactory createBusinessPolicyCommand(Long policyId, String cmdIfName);
```

In the preceding example, `cmdIfName` is used to specify the interface name of the business policy command to be created.

The command factory looks up the policy object in the POLICYCMD table to determine the command that implements this policy. It also fetches any default properties from the table and sets them as requestProperties in the business policy command.

The following code snippet shows an example of invoking a refund policy:

```
RefundPolicyCmd  cmd;

   ////////////////////////////////////////////////////////
   // Get the refund policy id from the refundTC object     //
   //  and use it to create the policy command.            //
   ////////////////////////////////////////////////////////
 cmd = (RefundPolicyCmd) CommandFactory
        createPolicyCommand (refundTC.getRefundPolicy);

  cmd.execute()
```

Creating a contract

The next step to fully integrate the extenstion to the contract model into your business process is to create a contract that includes the terms and condition which refers to the new business policy. A contract can be created using the WebSphere Commerce Accelerator or by using one of the contract URL commands (ContractImportApprovedVersion and ContractImportDraftVersion). For more information about creating contracts, refer to the WebSphere Commerce online help.

Contract customization scenarios

This section provides an overview of the steps involved for the following contract customization scenario:

- Enabling a rebate

Rebate scenario

In this example scenario, a flat rate rebate is created. Since the ToolTech sample store includes neither a term and condition nor a policy type that matches the rebate scenario, these must be created. Additionally, a new business policy must be created, as well as a database table to store the rebate codes.

Implementing this rebate scenario includes the following high-level steps:

1. Creating the XREBATECODE database table and a corresponding XRebateCodeBean entity bean that is used to access information from this table.
2. Creating a new 5DollarRebate business policy by performing the following sub-tasks:
   a. Create the corresponding new business policy type. This defines the interface (RebatePolicyCmd) that the new business policy command will implement.
   b. Create the new CalculateRebateCmdImpl business policy command.
   c. Register the new business policy command and business policy type in the database.
3. Create a new term and condition (RebateTC) for the rebate by performing the following sub-tasks:
   a. Register the RebateTC term and condition in the database.
   b. Update the `B2BTrading.dtd` file to reflect the new RebateTC.
   c. Create a new enterprise bean for the RebateTC.
   d. Update the WebSphere Commerce Accelerator to reflect the new RebateTC.
4. Create a new contract that uses the RebateTC.
5. Integrating the new business policy into the shopping flow.

Each of these steps is described in more detail in subsequent sections.

Step 1: Creating the new table and enterprise bean

Since the existing database schema does not include the specification of a rebate amount and code, a new table must be created. In general, when a new table is created, a new entity bean is also created that is used to when accessing information contained in this table.

For the purpose of this example, assume that the following XREBATECODE database table is created.

Table 2. XREBATECODE database table

	Column name		
	REBATECODE_ID	AMOUNT	CURRENCY
Sample data	201	5	CAD
	202	10	CAD

Additionally, a new CMP entity bean (XRebateCodeBean) would be created. For detailed information about creating this bean, refer to "Creating a new CMP enterprise bean" on page 57.

Step 2: Creating the "5DollarRebate" business policy

In order to create this new business policy, you must perform the following steps:

1. Create the new business policy type interface. This is the `RebatePolicyCmd` interface that the `CalculateRebateCmdImpl` will implement
2. Create the new `CalculateRebateCmdImpl` business policy command.
3. Register the new business policy and business policy command in the database.

Creating the "Rebate" business policy type: Since there is not an existing business policy type that corresponds to rebates, a new one must be created. Creating a new business policy type involves defining and registering a policy type in the database. The following tables must be updated:

- POLICYTYPE
- PLCYTYCMIF
- PLCYTYPDSC

For this scenario, to create the new REBATE policy type, the following SQL statements would be used:

```
insert into POLICYTYPE (POLICYTYPE_ID) values ('Rebate');
insert into PLCYTYCMIF (POLICYTYPE_ID, BUSINESSCMDIF)
   values ('Rebate',
   'com.mycompany.mybusinesspolicycommands.RebatePolicyCmd');
insert into  PLCYTYPDSC (POLICYTYPE_ID, LANGUAGE_ID, DESCRIPTION)
   values ('Rebate', -1,
   'Rebate policy type.');
```

As a result, the following table shows the relevant columns of the PLCYTYCMIF table that shows the relationship between the policy type and the business policy command to which it is related.

Table 3. Updates made to the PLCYTYCMIF table

	Column name	
	POLICYTYPE_ID	BUSINESSCMDIF
Sample data	Rebate	com.mycompany.mybusinesspolicycommands. RebatePolicyCmd

You must also code the new `RebatePolicyCmd` interface. This interface must extend the `com.ibm.commerce.command.BusinessPolicyCommand` interface. As suggested by the previous table, package this interface into your own package.

Creating the CalculateRebateCmdImpl business policy command: To create the new business policy command, you must create a new command called `CalculateRebateCmdImpl` that extends the `com.ibm.commerce.command.BusinessPolicyCommandImpl` implementation class. This command should implement the `RebatePolicyCmd` interface created in the previous step.

Note that in this example, the interface name and the command name are dissimilar. These names were chosen to intentionally show that there may be many business policy commands that implement the rebate type of business policy. Each implementation (that is, each business policy command) would then implement the rebate in a unique manner.

The logic of the command depends upon the particular implementation of how the customer is to pick up the goods. Additionally, this `CalculateRebateCmdImpl` should be invoked by a separate controller or task command in your application.

Registering the new business policy and new business policy command: The new business policy must be registered in the database. You must also register the relationship between the new business policy and the new business policy command.

To register this information, you can use the `com.ibm.commerce.contract.commands.PolicyAddCmd` command. The following shows an example usage of the PolicyAdd command for this scenario:

```
http://localhost:8080/webapp/wcs/stores/servlet/PolicyAdd?
   type=Rebate&name=5DollarRebate&plcyStoreId=-1
   &cmd_1=com.mycompany.mybusinesspolicycommands.CalculateRebateCmdImpl
   &startDate=2002-05-08%2000:00:00&endDate=2003-05-09%2000:00:00
   &commonProps=rebatecode_id%3D501&URL=aRedirectURL
```

Note that URL reserved characters must be replaced by their ASCII codes for input properties. As such, the typical = (equals) symbol is replaced with

"%3D", the & (ampersand) is replaced by "%26", and the space character is replaced by "%20". The date format used in the preceding example is yyyy-mm-dd hh:mm:ss, with ASCII code replacing URL reserved characters.

The following tables show the relevent columns of the affected database tables, after performing the updates.

Table 4. Updates made to the POLICY table

	Column name				
	POLICY _ID	POLICY NAME	POLICYTYPE _ID	STOREENT _ID	PROPERTIES
Sample data	301	5Dollar Rebate	Rebate	-1	rebatecode_id= 201

Note that it is also presumed that the start date and end date values are set to `null`.

Table 5. Updates made to the POLICYCMD table

	Column name		
	POLICY _ID	BUSINESS CMDCLASS	PROPERTIES
Sample data	301	com.mycompany. mybusinesspolicycommands. CalculateRebateCmdImpl	null

As a result, you now have a new business policy called "5DollarRebate" that is related to the CalculateRebateCmd business policy command.

Step 3: Creating the "RebateTC" term and condition

Creating the "RebateTC" term and condition requires that the following steps be performed:

1. Register the RebateTC term and condition in the database.
2. Update the B2BTrading.dtd file to reflect the new RebateTC.
3. Create a new enterprise bean for the RebateTC.
4. Update the WebSphere Commerce Accelerator to reflect the new RebateTC.

Registering the"RebateTC" term and condition in the database: When you are creating a new terms and condition object, you must update the database schema to include this object. The database tables that must be updated are TCTYPE and TCSUBTYPE.

The following SQL statement shows an example of how to register the RebateTC in the database:

```
insert into TCTYPE (TCTYPE_ID) values ('RebateTC');
insert into TCSUBTYPE (TCSUBTYPE_ID, TCTYPE_ID, ACCESSBEANNAME,
   DEPLOYCOMMAND)
values ('RebateTC, 'RebateTC ',
   'com.ibm.commerce.contract.objects.RebateTCAccessBean',
   null);
```

The following tables show an extract of the relevant columns in the TCTYPE and TCSUBTYPE tables.

Table 6. Updates made to the TCTYPE table

	Column name
	TCTYPE_ID
Sample data	RebateTC

Table 7. Updates made to the TCSUBTYPE table

	Column name			
	TCSUBTYPE _ID	TCTYPE _ID	ACCESSBEAN NAME	DEPLOY COMMAND
Sample data	RebateTC	RebateTC	com.ibm.commerce. contract.objects. RebateTCAccessBean	null

Updating the B2BTrading.dtd file to reflect the new RebateTC: To make the new term and condition available in contracts, you must update `B2BTrading.dtd` file to include the new term and condition. When updating this file, you must add the new term and condition to the TermCondition definition and then create a new element that describes the term and condition.

The text in bold shows an example of how to add the new RebateTC to the TermCondition definition:

```
<!ELEMENT TermCondition (TermConditionDescription?,Participant*,
CreateTime?,UpdateTime?,(PriceTC|ProductSetTC|ShippingTC|FulfillmentTC|
PaymentTC|ReturnTC|InvoiceTC|RightToBuyTC|ObligationToBuyTC|
PurchaseOrderTC|OrderApprovalTC|DisplayCustomizationTC|
OrderTC|RebateTC))>
```

Note that the line breaks are for display purposes only.

You must then add a new stanza that describes this term and condition to the file. The following shows an example for the RebateTC:

```
<!ELEMENT RebateTC (PolicyReference?)>
```

Creating a new enterprise bean for the RebateTC: You must create a new enterprise bean for the new RebateTC. This new bean should inherit from the WebSphere Commerce TermCondition bean.

A new enterprise bean for a term and condition is typically named after the subtype. Note that in this case, the term and condition subtype is the same as the term and condition type, and as such, the name of the bean is the same as the term and condition type.

The following table shows some of the general information about the new bean that must be created. For more details about the bean, including which methods to override, refer to "Creating a new CMP enterprise bean for the term and condition" on page 160.

Table 8.

Attribute	**Value**
Bean name	`RebateTC`
Inherit from	`TermCondition`
Package	`com.ibm.commerce.contract.objects`
Bean class	`RebateTCBean`
Remote interface	`RebateTC`
Home interface	`RebateTCHome`

Updating the WebSphere Commerce Accelerator to include the RebateTC: Once you have created new terms and conditions, you can update the WebSphere Commerce Accelerator so that it can be used to create new contracts that include those new terms and conditions. For information about how to update this tool, refer to "Updating the WebSphere Commerce Accelerator to use a new term and condition" on page 167.

Step 4: Creating a new contract

You must create a new contract that includes the "RebateTC" term and condition and that refers to the "5DollarRebate" business policy. You can use either the WebSphere Commerce Accelerator or XML to create a new contract. Each of these methods for creating new contracts are described in the WebSphere Commerce online help.

The following tables show the updates to the relevant columns of the TERMCOND and POLICYTC database tables, after the contract has been created.

Table 9. Updates made to the TERMCOND table

	Column name		
	TRADING _ID	TERMCOND _ID	TCSUBTYPE _ID
Sample data	25	901	RebateTC

Table 10. Updates made to the POLICYTC table

	Column name	
	POLICY _ID	TERMCOND_ID
Sample data	301	901

Step 5: Integrating the new business policy into the shopping flow

In this scenario, it is presumed that a new page is added to the store that allow customers to log on and claim their rebates. When the customer would click to claim the rebate, a command that invokes the new `RebatePolicyCmd` interface should be invoked. For example, there could be a new ClaimRebateCmd controller command that invokes the `RebatePolicyCmd`. The correct business policy is then found and (in this case) the “5DollarRebate” business policy is applied.

Part 3. Development environment

Chapter 8. Development tools and deployment

This chapter introduces the main development tools used for customizing a WebSphere Commerce application. It describes the process for deploying customized code from VisualAge for Java to a WebSphere Commerce Server. It also describes how to deploy code to Commerce Studio, so that you can take advantage of the customized code when developing JSP templates.

Development environment

The recommended development package for creating customized code to be used with WebSphere Commerce Business Edition is the WebSphere Commerce Studio, Business Developer Edition product. The recommended development package for creating customized code to be used with WebSphere Commerce Professional Edition is the WebSphere Commerce Studio Professional Developer Edition product. Both of these packages include all the tools you require to create customized code and perform Web development tasks.

Both the WebSphere Commerce Studio Business Developer Edition and the WebSphere Commerce Studio Professional Developer Edition provide the option of including a sample WebSphere Commerce store that is used in the WebSphere Test Environment component of VisualAge for Java. This simplifies the configuration of the development environment, since a developer is not required to use the tools of WebSphere Commerce to create a store and then move the store assets over to the development environment.

Another key component to the development environment is the repository of WebSphere Commerce code. This repository must be imported into the VisualAge for Java workspace, after the product installation is complete. Subsequent to importing this repository, the developer must perform some configuration steps related to the WebSphere Test Environment.

After all installation and configuration tasks are complete, the developer has a stand-alone development machine, on which customized WebSphere Commerce code can be created and tested. The developer is not required to install WebSphere Commerce on the development machine.

WebSphere Commerce Studio

WebSphere Commerce Studio Business Developer Edition and WebSphere Commerce Studio Professional Developer Edition both include VisualAge for Java, Enterprise Edition, Version 4.0. This edition of VisualAge for Java includes features such as robust tooling for developing and debugging advanced JSP templates that assist with developing store-front assets. It also includes enhanced integration with WebSphere Studio to allow the quick addition of content to these JSP templates, increasing productivity for programmers and Web developers. For the development of back-office application code, it includes support for Enterprise JavaBeans technology and connectivity features to support integration to other systems, such as CICS® TS, MQSeries, SAP R/3 and more. In addition, the integrated WebSphere Test Environment of VisualAge for Java, Enterprise Edition, allows developers to run WebSphere Commerce functions, without ever exiting VisualAge for Java. This means that testing of customized WebSphere Commerce code can occur without deploying the code to a WebSphere Commerce Server.

Note: The WebSphere Test Environment component of VisualAge for Java, Enterprise Edition, Version 4.0 runs applications within WebSphere Application Server V3.5.4. Note that both WebSphere Commerce Business Edition and WebSphere Commerce Professional Edition use WebSphere Application Server V4.0. The result of this difference in versions is that before deploying new or modified enterprise beans to a WebSphere Commerce instance running within WebSphere Application Server V4.0, you must generate deployed code outside of VisualAge for Java using the EJBDeploy tool that is shipped with WebSphere Application Server V4.0. In fact, this deployed code must be generated on a machine that has WebSphere Application Server V4.0 installed. Refer to "Information about EJB deployed code" on page 188 for more details.

In order to complete the tutorials described in Part 4, "Tutorials" on page 199, you must install either WebSphere Commerce Studio Business Developer Edition or WebSphere Commerce Studio Professional Developer Edition. Refer to the either the *WebSphere Commerce Studio Business Developer Edition Installation Guide* or the *WebSphere Commerce Studio Professional Developer Edition Installation Guide* for more information about installing Commerce Studio and configuring VisualAge for Java.

Features and functions of VisualAge for Java

This *Programmer's Guide* is not intended to teach you how to use VisualAge for Java. With respect to that product, it generally shows how to perform a task in VisualAge for Java more as a way of accomplishing a programming task for creating an e-commerce application for WebSphere Commerce. As such, if you are a new user of VisualAge for Java, you will start to learn how to perform some tasks in VisualAge for Java by performing the tutorials contained in this book. You must however, refer to VisualAge for Java documentation, tutorials and courses in order to become proficient in using this tool.

For example, the (Optional) Using the Debugger in VisualAge for Java tutorial topic provides a quick introduction to how you can use the Debugger to view the value of variables in a task command during code execution. The Debugger component of VisualAge for Java is a robust tool for debugging, and as such, you should refer to the VisualAge for Java documentation for complete details about its features and how to use them.

WebSphere Commerce code repository

In order to create customized code for a WebSphere Commerce application, you must import a repository of WebSphere Commerce code into the VisualAge for Java workspace. The repository is available on the WebSphere Commerce Business Edition Disk 2 CD and the WebSphere Commerce Professional Edition Disk 2 CD. The current repository (current at the time of publication) is called `WC_54.dat`.

Code deployment

When customizing your e-commerce application, you may do any of the following:

- Create new commands, data beans or entity beans
- Extend existing WebSphere Commerce entity beans
- Modify the logic of existing commands or data beans

When developing code within VisualAge for Java, you can test your code within the WebSphere Test Environment. At some point, you must deploy your code to a WebSphere Commerce Server outside of the development environment.

In the following sections, *target WebSphere Commerce Server* refers to the WebSphere Commerce Server to which you are deploying the customized code. In some testing scenarios, you may deploy to a WebSphere Commerce Server that is running on the same machine as VisualAge for Java. In other situations, the target WebSphere Commerce Server is on another machine, and may even be running on a different platform.

The following sections describe the high-level steps for deploying the various types of customized code. Use them to understand the steps involved in the deployment process and refer to Appendix B, "Deployment details" on page 335 for step-by-step instructions.

In addition to the following sections that describe deployment of customized code, if you have created new access control policies in your development environment, the same access control policies must be created on the target WebSphere Commerce Server. Refer to "Loading the access control policies for the new resources" on page 272 in the tutorials for an example of this procedure.

Information about EJB deployed code

It is important to recognize that the WebSphere Test Environment component of VisualAge for Java V4.0 uses WebSphere Application Server V3.5.4. In this version of WebSphere Application Server, enterprise beans are supported at the level of the Enterprise JavaBeans (EJB) V1.0 specification. As such, in order to run any enterprise beans within the WebSphere Test Environment, you use the tools of VisualAge for Java to generate deployed code for your enterprise beans that complies with the EJB V1.0 specification.

In contrast, both the WebSphere Commerce Business Edition and the WebSphere Commerce Professional Edition use WebSphere Application Server V4.0. WebSphere Application Server V4.0 supports the EJB V1.1 specification. As such, the deployed code for enterprise beans running within the WebSphere Test Environment is different than the deployed code for enterprise beans running within WebSphere Application Server V4.0.

The impact to development and deployment is as follows:

- To test enterprise beans within the WebSphere Test Environment, use the tools of VisualAge for Java to generate the deployed code that meets the requirements of the WebSphere Application Server V3.5.4 used in the WebSphere Test Environment. This code is generated by right-clicking the enterprise bean and selecting **Generate Deployed Code**. Note that this does not create a JAR file.
- To deploy enterprise beans to a WebSphere Application Server V4.0, you must do the following:
  1. Use the tools of VisualAge for Java to export the enterprise beans to what this document refers to as an *EJB 1.1 Export JAR file*. This JAR file packages the code into a format that is used by the EJBDeploy tool. This file is created by right-clicking the EJB group that contains the enterprise bean to be deployed and selecting **Export > EJB 1.1 JAR**. Note that when creating this JAR file, you must select the appropriate database type for the machine to which you will deploy your code.

2. Transfer this EJB 1.1 Export JAR file to a target server running WebSphere Application Server V4.0.
3. On the target server, run the EJBDeploy tool with the EJB 1.1 Export JAR file as input to create a new JAR file of deployed code that complies to the Enterprise JavaBeans V1.1 specification.

There are other steps involved in the deployment of enterprise beans. For more information refer to "Deployment of new entity beans" on page 190 and "Deployment of modified WebSphere Commerce public entity beans" on page 192. For more information about using the EJBDeploy tool, refer to "Generating deployed code" on page 346.

Deployment of new commands and data beans

When creating new commands and data beans, you should place them in a package that is named appropriately for your application. For example, you could create a new package by the name of `com.mycompany.mycommands` in which you keep your new commands. This package must be stored within a project that is separate from the WebSphere Commerce projects (IBM WC Commerce Server and IBM WC Enterprise Beans). For example, you could create a new project called `My Project`. Careful organization of code is required, to ensure that deployment is successful.

With all of your new commands and data beans stored in your own project, deployment consists of the following high-level steps:

1. Using the development machine, create a JAR file for the project. From the Project page in VisualAge for Java, use the tools to export the project to a JAR file. Name the JAR file appropriately for your code, for example, `CustomCommands.jar`. In addition, you must rejar the JAR file using tools outside of VisualAge for Java to ensure that all required naming information is included for deployment to WebSphere Application Server. Refer to "JAR files for customized commands and data beans" on page 335 for more information.
2. Copy the JAR file, JSP templates and any other store assets into the appropriate directory on the target WebSphere Commerce Server. Refer to "Storing assets on the target WebSphere Commerce Server" on page 342 for more information.
3. Register the new command in the command registry on the target WebSphere Commerce Server. Refer to "Command registration framework" on page 26 for more information.
4. Stop and restart the WebSphere Commerce enterprise application, using the WebSphere Application Server Administrator's Console. Refer to the appropriate *WebSphere Commerce Installation Guide* for more information about starting and stopping this application.

Deployment of new entity beans

When creating new entity beans, you must create them in an EJB group that is separate from the EJB groups that contain the WebSphere Commerce entity beans. You must also use your own project and ensure that this project does not contain the code for any commands or data beans. For example, you could create a new EJB group called `MyEntityBeans` and a project by the name of `MyEntityBeansProject`. If the project contains code other than the code for new entity beans, deployment may not be successful.

Once you are satisfied with the way your entity bean functions within the WebSphere Test Environment, you must deploy it. The following information provides an overview of the deployment steps:

1. Using the development machine, create a new EJB 1.1 Export JAR file for your new EJB group. From the EJB page in VisualAge for Java, right-click the new EJB group and select to export the code to an EJB 1.1 JAR file. Name the file appropriately for your code, for example, `MyEntityBeans_DT.jar`. Refer to “Creating JAR files for new entity beans” on page 337 for more information.
2. Create a new implementation JAR file for the new EJB project. To create this JAR file, select the Project page, then right-click the new EJB project and select to export the code to a JAR file. Name the file appropriately for your code, for example, `MyEntityBeansImpl.jar`. In addition, you must rejar the implementation JAR file using tools outside of VisualAge for Java to ensure that all required naming information is included for deployment to WebSphere Application Server. Refer to “Creating JAR files for new entity beans” on page 337 for more information.
3. Copy the JAR files into the appropriate directory on the target WebSphere Commerce Server. Refer to “Storing assets on the target WebSphere Commerce Server” on page 342 for more information.
4. On the target WebSphere Commerce Server, use the EJB Deploy Tool provided by WebSphere Application Server to generate the deployed code for the new enterprise beans. This tool takes the JAR file created in step 1 as input and it creates the corresponding JAR file containing the deployed code for all enterprise beans in your EJB group. Refer to “Generating deployed code” on page 346 for more information.
5. On the target WebSphere Commerce Server, modify the transaction isolation level of the enterprise beans that are contained in the JAR file of deployed code. Use the `modifyIsolationLevel` command line utility provided by WebSphere Commerce to set the transaction isolation level to the appropriate level for your database type. Refer to “Modifying transaction isolation level of entity beans” on page 349 for more information.
6. On the target WebSphere Commerce Server, load the access control policies for any new resources that you have created. Use the WebSphere

Commerce `acpload` and `acpnlsload` commands to load the policy information. Refer to the “Loading the access control policies for the new resources” on page 272 section in the Chapter 9, “Tutorial: Creating new business logic” tutorial for an example of loading access control policies for new resources.

7. On the target WebSphere Commerce Server, export the current WebSphere Commerce enterprise application from WebSphere Application Server. After the export has completed, an .ear file is created that contains the entire application. To export the current application, open the WebSphere Application Server Administration Console, select the WebSphere Commerce enterprise application and then select the export option. Upon completion, a `WC_Enterprise_App_`*`instanceName`*`.ear` file is created. Refer to “Exporting the current WebSphere Commerce enterprise application” on page 350 for more information.
8. On the target WebSphere Commerce Server, export the configuration information for the enterprise beans that are contained in the current WebSphere Commerce enterprise application running within WebSphere Application Server. This information is exported to an XML file by using the `-export` option of the `XMLConfig` command line utility that is provided by WebSphere Application Server. The generated XML file (referred to here as the `OutputFile.xml` file) contains a stanza of configuration information for each enterprise bean contained in the enterprise application. You must then add a new stanza to this file for each new enterprise bean that you are deploying. Refer to “Exporting configuration information for enterprise beans” on page 351 for more information.
9. Add the your new enterprise bean or beans into your enterprise application using the Application Assembly Tool provided by WebSphere Application Server. Using this tool, you can open the `WC_Enterprise_App_`*`instanceName`*`.ear` for the current application and then import any new enterprise beans into the application. You also set the class path for the new enterprise bean or beans to include any dependent JAR files and add the implementation JAR file as a file to the application. Finally, you use this tool to configure WebSphere Application Server security for methods that are contained in your enteprise bean or beans. After these steps are complete, save the application and a new `.ear` file is created for your enterprise application. Refer to “Assembling new enterprise beans into an enterprise application” on page 357 for more information.
10. Import the new enterprise application into WebSphere Application Server. This step is comprised of the following three subtasks:
    a. Using the WebSphere Application Server Administration Console, stop and remove the original WebSphere Commerce enterprise application. Refer to “Stopping and removing an enterprise application” on page 366 for more information.

b. Using the `-import` option of the `XMLConfig` command line utility to import the new enterprise application into WebSphere Application Server. Refer to "Importing an enterprise application" on page 366 for more information.
c. Using the WebSphere Application Server Administration Console, refresh the view to display the new enterprise application and then start the new application. Refer to "Starting an enterprise application" on page 368 for more information.

Deployment of extensions to existing commands and data beans

The method for extending existing commands depends upon the type of modification required. The methods for extension are explained in "Customization of existing commands" on page 139. In general, modification of existing logic involves creating a new class that inherits from the class that requires customization. Override methods of the superclass as required, to replace or modify logic.

When customizing data beans, you also create a new class that extends an existing data bean. Within the new class, make the required modifications.

When creating these new classes, ensure that they are stored within one of your own packages, which is itself stored within one of your own projects.

Since extensions are effectively handled by subclassing, deployment for extensions to commands and data beans is the same as deployment for new commands and data beans. For more information, refer to "Deployment of new commands and data beans" on page 189.

Deployment of modified WebSphere Commerce public entity beans

When you modify a WebSphere Commerce public enterprise bean you modify WebSphere Commerce code. The deployment technique for customized entity beans is, therefore, slightly different than that used for new entity beans. The following provides an overview of the deployment steps:

1. Create an EJB 1.1 Export JAR file for the WebSphere Commerce EJB group that contains the modified entity bean. With the EJB tab in the VisualAge for Java Workbench selected, select the appropriate EJB group and then select to export that group to an EJB 1.1 Export JAR file. When naming the JAR file, you can use the `Cust_`*`EJBGroupName`*`-ejb_DT.jar` naming convention where *`EJBGroupName`* is the name of the EJB group that has been modidfied and the _DT suffix is appended to the end of the name of the JAR file. For example, when naming the JAR file for the WCSUser EJB group, you could name the file as `Cust_WCSUser-ejb_DT.jar`. Note that the _DT suffix is simply used as a reminder that you must later pass this JAR file to the EJBDeploy Tool to generate the deployed code for the beans contained in the EJB group. Refer to "Creating JAR files for customized WebSphere Commerce entity beans" on page 339 for more information.

2. Create a new client JAR file that contains the client code for all of the WebSphere Commerce EJB groups. With all of the WebSphere Commerce EJB groups selected (all names begin with `WCS`), select to export to a client JAR file. Refer to "Creating JAR files for customized WebSphere Commerce entity beans" on page 339 for more information.
3. Copy the JAR files into the appropriate directory on the target WebSphere Commerce Server. Refer to "Storing assets on the target WebSphere Commerce Server" on page 342 for more information.
4. On the target WebSphere Commerce Server, use the EJBDeploy Tool provided by WebSphere Application Server to generate the deployed code for the enterprise beans contained in the EJB group that you are deploying. This tool takes the JAR file created in step 1 as input and it creates the corresponding JAR file containing the deployed code for all enterprise beans in your EJB group. Refer to "Generating deployed code" on page 346 for more information.
5. On the target WebSphere Commerce Server, modify the transaction isolation level of the enterprise beans that are contained in the JAR file of deployed code. Use the `modifyIsolationLevel` command line utility provided by WebSphere Commerce to set the transaction isolation level to the appropriate level for your database type. Refer to "Modifying transaction isolation level of entity beans" on page 349 for more information.
6. On the target WebSphere Commerce Server, export the current WebSphere Commerce enterprise application from WebSphere Application Server. After the export has completed, an .ear file is created that contains the entire application. To export the current application, open the WebSphere Application Server Administration Console, select the WebSphere Commerce enterprise application and then select the export option. Upon completion, a `WC_Enterprise_App_`*`instanceName`*`.ear` file is created. Refer to "Exporting the current WebSphere Commerce enterprise application" on page 350 for more information.
7. On the target WebSphere Commerce Server, export the configuration information for the enterprise beans that are contained in the current WebSphere Commerce enterprise application running within WebSphere Application Server. This information is exported to an XML file by using the `-export` option of the `XMLConfig` command line utility that is provided by WebSphere Application Server. The generated XML file (referred to here as the `OutputFile.xml` file) contains a stanza of configuration information for each enterprise bean contained in the enterprise application. Refer to "Exporting configuration information for enterprise beans" on page 351 for more information. Within this file, you must modify the stanza describing the enterprise bean that you have modified to point to your new `Cust_`*`EJBGroupName`*`-ejb.jar` file.

8. On the target WebSphere Commerce Server, use the Application Assembly Tool to integrate the modified EJB group into the enterprise applicacation. Using this tool, you open the `.ear` file for the current application. Once it is open, you perform the following tasks:
   a. Make note of the class path information for the original version of the EJB group that you have modified. For example, if you have modified the User bean in the WCSUser EJB group, then you copy the class path information for the WCSUser group into a text file.
   b. Delete the original version of the EJB group that you have modified (for example, delete the WCSUser group).
   c. Import the new version of the EJB group that you have modified.
   d. Apply the original class path information to the EJB group that you just imported.
   e. Configure WebSphere Application Server security for the methods contained in all of the enterprise beans in the modified EJB group.
   f. Save the application into a new `.ear`

   Refer to "Assembling modified enterprise beans into an enterprise application" on page 361 for more information.
9. Import the new enterprise application into WebSphere Application Server. This step is comprised of the following three subtasks:
   a. Using the WebSphere Application Server Administration Console, stop and remove the original WebSphere Commerce enterprise application. Refer to "Stopping and removing an enterprise application" on page 366 for more information.
   b. Using the `-import` option of the `XMLConfig` command line utility to import the new enterprise application into WebSphere Application Server. Refer to "Importing an enterprise application" on page 366 for more information.
   c. Using the WebSphere Application Server Administration Console, refresh the view to display the new enterprise application and then start the new application. Refer to "Starting an enterprise application" on page 368 for more information.

Deployment of new data beans for use in Commerce Studio

If you are using Commerce Studio for developing your JSP templates, you must deploy your new data beans to Commerce Studio. In particular, you must create a JAR file for the data beans.

Create this JAR file by right-clicking your package of data beans and selecting to export them to a JAR file. In the options, select to include the following:

- **class**
- **resource**

- **beans**

In addition, select **Select referenced types and resources** to include the commands and resources that are required by the data beans.

Once you have exported the JAR file, you must update the class path for Commerce Studio to include this JAR file.

When you need to modify an existing WebSphere Commerce data bean, you must create a new data bean that extends the data bean which requires customization. Therefore, you are in effect creating a new data bean and deployment is as described in this section.

Deployment of customized public entity beans for use in Commerce Studio

If you modify any of the public entity beans and use Commerce Studio for JSP template development, you must create a new client JAR file for all of the public entity beans and modify the class path for Commerce Studio to include the new JAR file name before the original JAR file name. Commerce Studio uses the client JAR file when data beans are used in the Page Designer tool.

To create this JAR file, use the tools in VisualAge for Java. From the EJB page in VisualAge for Java, select *all* of the WebSphere Commerce EJB groups (not just the ones you modified) and select to export them to a client JAR file. After the export has completed, ensure that you specify this new JAR file at the beginning of the class path for Commerce Studio.

Log files

Throughout the stages of product installation, code development and code deployment, log files are generated. This section lists some of the log files that you might consult, throughout these stages.

Commerce Studio log files
Commerce Studio builds log files during the installation process. To access these logs, open the following file:

```
C:\Winnt\WCStudioInstall.log
```

Logs for Commerce Studio configuration for the WebSphere Test Environment
A number of configuration steps for the WebSphere Test Environment are performed during the installation of WebSphere Commerce Studio, if you select that you want to do back-end development tasks. For example, if you select to include a sample store that runs within the WebSphere Test Environment component of VisualAge for Java, logs files related to the mass load process and database creation are created. To access these logs, navigate to the following directory:

`drive:\WebSphere\CommerceServerDev\instances\instance_name\logs`

Log files from running WebSphere Commerce components within the WebSphere Test Environment

When running a store, or testing an individual component within the WebSphere Test Environment, a trace file and message file may be generated. The default location for these files is in the following directory:

`drive:\WebSphere\CommerceServerDev\instances\instance_name\logs`

Logs from running modifyIsolationLevel command

When deploying enterprise beans, you use the modifyIsolationLevel command to modify the transaction isolation level of each enterprise bean in the JAR file. The log files generated are stored in the log file that you specify when running the command. Refer to "Modifying transaction isolation level of entity beans" on page 349 for more information.

Logging within VisualAge for Java

When running code within the WebSphere Test Environment, the Console window runs as an active log. In addition, you can modify the level of tracing for an EJB server to increase the amount of information that can be found within the Console window. This information is specified as the trace level, within the properties of the EJB server.

Test payment method

The InFashion sample store running within the WebSphere Test Environment uses a test payment method by default. This test payment method has been included so that you can complete the shopping flow within WebSphere Test Environment, without requiring a call out to a remote Payment Manager. Orders placed using this payment method have a status of 'M' (indicating that payment has been initiated and is awaiting processing).

This test payment method only lets you complete a purchase, it does not enable orders submitted with this payment method to be available for further processing. As such, the test payment method should only be used within the WebSphere Test Environment.

The implementation classes for payment related commands are types of business policy commands. As such, the selection of which implementation class to use is governed by a business policy. By default, the InFashion sample store running with the WebSphere Test Environment includes a business policy (TestPaymentMethod policy) that uses the following implementations of payment related commands:

- `com.ibm.commerce.payment.commands.DoPayment`**`Test`**`CmdImpl`

- `com.ibm.commerce.payment.commands.CheckPaymentAccept`**`Test`**`CmdImpl`
- `com.ibm.commerce.payment.commands.DoCancel`**`Test`**`CmdImpl`
- `com.ibm.commerce.payment.commands.DoDeposit`**`Test`**`CmdImpl`
- `com.ibm.commerce.payment.commands.DoRefund`**`Test`**`CmdImpl`

It must be stressed that these commands are provided for testing the checkout process only. While the preceding suite of commands is provided for completeness purposes, the DoDepositTestCmdImpl is a command stub that throws an exception if it is called. Do not attempt to use any order management functions with these orders. If you require the ability to test these other functions, you can configure your WebSphere Test Environment to use a remote Payment Manager.

It is very important that when you deploy your code to the target WebSphere Commerce Server, you should not copy the business policy related to the test payment method from your development to target machine. Additionally, you should not register the commands related to the test payment method on the target WebSphere Commerce Server.

Refer to the “Configuring VisualAge for Java” chapter in the *WebSphere Commerce Business Edition Installation Guide* or in the *WebSphere Commerce Professional Edition Installation Guide* for information about disabling the test payment method.

Using a remote Payment Manager

If your development work includes working with orders once they have been placed, for example, making a modification to a step in the order management process, you must configure the store running in the WebSphere Test Environment to use a remote Payment Manager. Refer to the *WebSphere Commerce Studio, Business Developer Edition Installation Guide* for the detailed configuration steps. That document also provides information about removing orders placed with the test payment method from your database.

Part 4. Tutorials

This section contains tutorials to help you get familiar with customizing your e-commerce application. The following tutorials are provided:

- Creating new business logic
  This tutorial demonstrates the development process for creating new business logic. It includes the following sub-tasks:
  - Preparing the sample project
  - Writing new commands
  - Creating a new data bean and getting properties from request
  - Using entity beans
  - Creating a new enterprise bean
- Updating existing business logic
  This tutorial shows the development process for updating existing WebSphere Commerce business logic. It includes the following sub-tasks:
  - Extending an existing WebSphere Commerce controller command
  - Modifying an existing WebSphere Commerce public entity bean
  - Creating a new task command that extends an existing WebSphere Commerce task command.

The tutorials contain sections of code that need to be entered into VisualAge for Java. It is recommended that you obtain a PDF version of this document from the following Web sites:

▶ Business

```
http://www.ibm.com/software/webservers/commerce/wc_be
   /lit-tech-general.html
```

▶ Professional

```
http://www.ibm.com/software/webservers/commerce/wc_pe
   /lit-tech-general.html
```

You can cut and paste the code snips from the PDF version of the Programmer's Guide.

Chapter 9. Tutorial: Creating new business logic

Tutorial environment

The tutorials contained in this book require that you have either WebSphere Commerce Business Edition V5.4 or WebSphere Commerce Professional Edition V5.4 installed. In addition, when you install the product you must select to install the following options:

- **Develop store-front assets using WebSphere Studio**
- **Develop store backend logic using VisualAge for Java**
- **Create database**
- **Include sample store**

Follow the instructions contained in the either the *WebSphere Commerce Studio, Business Developer Edition Installation Guide* or the *WebSphere Commerce Studio Professional Developer Edition Installation Guide*for complete installation and configuration information.

Before starting the tutorials, you must be able to run the sample store within the WebSphere Test Environment and complete a purchase in the store.

Tutorial code deployment steps

The tutorials contained in this book include optional steps that describe how to deploy the customized code to a target WebSphere Commerce Server. It is presumed that the target WebSphere Commerce Server is running on either Windows NT or Windows 2000 on a machine that is separate from your development environment. Note that if you are using WebSphere Commerce Studio Business Developer Edition as your development environment, you should deploy to WebSphere Commerce Business Edition. If you are using WebSphere Commerce Studio Professional Developer Edition as your development environment, you should deploy to WebSphere Commerce Professional Edition.

In addition, on the target WebSphere Commerce Server you must have published a store based on the InFashion sample store. Within that store, you must be able to complete a purchase. You can use either a remote or local Payment Manager for your payment processing.

If you are deploying to a target WebSphere Commerce Server that is either on the same machine as your development environment, or that is running on a different operating system, refer to Chapter 8, “Development tools and deployment” on page 185 and to Appendix B, “Deployment details” on page 335 for the appropriate deployment information.

Preparing the sample project

In this section, you import the `WC_SAMPLE_54.dat` repository that contains the starting point for the “Creating new business logic” tutorial.

To prepare your environment, do the following:

1. Ensure that you are using the `WC_54.dat` repository of WebSphere Commerce code. This is available on either the WebSphere Commerce Business Edition V5.4 Disk 2 CD or the WebSphere Commerce Professional Edition V5.4 Disk 2 CD.
2. Import the sample project to your workspace, by doing the following
   a. Insert the following CD into the CD drive on your development machine:
      - Business WebSphere Commerce Business Edition, V5.4 Disk 2
      - Professional WebSphere Commerce Professional Edition, V5.4 Disk 2
   b. Open VisualAge for Java.
   c. From the **File** menu, select **Import**
      The Import SmartGuide opens.
   d. Select to import a **Repository** and click **Next**.
   e. In the Import from another repository window, do the following:
      1) Select **Local repository**.
      2) In the **Repository name** field, enter `*CD_drive*:\repository\samples\programguide\WC_SAMPLE_54.dat` where *`CD_drive`* is the CD drive.
      3) Select **Projects** and click **Details**. Select the_**WCSamples** project. Select the **WC Sample 5.4** version and click **OK**.
      4) Ensure that **Add most recent project edition to workspace** is selected.
      5) Click **Finish** to begin importing.
         Importing the project may take several minutes.
3. Ensure that the workspace owner is set to WCS Developer, by doing the following:
   a. From the **Workspace** menu, select **Change Workspace** Owner.
   b. Select **WCS Developer** and click **OK**.
4. Copy the JSP templates for the exercises into the appropriate directory, so they can be used in the WebSphere Test Environment. To copy these files, do the following:

a. Locate the Sample.jsp and Sample_All.jsp files in the following directory

   ```
   CD_drive:\repository\samples\programguide\
   ```

   where *CD_drive* is the CD drive into the following directory:

b. Copy these two files into the following directory:

   ```
   vaj_drive:\VAJava\ide\project_resources\IBM WebSphere Test Environment
        \hosts\default_host\default_app\web
   ```

   where *vaj_drive* is the drive onto which you installed VisualAge for Java.

5. Copy the access control policy files into the appropriate directory. These files are used to load new access control policies for the new resources that you create during the tutorials. To copy these files, do the following:
   a. Switch to the following directory:

      ```
      CD_drive:\repository\samples\programguide\
      ```

   b. Locate the following files in that directory:
      - `SampleCmdACPolicy.xml`
        This XML file contains the access control policy that is used when you create a new controller command.
      - `SampleACPolicy.xml`
        This XML file contains the access control policy that is used when you create a new enterprise bean.
      - `SampleACPolicy_`*`locale`*`.xml`
        where *locale* is the language identifier.This XML file contains the access control policy description.
   c. Copy the preceding files into the following directory:
      drive`:\WebSphere\CommerceServerDev\xml\policies\xml`
      where *drive* is the drive on which you installed WebSphere Commerce Studio.
6. Test your environment to ensure that you are ready to start the tutorials, by doing the following:
   a. Start the persistent name server, as described on page 333.
   b. Start the EJB server, as described on page 334.
   c. Start the servlet engine, as described on page 334
   d. Open a browser and enter the following URL:

      ```
      http://localhost:8080/webapp/wcs/stores/servlet/StoreCatalogDisplay?
         storeId=store_ID&catalogId=catalog_Id&langId=-1
      ```

      where *store_ID* is the identifier for your sample store (`10001` is an example value) and *catalog_Id* is the identifier for your sample store's catalog (`10001` is an example value).

When the home page of the sample store is displayed, select a product and purchase it. You must be able to reach the "Order confirmation" page in the shopping flow.

To verify the storeId value for your store, you can refer to the STOREENT table.

e. Close all browsers and stop the WebSphere Test Environment.

You are now ready to begin the tutorials.

Writing commands

Write a controller command

This section shows you how to write a new controller command. Upon completion of this exercise, you will have a new command, called *MyNewControllerCmd*. This command is used by all stores and each store uses the same implementation of the command.

There are three basic steps when creating a new controller command:

1. Register the command in the command registry framework.
2. Create the interface for the command.
3. Create the implementation class for the command.

For more information about commands, refer to "Command design pattern" on page 20.

Before starting this tutorial

You should have already completed the steps in "Preparing the sample project" on page 202.

To write your new command, do the following:

1. The first step in writing a controller command is to establish a name for your command. In this example, the command is called *MyNewControllerCmd*.
2. The command must be registered in the command registry framework. The registration process for the new controller command involves creating an entry in the URLREG table.

   DB2 If you are using a DB2 database, do the following to register your command:

   a. Open the DB2 Command Center (**Start > Programs >IBM DB2 > Command Center**).

b. From the **Tools** menu, select **Tools Settings**.

c. Select the **Use statement termination character** checkbox and ensure the character specified is a semicolon (;)

d. With the Script tab selected, create the required entry in the URLREG table, by entering the following information in the script window:

```
connect to your_database_name;
insert into URLREG (URL, STOREENT_ID, INTERFACENAME, HTTPS,
    DESCRIPTION, AUTHENTICATED) values ('MyNewControllerCmd',0,
    'com.ibm.commerce.sample.commands.MyNewControllerCmd',0,
    'This is a new controller command for test/education purposes.',
    null)
```

where *your_database_name* is the name of your database and click the Execute icon.
This command is used by all merchants (indicated by the 0 value for STOREENT_ID).

Oracle If you are using an Oracle database, do the following to register your command:

a. Open the Oracle SQL Plus command window (**Start > Programs > Oracle > Application Development > SQL Plus**).

b. In the **User Name** field, enter your Oracle user name.

c. In the **Password** field, enter your Oracle password.

d. In the **Host String** field, enter your connect string.

e. In the SQL Plus window, enter the following to create the required entry in the URLREG table:

```
insert into URLREG (URL, STOREENT_ID, INTERFACENAME, HTTPS,
    DESCRIPTION, AUTHENTICATED) values ('MyNewControllerCmd',0,
    'com.ibm.commerce.sample.commands.MyNewControllerCmd',0,
    'This is a new controller command for test/education purposes.',
    null);
```

and press Enter to run the SQL statement.

f. Enter the following to commit your database changes:

```
commit;
```

and press Enter to run the SQL statement.

Note: For simplicity in this exercise, the new command has only one implementation class, therefore the same implementation class is used by all stores. This implementation class is specified directly in code for the interface. As such, there is no need to register the mapping between the interface and the implementation class in the

CMDREG table. Outside of a tutorial environment, you should register the controller command in the CMDREG table, as well as the URLREG table.

3. Controller commands must return a view. The new controller command that you will create returns the SampleViewTask view. You must register the SampleViewTask view in the VIEWREG table.

   DB2 If you are using a DB2 database, do the following to register the view:

   a. Create an entry in the VIEWREG table, by entering the following in the script window (note you may need to clear the previous SQL statement from the window first):

      ```
      insert into VIEWREG (VIEWNAME, DEVICEFMT_ID, STOREENT_ID, INTERFACENAME,
         CLASSNAME, PROPERTIES, DESCRIPTION, HTTPS, LASTUPDATE)
      values ('SampleViewTask',-1, 0,
          'com.ibm.commerce.command.ForwardViewCommand',
          'com.ibm.commerce.command.HttpForwardViewCommandImpl',
          'docname=Sample.jsp','This is a sample view for the
          Bonus Point exercise', 0, null)
      ```

      and click the Execute icon.

   Oracle If you are using an Oracle database, do the following to register your view in the database:

   a. In the SQL Plus window, enter the following SQL statement:

      ```
      insert into VIEWREG (VIEWNAME, DEVICEFMT_ID, STOREENT_ID, INTERFACENAME,
         CLASSNAME, PROPERTIES, DESCRIPTION, HTTPS, LASTUPDATE)
      values ('SampleViewTask',-1, 0,
         'com.ibm.commerce.command.ForwardViewCommand',
         'com.ibm.commerce.command.HttpForwardViewCommandImpl',
         'docname=Sample.jsp','This is a sample view for the
         Bonus Point exercise', 0, null);
      ```

      and press Enter to run the SQL statement.

   b. Enter the following to commit your database changes:

      ```
      commit;
      ```

      and press Enter to run the SQL statement.

4. In the VisualAge for Java Workbench window, expand the **_WCSamples** project.

5. Expand the **com.ibm.commerce.sample.commands** package, right-click the **MyNewControllerCmd** interface, and select **Add > Field**.
   The Create Field SmartGuide opens.

6. Create a field that specifies the default implementation class for the interface, by doing the following:

   a. In the **Field Name** field, enter `defaultCommandClassName`.

b. From the **Field Type** drop-down list, select `String`.

c. In the Initial Value field, enter `"com.ibm.commerce.sample.commands.MyNewControllerCmdImpl"`. (Ensure that you include the double-quotation marks.)

d. Click **Finish**.
The code for the new field is generated.

At any point in this tutorial when you override a field or method that exists in the superclass, a warning indicating that the new field or method will hide an inherited field or method may be displayed. If so, click **Yes** to continue.

---

7. Expand the **MyNewControllerCmdImpl** class and select its **performExecute** method to view its source code.
8. In the source code for the performExecute method, uncomment Section 1 and Section 5. Section 1 introduces the following code into the method:

```
// Create a new TypedProperties for output.
  TypedProperty rspProp = new TypedProperty();
```

Section 5 introduces the following code into the method:

```
// see how controller command call a JSP

 rspProp.put(ECConstants.EC_VIEWTASKNAME, "SampleViewTask");
 setResponseProperties(rspProp);
```

Save your work (**Ctrl + S**). The preceding code snippet sets the view name to be returned by the controller command.

9. Command-level access control must be specified for this new controller command. In this case, command-level access control policy will specify that all users are allowed to execute the command. This policy is specified in the `SampleCmdACPolicy.xml` file. To load the new access control policy, do the following:

a. At a command prompt, switch to the following directory: *`drive`*`:\WebSphere\CommerceServerDev\bin`

b. You must issue the `acpload` command, which has the following form:

```
acpload db_name db_user db_password inputXMLFile
```

where

- *`db_name`* is the name of your database
- *`db_user`* is your database user name
- *`db_password`* is your database password
- *`inputXMLFile`* is the name of the XML file containing the policy.

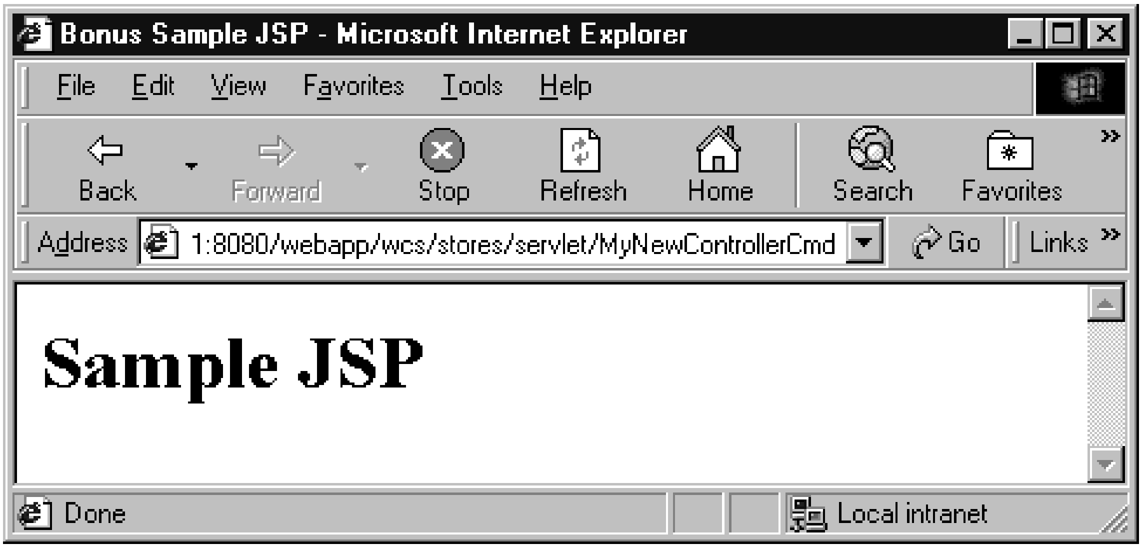
For example, you may issue the following command:

```
acpload VAJ_Demo user password SampleCmdACPolicy.xml
```

10. Add the new `_WCSamples` project to the class path for the servlet engine, by doing the following:
    a. From the **Workspace** menu, select **Tools > WebSphere Test Environment**.
       The WebSphere Test Environment Control Center opens.
    b. Click **Servlet Engine** and then **Edit Class Path**
       In the Servlet Engine Class Path window, click **Select All**, then **OK**.
11. Test your new command, by doing the following:
    a. Start the persistent name server, as described on page 333.
    b. Start the EJB server, as described on page 334.
    c. Start the servlet engine, as described on page 334
    d. Open a browser and enter the following URL:

       ```
       http://localhost:8080/webapp/wcs/stores/servlet/
         MyNewControllerCmd
       ```

       After a few minutes the browser displays a page that shows "Sample JSP", as shown below:



Sample JSP

Figure 31.

12. Create a version of your code in its current state. This allows you to restore your code to this state at any point in time. To create the version, do the following:

a. Select the **com.ibm.commerce.sample.commands** and **com.ibm.commerce.sample.databeans** packages (hold the `Ctrl` key while highlighting to select multiple packages), right-click and select **Manage > Create Open Edition**.
b. Right-click the **_WCSamples** project and select **Manage > Create Open Edition**.
c. Right-click the **_WCSamples** project again and select **Manage > Version**.
   The Versioning Selected Items window opens.
d. Select the **One Name** radio button, enter `mySample 1.2 Completed` and click **OK**.

You have now added a new command and tested it (briefly) in the integrated testing environment.

Modify MyNewControllerCmd

In the previous section, you created MyNewControllerCmd. In this exercise, you examine the contents of your new command more closely to develop a better understanding of how to write your own, customized controller command.

Begin by examining the structure of the code that you have created. The MyNewControllerCmd interface extends the ControllerCommand interface. It also defines the implementation class to use as a default. This class is used when the command is either not registered in the CMDREG table, or when an implementation class is not specified in that table.

You also created the MyNewControllerCmdImpl class. The implementation class is the class that will eventually contain the business logic (or call out to task commands to perform individual business tasks) that you want to implement. Your implementation class contains the following methods:

Methods in MyNewControllerCmdImpl	**Description**
MyNewControllerCmdImpl()	The constructor method.
validateParameters()	Used for server-side validation of the command's input parameters.
isGeneric()	Determines whether or not a generic user can call the command.
isRetriable()	Determines whether or not the command will be retried after a database rollback.
performExecute()	Contains the business logic for your command.

The following sections show more details about how to update your new controller command.

Pass variables to the JSP template

In this section, you modify MyNewControllerCmd to pass variables to the JSP template. In order to display the variables, you must use a new data bean, called DataBeanSampleBean. To enable the display of variables in your JSP template, do the following:

1. In the VisualAge for Java Workbench window, expand the **_WCSamples** project.
2. Expand the **com.ibm.commerce.sample.commands** package, then the **MyNewControllerCmdImpl** class and select its **performExecute** method.
3. In the source code for the performExecute method, uncomment Section 2. This introduces the following code into the method:

   ```
   // see how controller command pass in variables to JSP

    // Add additional parameters in controller command
    // to rspProp for response
    //
    rspProp.put("ControllerParm1", "Hello world");
    rspProp.put("ControllerParm2", "Have a nice day!");
   ```

   The above code snippet creates two new parameters that are put into the properties to be passed to the JSP template. Save your work (**Ctrl + S**).
4. The DataBeanSampleBean data bean is used by the JSP template to display the variables. The bean has been created for you, but you need to modify the source code as follows:
   a. Expand the **com.ibm.commerce.sample.databeans** package.
   b. Expand the **DataBeanSampleBean** class and then select the **setRequestProperties** method, to view its source code.
   c. In the source code for the setRequestProperties method, uncomment Section 1. This introduces the following code into the method:

      ```
      // copy input TypedProperties to local

       requestProperties = aParam;
      ```

      Save your work. The preceding code snippet copies the values from aParam (defined in the super class) locally. This is used to get properties from the request object.
5. Update the Sample.jsp file to use the new data bean and display the variables, by doing the following:
   a. Using a text editor, open the `Sample.jsp` and `Sample_All.jsp` files. These files are located in the following directory: `*vaj_drive*:\VAJava\ide\project_resources\IBM WebSphere Test Environment\hosts\default_host \default_app\web`

b. Copy Section 1 of the code from Sample_All.jsp into Sample.jsp between the `<!-- SECTION 1 -->` and `<!-- END OF SECTION 1 -->` markers. This introduces the following code to the JSP template:

```
<!-- SECTION 1 -->
<%
DataBeanSampleBean testBean = new DataBeanSampleBean ();
com.ibm.commerce.beans.DataBeanManager.activate (testBean, request);
%>
<!-- END OF SECTION 1 -->
```

This section of code instantiates the data bean.

c. Copy Section 2 from Sample_All.jsp into Sample.jsp between the `<!-- SECTION 2 -->` and `<!-- END OF SECTION 2 -->` markers. This introduces the following code into the JSP template:

```
<!-- SECTION 2 -->
<%
TypedProperty prop = testBean.getRequestProperties();

 out.print("<B>List of name value pairs in TypedProperties
            object</B><P>");

 // convert from request Properties to query string
 for (Enumeration pns = prop.keys(); pns.hasMoreElements();) {
     String paramName = (String) pns.nextElement();
      // do not add the url parameter to the query string
      Object val = prop.get(paramName,null);
     if (val != null) {
          if (val.getClass().isArray()) {
           // flatten the array
          String[] oarray = (String[]) val;
          int len = java.lang.reflect.Array.getLength(val);
          for (int i = 0; i < len; i++) {
          out.print(paramName + "[" + i + "]=" + oarray[i] + "<br>");
          }
     } else {
     // assume that it is a String
     out.print(paramName + "=" + val.toString() + "<br>");
     }
   }
 }
%>
<P>
<!-- END OF SECTION 2 -->
```

Save this file.
This section of code uses the getRequestProperties method of the testBean data bean object. It cycles through the properties and displays them in the browser.

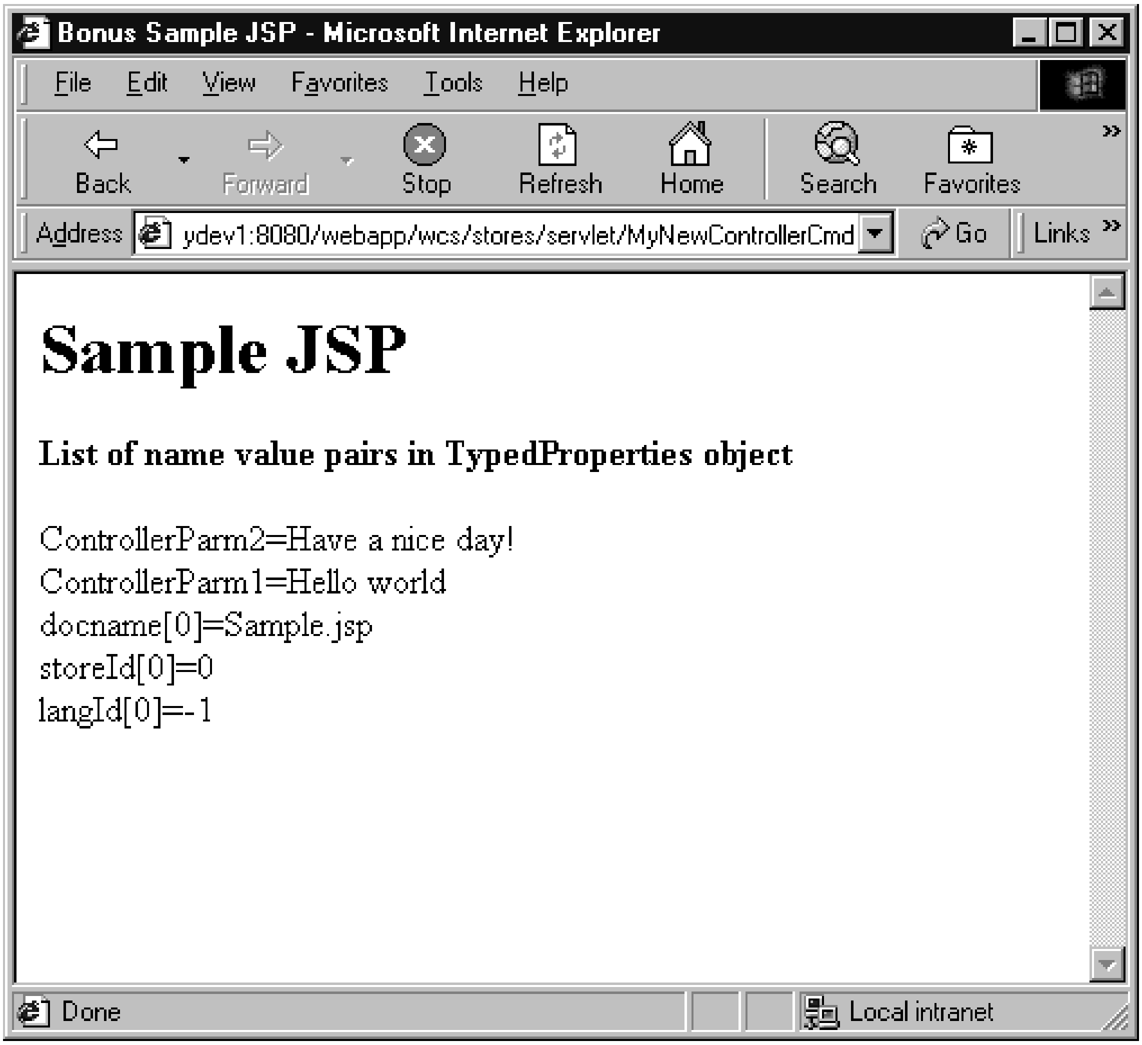
6. Test the modifications to verify that variables are displayed in the JSP template, by doing the following:

a. Ensure that the WebSphere Test Environment is running.

b. In a browser, enter the following URL:

```
http://localhost:8080/webapp/wcs/stores/servlet/MyNewControllerCmd
```

The Sample JSP file is displayed, showing properties of the controller command. The output appears as follows:


Bonus Sample JSP - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address ydev1:8080/webapp/wcs/stores/servlet/MyNewControllerCmd Go Links
Done
Local intranet

Sample JSP

List of name value pairs in TypedProperties object

```
ControllerParm2=Have a nice day!
ControllerParm1=Hello world
docname[0]=Sample.jsp
storeId[0]=0
langId[0]=-1
```

Figure 32.

7. Create a version of your code in its current state. Name the version `mySample 1.3 Completed`. If you require details on versioning code, refer to step 12 on page 208.

Modify the validateParameters Method

The validateParameters method that is currently in your new command is actually just a command stub. It consists of the following code:

```
public void validateParameters() throws ECException {
     }
```

In this section, you will add your own customized parameter checking to your command and then pass the parameters to the JSP template.

When modifying the validateParameters method, you add new fields to the class that are used for the parameters.

Creating new fields: When adding new fields to an existing interface or class, you can use the Create Fields SmartGuide in VisualAge for Java. This section describes the generic steps for creating a new field. Use this information in the following sections of the tutorial, when you must add new fields to interfaces and classes.

To create a new field, do the following:

1. Right-click the interface or class to which you are adding the new field and select **Add > Field**.
   The Create Field SmartGuide opens.
2. In the **Field Name** field, enter the name of the new field.
3. To specify the field type, do one of the following:
   - From the **Field Type** drop-down list, select the field type.
   - If the required field type is not specified in the list, click **Browse**. In the **Pattern** field, enter the name (or partial name) of the field type and click **OK**.

   Note: In the tutorials, whenever the field type of `String` is specified, that is from the `java.lang` package.
4. In the **Initial Value** field, enter the field's initial value. For initial values that are strings, be sure to enclose the value in double-quotes (" ").
5. For the **Access Modifiers** value, select the appropriate radio button (public, protected, none, or private), if required.
6. Select the **Access with getter and setter methods** check box if you would to have getter and setter methods generated for the field. If you select this, you must also specify properties for the getter and setter methods, as follows:
   - For the getter method, select one of the public, protected, private, or none radio buttons.
   - For the setter method, select one of the public, protected, private, or none radio buttons.

7. Click **Finish**.

To modify the validateParameters method, do the following:

1. You must create two new fields in the MyNewControllerCommandImpl class. The first is used for input strings, and the second is used for input integers. To create these fields, do the following:
   a. Expand the **com.ibm.commerce.sample.commands** package.
   b. Select the **MyNewControllerCmdImpl** class.
   c. Add a field to the class, using the following values. For detailed steps on how to create a new field, refer to "Creating new fields" on page 213.

Attribute Name	Value
Field Name	`inputString`
Field Type	`String`
Initial Value	Leave blank.
Access Modifiers	`private`
Other Modifiers	Leave all unchecked.
Access with getter and setter methods	`checked`
Getter	`public`
Setter	`public`

   d. Create another field for input integers, using the following values:

Attribute Name	Value
Field Name	`inputInteger`
Field Type	`Integer` **Note:** Click **Browse** and enter `Integer`. Do not select `int`.
Initial Value	Leave blank.
Access Modifiers	`private`
Other Modifiers	Leave all unchecked.
Access with getter and setter methods	`checked`
Getter	`public`
Setter	`public`

2. Select the **performExecute** method in the MyNewControllerCmdImpl class.
3. In the source code of the performExecute method, uncomment Section 3 to introduce the following code into the method:

```
// see how controller command pass in input variables to JSP

     rspProp.put("ControllerInput1", getInputString());
     rspProp.put("ControllerInput2", getInputInteger().toString());
```

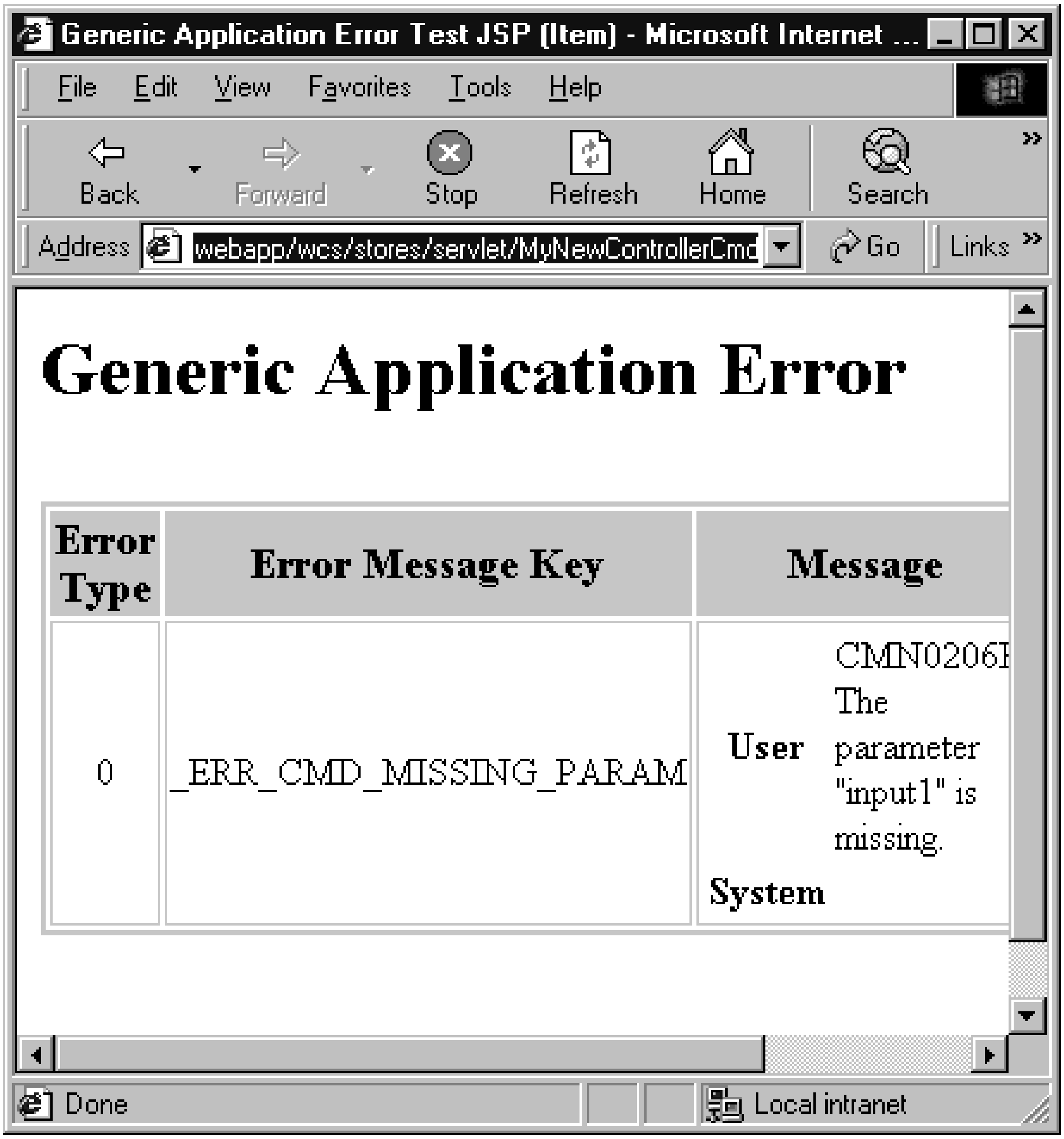
This code passes variables from the controller command to the data bean. Save your work.

4. Select the **validateParameters** method in the MyNewControllerCmdImpl class.
5. Uncomment Section 1 in the validateParameters source code, to introduce the following code into the method:

```
// uncomment to check parameters

      TypedProperty prop = getRequestProperties();

// retrieve required parameters
 //
 try {
          setInputString(prop.getString("input1"));
 } catch (ParameterNotFoundException e) {
          throw new ECApplicationException(
               ECMessage._ERR_CMD_MISSING_PARAM,
               "MyControllerCmdImpl","validateParameters",
               ECMessageHelper.generateMsgParms(e.getParamName()));
 }

 // retrieve optional Integer
 // set input2 = 0 if no input value
 //
 setInputInteger(prop.getInteger("input2", 0));
```

Save your work.
The preceding code snippet checks the two input parameters. The try block determines if the first parameter is there, if it is not, an exception is thrown. Since the second parameter is optional, this code will set the value for the parameter to `0` if the parameter is either missing or the wrong type.

6. Test the command by doing the following:
   a. Ensure that the persistent name server, EJB server and servlet engine are running.
   b. In your browser, by entering the following URLs:
      - Case 1: Missing parameter:
        Enter
        ```
        http://localhost:8080/webapp/wcs/stores/servlet/MyNewControllerCmd
        ```
        Since no parameters are passed to the command a generic application error is shown to indicate that a parameter is missing. The result is shown in the following screen shot:


Generic Application Error Test JSP (Item) - Microsoft Internet ...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search
Address webapp/wcs/stores/servlet/MyNewControllerCmd Go Links
Done
Local intranet

Generic Application Error

Error Type	Error Message Key	Message	
0	_ERR_CMD_MISSING_PARAM	User	CMN0206I The parameter "input1" is missing.
		System	

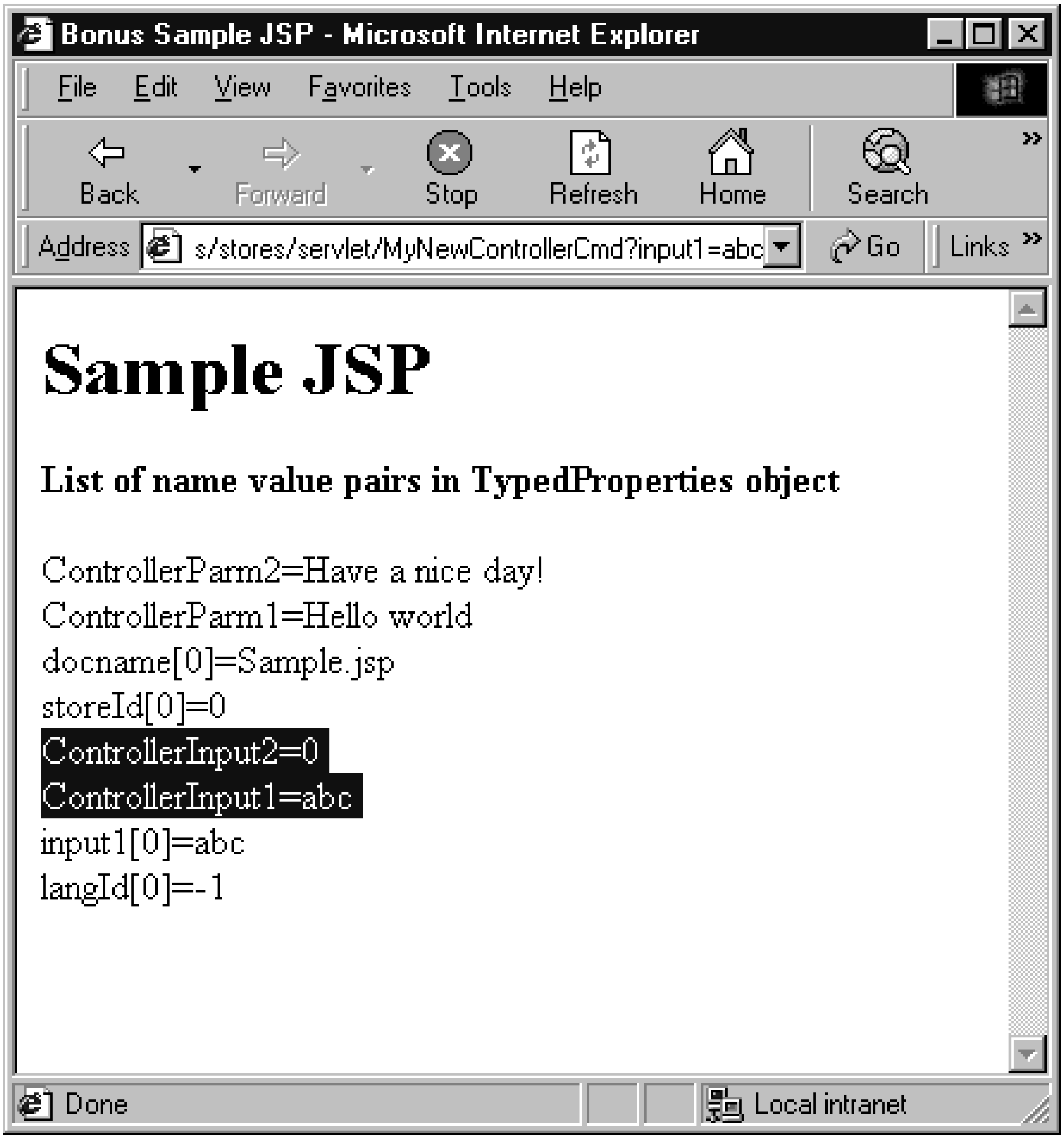
Figure 33.

Note: If a "Page not found" error is displayed, your servlet engine may have stopped. Check the WebSphere Test Environment Control Center for details. If the Sample JSP page is displayed instead of the Generic Application Error page, you may need to stop and restart the servlet engine, or reload the page in your browser.

- Case 2: First parameter valid, second parameter missing: Enter

  ```
  http://localhost:8080/webapp/wcs/stores/servlet/MyNewControllerCmd?
        input1=abc
  ```

  The result of this command is that the Sample JSP page is displayed despite the fact that the second parameter was omitted. The following screen shot displays this result. An explanation of why an error was not returned follows the screen shot.



Sample JSP

List of name value pairs in TypedProperties object

```
ControllerParm2=Have a nice day!
ControllerParm1=Hello world
docname[0]=Sample.jsp
storeId[0]=0
ControllerInput2=0
ControllerInput1=abc
input1[0]=abc
langId[0]=-1
```

Figure 34.

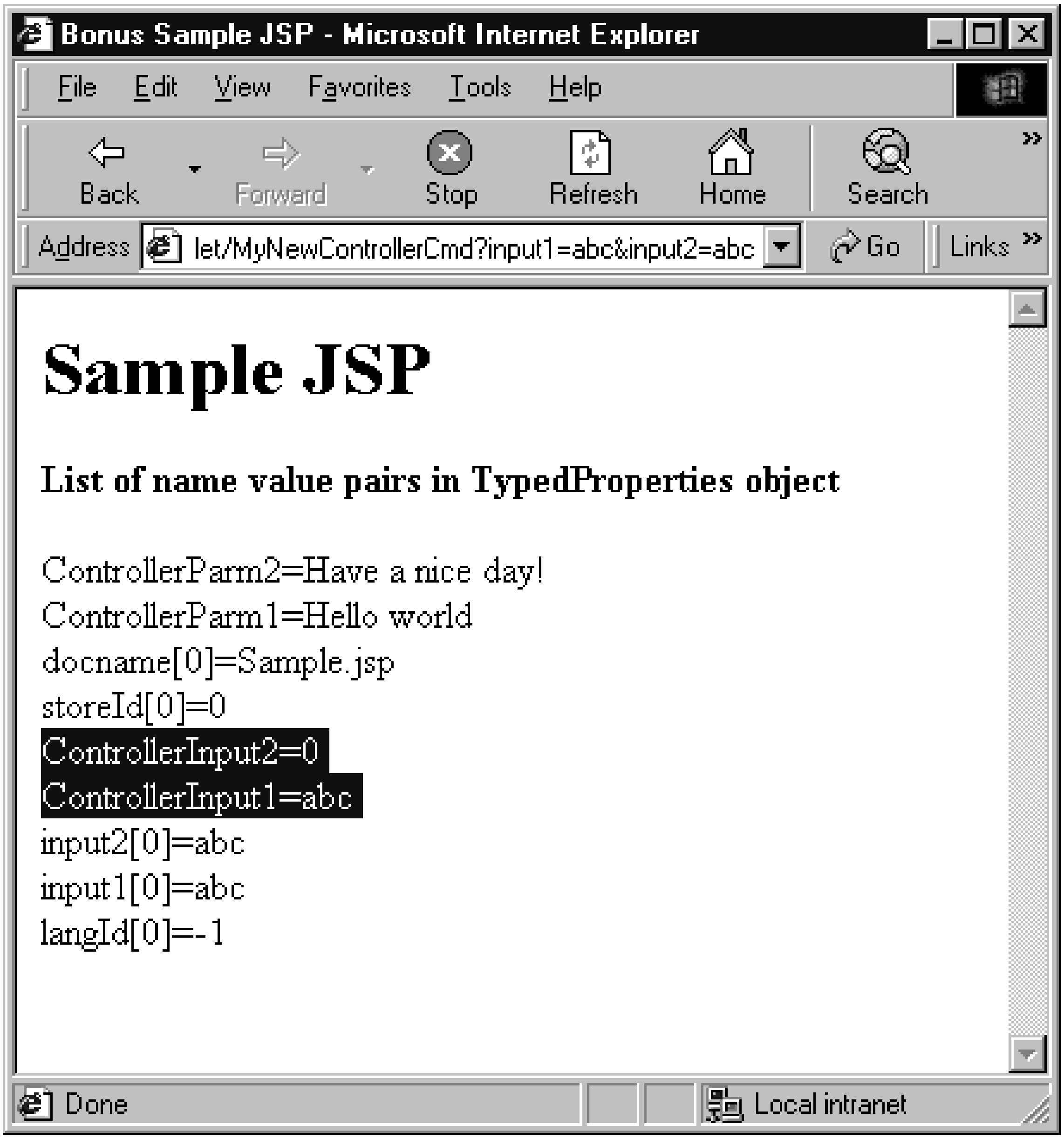
An error was not returned because of the manner in which the `getInteger` method was used. Specifically, the `setInputInteger(prop.getInteger("input2", 0))` line of code sets a default value of 0 for input2. This default value is used when the parameter is either missing or it is the wrong type. In order to force type checking on this parameter, change the code to `setInputInteger(prop.getInteger("input2"))` and enter the URL again (be sure to refresh your browser). A generic application error page should be shown.

Note: If you test the `setInputInteger(prop.getInteger("input2"))` modification in your code, switch it back to `setInputInteger(prop.getInteger("input2", 0))` before continuing with the next test case.

- Case 3: First parameter valid, second parameter invalid:
  Enter

  ```
  http://localhost:8080/webapp/wcs/stores/servlet/MyNewControllerCmd?
       input1=abc&input2=abc
  ```

  The result of this command is that the Sample JSP page is displayed and the value for the second parameter (an integer) is set to the default value of `0`. This is a result of the `getInteger` method used, as described in the preceding test case. The result is shown in the following screen shot:



Sample JSP

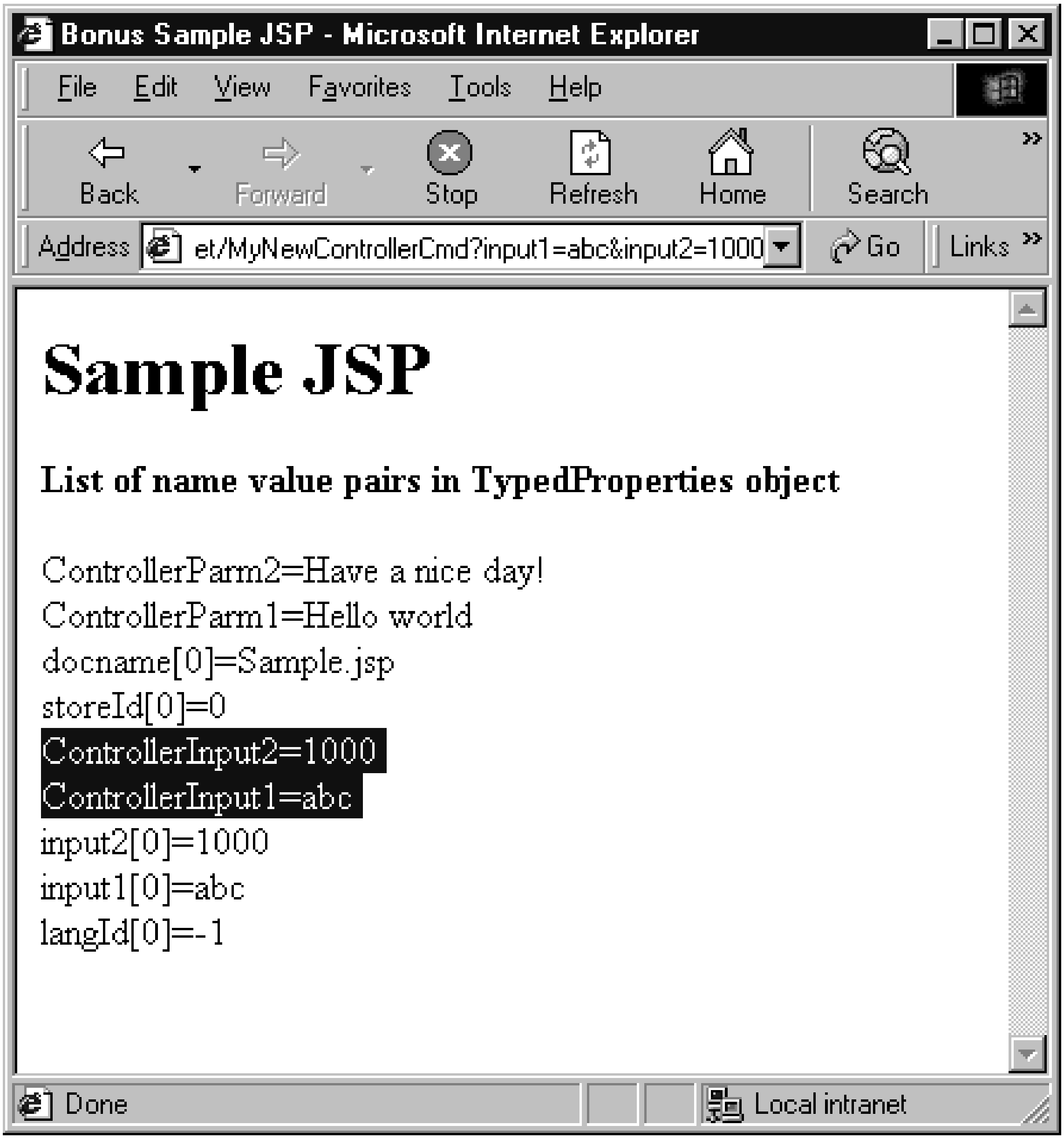
List of name value pairs in TypedProperties object

```
ControllerParm2=Have a nice day!
ControllerParm1=Hello world
docname[0]=Sample.jsp
storeId[0]=0
ControllerInput2=0
ControllerInput1=abc
input2[0]=abc
input1[0]=abc
langId[0]=-1
```

Figure 35.

- Case 4: Two valid parameters:
  Enter
  ```
  http://localhost:8080/webapp/wcs/stores/servlet/MyNewControllerCmd?
       input1=abc&input2=1000
  ```

The result of this command is that the Sample JSP page is displayed and both input values are displayed as entered. The result is shown in the following screen shot:



Sample JSP

List of name value pairs in TypedProperties object

```
ControllerParm2=Have a nice day!
ControllerParm1=Hello world
docname[0]=Sample.jsp
storeId[0]=0
ControllerInput2=1000
ControllerInput1=abc
input2[0]=1000
input1[0]=abc
langId[0]=-1
```

Figure 36.

7. Create a version of your code in its current state. Name the version `mySample 1.4 Completed`. If you require detailed information on how to version the code, refer to step 12 on page 208.

Create a task command

A controller command typically represents a business process or complex function. For example, all of the business logic related to processing orders is encapsulated in the OrderProcessCmd controller command. A business process can often be divided up into smaller, specific tasks. For example, within the OrderProcessCmd controller command, there are several task commands that get called to perform individual units of work. One of these task commands called within by the OrderProcessCmd controller command is CalculateOrderTaxTotalCmd.

MyNewControllerCmdImpl does not currently call any task commands. This exercise has two sections. In the first section, you create the new task command. In the second section, you modify the performExecute method of your controller command to call the task command.

The following code snippet shows the current performExecute method from the MyNewControllerCmdImpl class, with all comments removed:

```
public void performExecute() throws ECException {

     super.performExecute();

     TypedProperty rspProp = new TypedProperty();
     rspProp.put("ControllerParm1", "Hello world");
     rspProp.put("ControllerParm2", "Have a nice day!");

     rspProp.put("ControllerInput1", getInputString());
     rspProp.put("ControllerInput2", getInputInteger().toString());

     rspProp.put(ECConstants.EC_VIEWTASKNAME, "SampleViewTask");
     setResponseProperties(rspProp);

}
```

Write the Task Command Code: This section shows you how to write a new task command. Creating a completely new task command involves creating an interface and an implementation class. When creating a task command, the interface should extend `com.ibm.commerce.commands.TaskCommand`. The implementation class should extend `com.ibm.commerce.command.TaskCommandImpl`.

Upon completion of this exercise, you will have a new command, called *MyNewTaskCmd*. This command is used by all stores and each store uses the same implementation of the command.

In this part of the tutorial, you add fields and methods to the interface of the new task command. You have previously created new fields for the MyNewControllerCmdImpl class. Try creating the fields for this interface on your own, but if you need more details, refer to "Creating new fields" on page 213.

Creating methods: This section describes the generic steps for adding methods to existing classes and interfaces. Read the instructions here and refer back to them when the tutorial requires you to create new methods.

To create a new method, do the following:

1. Right-click the class or interface to which you are adding the method and select **Add > Method**.
   The Create Method SmartGuide opens.
2. Ensure that **Create a new method** is selected and click **Next**.
3. In the **Method Name** field, enter the name for the new method.
4. Specify the method's return type by doing one of the following:
   - From the **Return Type** drop-down list, select the appropriate return type. For example, select `String`.
   - If the return type is not in the list, click **Browse**. Then in the **Pattern** field, enter the return type and click **OK**.

   Note: In the tutorials, whenever `String` is specified as the return type, this is from the `java.lang` package.
5. If the method takes parameters, click **Add**. In the Parameters window, specify the parameter name and any other required information and click **Add**. After all parameters have been added, click **Close**.
6. Click **Next**.
   The Attributes window opens.
7. If the method throws exceptions, click **Add** in the Attributes window. In the **Pattern** field, enter the name of the exception, then click **Add**. After all exceptions have been added, click **Close**.
8. Click **Finish**.
   The code for the method is generated.

To create MyNewTaskCmd command, do the following:

1. Expand the **com.ibm.commerce.sample.commands** package.
2. Right-click the **MyNewTaskCmd** interface and select **Add > Field**.
3. Using the Create Field Smart Guide, create a field that specifies the default implementation class to be used by the interface. Use the values in the following table. If you need further details on creating a field, refer back to "Creating new fields" on page 213.

Attribute Name	Value
Field Name	`defaultCommandClassName`
Field Type	`String`
Initial Value	`"com.ibm.commerce.sample.commands.MyNewTaskCmdImpl"`

When the code for the field is generated, it appears as follows:

```
java.lang.String defaultCommandClassName =
     "com.ibm.commerce.sample.commands.MyNewTaskCmdImpl";
```

Since the same implementation class is used for the entire site and no default properties are passed to the command, you can specify the default implementation right in the code. If you have a command that either has multiple implementations, or has default properties (which are stored in the CMDREG table), you must register the command in the CMDREG table to create the mapping between the interface and implementation class.

4. Add new methods to the MyNewTaskCmd interface, by doing the following:
   a. Right-click the **MyNewTaskCmd** interface and select **Add > Method**. Using the Create Method SmartGuide, create new methods using the values specified in the following steps. If you require detailed information for creating the methods, refer to "Creating methods" on page 223.
   b. Create a new method that retrieves a customer's balance of bonus points, using the following values:

Attribute Name	Value
Method Name	`getOldBonusPoint`
Return Type	`String`
Parameters	none
Exceptions	none

Note: An error may be displayed indicating that the inherited abstract method just created is not implemented in the MyNewTaskCmdImpl implementation class. This is corrected in a subsequent step.

   c. Create a new method that retrieves the user ID output from the task command. When creating this method, use the following values:

Attribute Name	Value
Method Name	`getTask_output_userId`

Attribute Name	Value
Return Type	`String`
Parameters	none
Exceptions	none

d. Create a new method that retrieves an output value from the task command. When creating this method, use the following values:

Attribute Name	Value
Method Name	`getTask_output1`
Return Type	`String`
Parameters	none
Exceptions	none

e. Create a new method that sets the first input value to the task command. When creating this method, use the following values:

Attribute Name	Value
Method Name	`setTask_input1`
Return Type	`void`
Parameter Name	`newTask_input1`
Reference Type	`String`
Exceptions	none

f. Create a new method that sets the second input value to the task command. When creating this method, use the following values:

Attribute Name	Value
Method Name	`setTask_input2`
Return Type	`void`
Parameter Name	`newTask_input2`
Primitive Type	`int`
Exceptions	none

5. Add new fields to the MyNewTaskCmdImpl class, by doing the following:
   a. Right-click the **MyNewTaskCmdImpl** class and select **Add > Field**. Using the Create Field SmartGuide, create new fields using the values specified in the following steps. If you require additional information about creating new fields, refer to “Creating new fields” on page 213.

b. Create a new field in the implementation class, using the following values:

Attribute Name	**Value**
Field Name	`task_input1`
Field Type	`String`
Initial Value	Leave blank.
Access Modifiers	`private`
Other Modifiers	Leave all unchecked.
Access with getter and setter methods	`checked`
Getter	`public`
Setter	`public`

c. Create a new field in the implementation class, using the following values:

Attribute Name	**Value**
Field Name	`task_input2`
Field Type	`int`
Initial Value	Leave blank.
Access Modifiers	`private`
Other Modifiers	Leave all unchecked.
Access with getter and setter methods	`checked`
Getter	`public`
Setter	`public`

d. Create a new field in the implementation class, using the following values:

Attribute Name	**Value**
Field Name	`task_output_userId`
Field Type	`String`
Initial Value	Leave blank.
Access Modifiers	`private`
Other Modifiers	Leave all unchecked.
Access with getter and setter methods	`checked`
Getter	`public`
Setter	`public`

e. Create a new field in the implementation class, using the following values:

Attribute Name	**Value**
Field Name	`oldBonusPoint`
Field Type	`String`
Initial Value	Leave blank.
Access Modifiers	`private`
Other Modifiers	Leave all unchecked.
Access with getter and setter methods	`checked`
Getter	`public`
Setter	`public`

f. Create a new field in the implementation class, using the following values:

Attribute Name	**Value**
Field Name	`task_output1`
Field Type	`String`
Initial Value	Leave blank.
Access Modifiers	`private`
Other Modifiers	Leave all unchecked.
Access with getter and setter methods	`checked`
Getter	`public`
Setter	`public`

6. Select the **performExecute** method of the **MyNewTaskCmdImpl** class to view its source code.
7. In the source code, uncomment Section 1 to introduce the following code into the method:

```
// modify the task_input1 and see it in the NVP list

     setTask_output1( "Hello ! " + getTask_input1() );
```

Save your work.
The preceding section of code makes the new attributes available as output from the command.

Call the Task Command: Once you have created your task command, you need to call the command from within your controller command. The following steps describe how to modify your controller command in this manner.

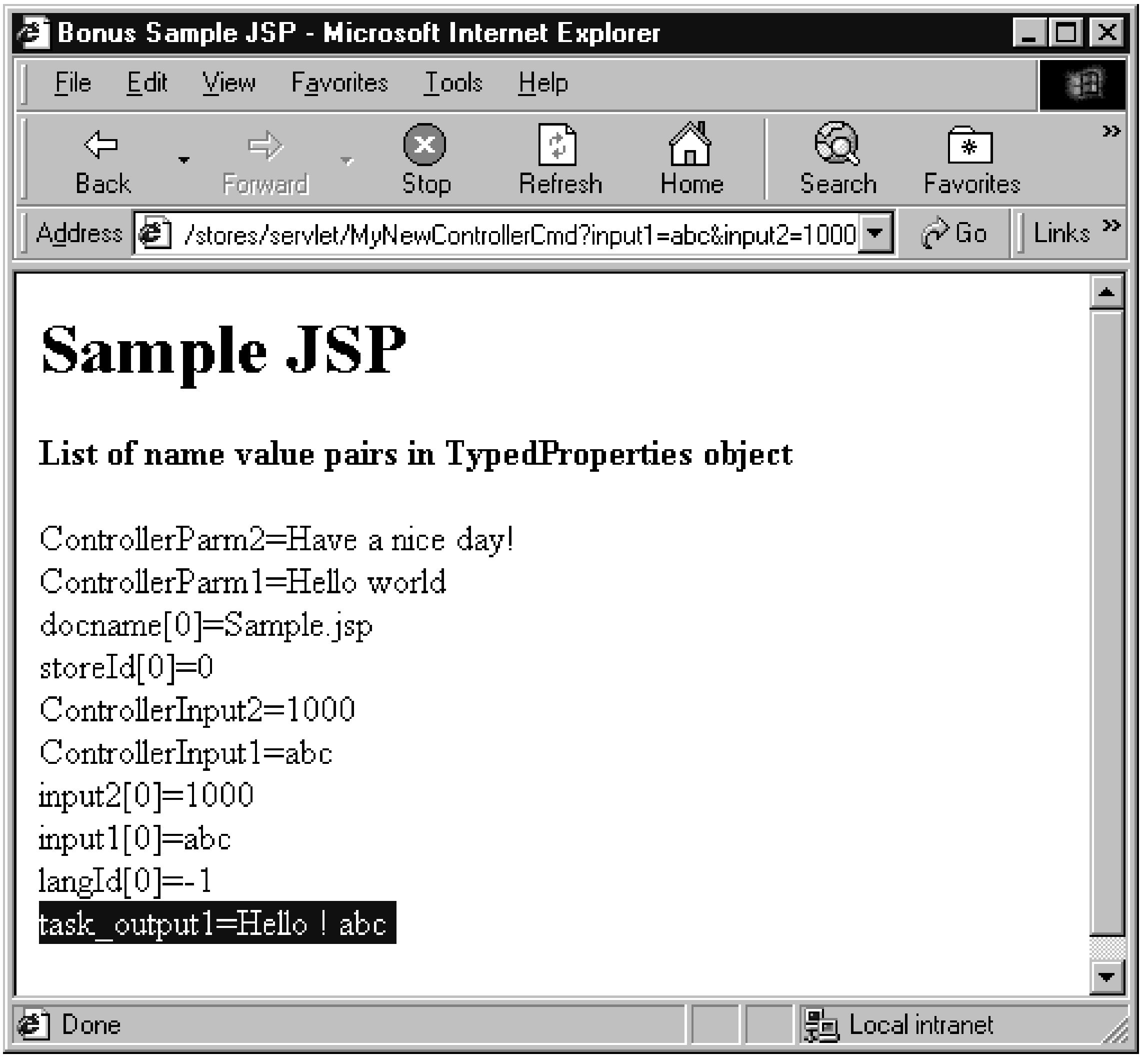
1. In the Workbench, select the **performExecute** method of your **MyNewControllerCmdImpl** class.
2. In the Source pane, uncomment Section 4 to call the task command. This introduces the following code into the method:

```
// see how controller command call a task command

     MyNewTaskCmd cmd = null;
     try {
          cmd = (MyNewTaskCmd) CommandFactory.createCommand(
               "com.ibm.commerce.sample.commands.MyNewTaskCmd",
               getStoreId());
          // Set input parameters to task command
          cmd.setTask_input1(getInputString());
          cmd.setTask_input2(getInputInteger().intValue());
          // This is required for all commands
          cmd.setCommandContext(getCommandContext());
          // Invoke the command's performExecute method
          cmd.execute();
          // retrieve output parameter from task command

          rspProp.put("task_output1", cmd.getTask_output1());

          if (cmd.getTask_output_userId() != null) {
               rspProp.put("task_output_userId",
               cmd.getTask_output_userId());
          }

          if (cmd.getOldBonusPoint() != null) {
               rspProp.put("task_output_oldBonusPoint",
               cmd.getOldBonusPoint());
          }

 } catch (ECException ex) {
          // throw the exception as is
          throw (ECException) ex;
 }
```

   Save your work.
   The preceding code snippet creates a new task command using the command factory. It then sets the command context, invokes the perform execute of the task command and retrieves output parameters from the task command.
3. Test the command by entering the URL for your controller command, as follows:

```
http://localhost:8080/webapp/wcs/stores/servlet/MyNewControllerCmd?
     input1=abc&input2=1000
```

The Sample JSP displays the list of name value pairs in the request object, including the task output values. It should appear as follows:


Bonus Sample JSP - Microsoft Internet Explorer
File
Edit
View
Favorites
Tools
Help
Back
Forward
Stop
Refresh
Home
Search
Favorites
Address
/stores/servlet/MyNewControllerCmd?input1=abc&input2=1000
Go
Links
Done
Local intranet

Sample JSP

List of name value pairs in TypedProperties object

```
ControllerParm2=Have a nice day!
ControllerParm1=Hello world
docname[0]=Sample.jsp
storeId[0]=0
ControllerInput2=1000
ControllerInput1=abc
input2[0]=1000
input1[0]=abc
langId[0]=-1
task_output1=Hello ! abc
```

Figure 37.

4. Create a version of your code in its current state. Name the version `mySample 1.5 Completed`. If you require more details on how to version your code, refer to step 12 on page 208.

Validate a user ID

The next step is to modify the task command to use the UserRegistryAccessBean for the purpose of validating whether or not the value entered by the user is that of a registered user. In addition to modifying the task command, you must also modify the DataBeanSampleBean.

Modify MyNewTaskCmdImpl for user ID validation: You must modify the performExecute method in the MyNewTaskCmdImpl class so that it will validate the user's ID, by using the UserRegistryAccessBean. To add this new functionality to performExecute method, do the following:

1. In the Workbench, select the **performExecute** method of your **MyNewTaskCmdImpl** class.
2. In the source pane, uncomment Section 2 of the performExecute method. This introduces the following code into the method:

```
// use UserRegistryAccessBean to check member reference number

     String refNum;

     UserRegistryAccessBean rrb = new UserRegistryAccessBean();

     try {
          rrb = rrb.findByUserLogonId(getTask_input1());
          refNum = rrb.getUserId();

     } catch (javax.ejb.FinderException e) {

     return;

     } catch (javax.naming.NamingException e) {
          throw new ECSystemException(ECMessage._ERR_NAMING_EXCEPTION,
          this.getClass().getName(), "performExecute");
     } catch (java.rmi.RemoteException e) {
          throw new ECSystemException(ECMessage._ERR_REMOTE_EXCEPTION,
          this.getClass().getName(), "performExecute");
     } catch (javax.ejb.CreateException e) {
          throw new ECSystemException(ECMessage._ERR_CREATE_EXCEPTION,
          this.getClass().getName(), "performExecute");
     }

      setTask_output_userId(refNum);
```

   Save your work.

Modify the DataBeanSampleBean: You must modify the DataBeanSampleBean to use your predefined input parameters. To modify this bean, do the following:

1. In the Workbench, expand the **com.ibm.commerce.sample.databeans** package.
2. Add new fields to the data bean, by doing the following:
   a. Right-click the **DataBeanSampleBean** class and select **Add > Field**. Using the Create Field SmartGuide, create new fields using the values

specified in the following steps. If you require additional information about creating fields, refer to "Creating new fields" on page 213.

b. Add a new field to the data bean, using the following values:

Attribute Name	Value
Field Name	`input1`
Field Type	`String`
Initial Value	Leave blank.
Access Modifiers	`private`
Other Modifiers	Leave all unchecked.
Access with getter and setter methods	`checked`
Getter	`public`
Setter	`public`

c. Add a new field to the data bean, using the following values:

Attribute Name	Value
Field Name	`input2`
Field Type	`int`
Initial Value	Leave blank.
Access Modifiers	`private`
Other Modifiers	Leave all unchecked.
Access with getter and setter methods	`checked`
Getter	`public`
Setter	`public`

d. Add a new field to the data bean, using the following values:

Attribute Name	Value
Field Name	`task_output_userId`
Field Type	`String`
Initial Value	Leave blank.
Access Modifiers	`private`
Other Modifiers	Leave all unchecked.
Access with getter and setter methods	`checked`
Getter	`public`
Setter	`public`

3. Select the **populate** method of the DataBeanSampleBean class to view its source code.
4. In the source code, uncomment Section 1A and Section 1B. This introduces the following code into the method:

```
//// Section 1A //////////
//  set additional data fields

try {
   setInput1( getRequestProperties().getString("input1"));
   setInput2( getRequestProperties().getIntValue("input2", 0)  );
   setTask_output_userId(getRequestProperties().getString
     ("task_output_userId"));

//// End of Section 1A /////

   //// Section 2 ////////////
   /*
   // instantiate databean to BonusAccessBean
   setTask_output_oldBonusPoint(getRequestProperties().getString(
      "task_output_oldBonusPoint"));
   */
   //// End of Section 2 ////

//// Section 1A ///////////
// instantiate databean to BonusAccessBean and
// set additional data fields

   }
catch (ParameterNotFoundException e){}

//// End of Section 1B ////
```

   Save your work.
   The preceding code snippet gets data field values from the controller command, using the getRequestProperties method.

Modify the Sample.jsp for user ID validation: The current JSP template must be modified in order to perform user ID validation. To modify the Sample.jsp file, do the following:

1. Open the `Sample.jsp` and `Sample_All.jsp` files in a text editor.
2. Copy Section 3 of the code from Sample_All.jsp into Sample.jsp between the `<!-- SECTION 3 -->` and `<!-- END OF SECTION 3 -->` markers. The following code is introduced into the JSP template

```
<!-- SECTION 3 -->

<B>Your first input is &lt; <%=testBean.getInput1()%> &gt;</B>

<%
String userId = testBean.getTask_output_userId();

if (userId == null) {
```

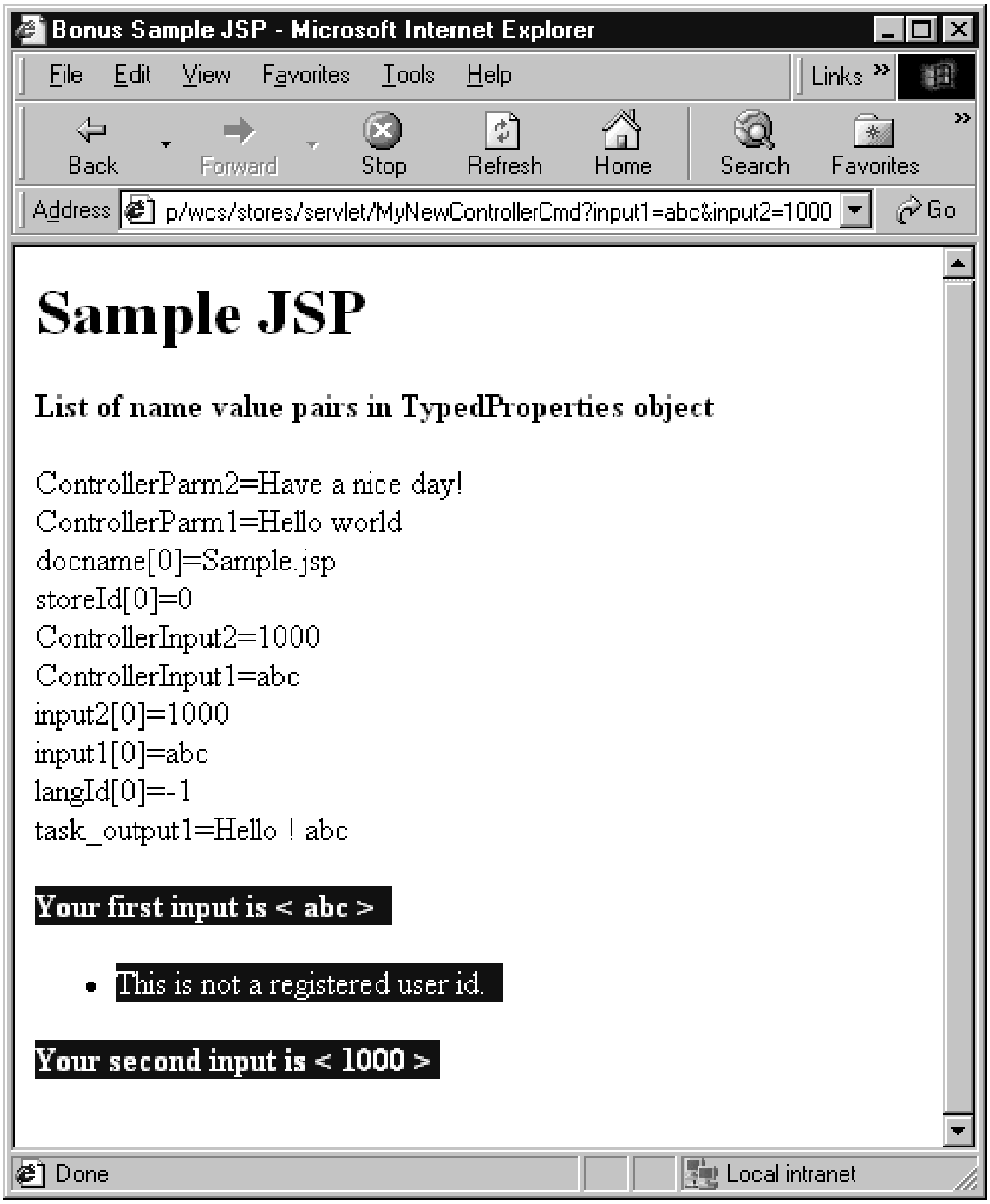
```
%>

<UL>
     <LI> This is not a registered user id.
</UL>

<%

} else {

%>

<B>
<UL>
     <LI> 'lt; <%=testBean.getInput1()%> &gt; is a registed user id.
     <LI> The member reference number of this user is <%=userId%>.
</UL>
</B>

<%

}

%>

<B>Your second input is < <%=testBean.getInput2()%> ></B> <P>

<!-- END OF SECTION 3 -->
```

Save the Sample.jsp file.

3. Open a browser and enter the following URL:

```
http://localhost:8080/webapp/wcs/stores/servlet/MyNewControllerCmd?
     input1=abc&input2=1000
```

The browser should show the Sample JSP file indicating that the value for input1 is not a valid user ID, as shown in the following screen shot:


Bonus Sample JSP - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Links
Back Forward Stop Refresh Home Search Favorites
Address p/wcs/stores/servlet/MyNewControllerCmd?input1=abc&input2=1000
Go
Done
Local intranet

Sample JSP

List of name value pairs in TypedProperties object

```
ControllerParm2=Have a nice day!
ControllerParm1=Hello world
docname[0]=Sample.jsp
storeId[0]=0
ControllerInput2=1000
ControllerInput1=abc
input2[0]=1000
input1[0]=abc
langId[0]=-1
task_output1=Hello ! abc
```

Your first input is < abc >

- This is not a registered user id.

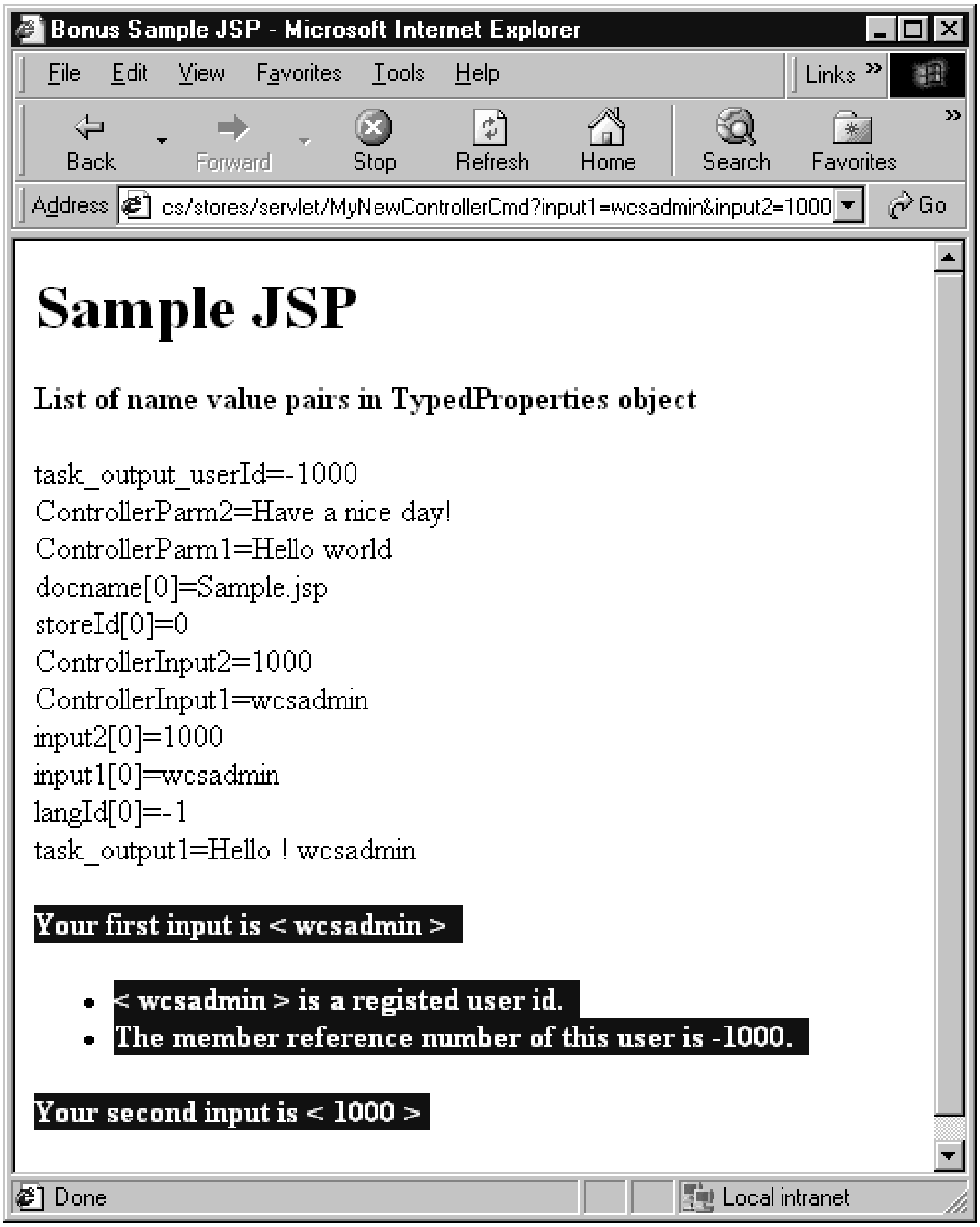
Your second input is < 1000 >

Figure 38.

4. To see the result when the value for input1 is a valid user ID, enter the following URL:

```
http://localhost:8080/webapp/wcs/stores/servlet/MyNewControllerCmd?
     input1=wcsadmin&input2=1000
```

This is shown in the following screen shot:



Bonus Sample JSP - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Links
Back Forward Stop Refresh Home Search Favorites
Address cs/stores/servlet/MyNewControllerCmd?input1=wcsadmin&input2=1000
Go
Done
Local intranet

Sample JSP

List of name value pairs in TypedProperties object

```
task_output_userId=-1000
ControllerParm2=Have a nice day!
ControllerParm1=Hello world
docname[0]=Sample.jsp
storeId[0]=0
ControllerInput2=1000
ControllerInput1=wcsadmin
input2[0]=1000
input1[0]=wcsadmin
langId[0]=-1
task_output1=Hello ! wcsadmin
```

Your first input is < wcsadmin >

- < wcsadmin > is a registed user id.
- The member reference number of this user is -1000.

Your second input is < 1000 >

Figure 39.

5. Create a version of your code in its current state. Name the version `mySample 1.6 Completed`. If you require detailed information on how to version your code, refer to step 12 on page 208.

Creating a new entity bean

This section describes how to use VisualAge for Java to create a new entity bean. In this example scenario, you have a business requirement to include a tally of bonus points for each user in the commerce application. The WebSphere Commerce database schema does not contain this information, so you need to create a new database table to hold this information. In accordance with the WebSphere Commerce programming model, once the database table is created, you must create an entity bean (which is an enterprise bean) to access the data.

In this example you will use a VisualAge for Java SmartGuide to create this entity bean.

Creating the new database table

In preparation for creating the entity bean, you must first create the new database table. The table to be created is called Bonus.

DB2 If you are using a DB2 database, do the following to create the table:

1. Open the DB2 Command Center (**Start > Programs > IBM DB2 > Command Center**) and click the **Scripting** tab.
2. In the Script window, enter the following:

   ```
   connect to your_database_name;
   CREATE TABLE Bonus (MEMBERID BIGINT NOT NULL,
       BONUSPOINT INTEGER NOT NULL, constraint p_memberid primary key (MEMBERID),
       constraint f_memberid foreign key (MEMBERID)
       references users (users_id) on delete cascade)
   ```

   where *your_database_name* is the name of your database. Click the Execute icon.
   The Bonus table is now created.

   Note: You must issue the following command before creating the Bonus table, if anyone has previously run this example using this database:

   ```
   drop table Bonus
   ```

Oracle If you are using an Oracle database, do the following to create the table:

1. Open the Oracle SQL Plus command window (**Start > Programs > Oracle > Application Development > SQL Plus**).
2. In the **User Name** field, enter your Oracle user name.
3. In the **Password** field, enter your Oracle password.
4. In the **Host String** field, enter your connect string.
5. In the SQL Plus window, enter the following SQL statement:

```
CREATE TABLE Bonus (MEMBERID NUMBER NOT NULL,
    BONUSPOINT INTEGER NOT NULL, constraint p_memberid primary key (MEMBERID),
    constraint f_memberid foreign key (MEMBERID)
    references users (users_id) on delete cascade);
```

and press Enter to run the SQL statement. The BONUS table is now created.

Note: You must issue the following command before creating the Bonus table, if anyone has previously run this example using this database:

```
drop table Bonus;
```

6. Enter the following to commit your database changes:

```
commit;
```

and press Enter to run the SQL statement.

Creating the BonusBean entity bean

Once the database has been created, you are ready to begin creating the new entity bean. The next steps use VisualAge for Java. To create the new entity bean, do the following:

1. Create an EJB group. An EJB group is a logical group that allows you to organize your enterprise beans. You can perform global operations on an EJB group that iterates on all of the enterprise beans residing in that group. For example, if you select an EJB group to export to an EJB JAR file, all of the enterprise beans in the group are exported.
   In this tutorial, you create an EJB group to organize all enterprise beans related to your Bonus table customization.
   To create your EJB group, do the following:
   a. In the Workbench, click the **EJB** tab.
   b. From the **EJB** menu, select **Add > EJB Group**.
      The Add EJB Group SmartGuide opens.
   c. In the **Project** field, enter `_WCSamplesEntityBeansProject`.

      Note: Entity bean code must be stored within its own project, for deployment purposes.
   d. In the **Create a new EJB group named** field, enter `WCSSamplesEntityBeans` and click **Finish**.
2. Create your new entity bean.
   To create your new entity bean, do the following:
   a. In the Enterprise Beans pane, Right-click the **WCSSamplesEntityBeans** EJB group and select **Add > Enterprise Bean**.
      The Create Enterprise Bean SmartGuide opens.

b. Enter the following information

Attribute	Value
Bean Name	`Bonus` **Note:** The naming convention is to call the entity bean by the same name as the table to which it accesses.
Bean Type	`Entity bean with container-managed persistence (CMP) fields`
Create a new bean class	`enable`
Project	`_WCSamplesEntityBeansProject`
Package	`com.ibm.commerce.sample.objects`
Class Name	`BonusBean`
Superclass	`com.ibm.commerce.base.objects.ECEntityBean`

and click **Next**.

c. Click the **Add** button beside the **Add CMP fields to the bean** text box to add a field for the MEMBERID column in the BONUS table.
The Create CMP Fields SmartGuide opens.

d. Enter the following information:

Attribute	Value
Field Name	memberId
Field Type	Long **Note:** You you must use the *Long* data type, not *long*.
Key Field	enable

and click **Finish**.

e. Click **Add** again to add a field for the BONUSPOINT column in the BONUS table.

f. Create another field with the following information:

Attribute	Value
Field Name	bonusPoint
Field Type	Integer **Note:** You you must use the *Integer* data type, not *int*.
Access with getter and setter methods	enable
Promote getter and setter methods to remote interface	enable

then click **Finish** in the Create CMP Field window.

g. Protect this enterprise bean using access control, by doing the following:
   1) Click **Add** beside **Which interfaces should the remote interface extend?**.
   2) Enter `com.ibm.commerce.security.Protectable` in the **Pattern** field and click **Add**. Click **Close** to close the window.

h. Click **Finish** again.

The Bonus entity is created as an enterprise bean.

3. Set the isolation level of the entity bean, by doing the following:
   a. Right-click the **Bonus** bean and select **Properties**.
   b. From the **Isolation Level** drop-down list, select **TRANSACTION_READ_COMMITTED** and click **OK**.
4. When you create a new enterprise bean VisualAge for Java generates an EntityContext field as well as corresponding getEntityContext() and setEntityContext(EntityContext) methods in the bean. Following the WebSphere Commerce programming model, your new bean extends the com.ibm.commerce.base.objects.ECEntityBean class and ECEntityBean provides its own implementation of this field and these methods. Since you should not override EntityContext, getEntityContext and setEntityContext, you must now delete the generated field and methods from your bean.
   To delete the generated EntityContext field and its getter and setter methods, do the following:
   a. In the Types pane, select the **BonusBean** class.
      The Members pane displays the fields and methods for this class.

      Note: If the Types pane is not visible, click the C/I icon (class/interface) in the Properties pane. The Types pane then opens.
   b. In the Members pane, do the following:
      1) Right-click the **entityContext** field and select **Delete**.
      2) Right-click the **getEntityContext()** method and select **Delete**.
      3) Right-click the **setEntityContext(EntityContext)** method and select **Delete**.
   c. Save your work (Ctrl+S).
5. Add a new getMemberId method to the enterprise bean, by doing the following:
   a. Right-click the **BonusBean** class and select **Add > Method**.
      The Create Method SmartGuide opens.

b. Create the new method using the values specified in the following table. If you require more detailed information about creating a new method, refer to “Creating methods” on page 223.

Table 11.

Attribute Name	Value
Method Name	`getMemberId`
Return Type	`Long`
Parameters	none
Exceptions	none

c. When the new method is generated, view the source code.

d. By default, the method contains the following code:

```
return null;
```

Change this code to

```
return memberId;
```

e. Add the new method to the remote interface by right-clicking the **getMemberId** method and selecting **Add to > EJB Remote Interface**.

6. Add new FinderHelper fields in BonusBeanFinderHelper.
This interface contains a search clause that corresponds to a FinderHelper method that is created in the next step. To add the FinderHelper fields, do the following:

   a. In the Types pane, click the **BonusBeanFinderHelper** interface.

   b. Modify the code in the Source pane so it appears as follows:

```
public interface BonusBeanFinderHelper {
    public static final String
                findByMemberIdWhereClause = " (MEMBERID = ?) ";
        }
```

Note: The syntax of the “WhereClause” is very important; it must match the method name used for the FinderHelper method. In this case the “findByMemberId” in `findByMemberIdWhereClause` matches exactly with the name of the method you create in the next step (`findByMemberId`).

7. Add new FinderHelper methods in the BonusHome interface.
To add new FinderHelper methods, do the following:

   a. In the Types pane, right-click the **BonusHome** interface and select **Add > Method**.
   The Create Method SmartGuide opens.

   b. Select **Create a new method** and click **Next**.

   c. In the **Method Name** field, enter `findByMemberId`.

d. In the **ReturnType** field, enter `Bonus`.

e. Click **Add** next to **What parameters should this method have?** The Parameters window opens.

f. In the **Name** field, enter `argMemberId`.

g. Select **Reference Types** and enter `Long`. Click **Add** then **Close**.

h. Click **Next**.

i. Click **Add** next to the **What exceptions may this method throw?** field, enter `RemoteException` in the **Pattern** field and click **Add**. This adds the `java.rmi.RemoteException` exception in the Attributes window. (The Attributes window may be positioned behind the Exceptions window.)

j. In the **Pattern** field of the Exceptions window, enter `FinderException`, click **Add**, and then click **Close**. The `javax.ejb.FinderException` exception is listed in the Attributes window.

k. Click **Finish**.

8. Add a new `ejbCreate` method to the EJB. This method is promoted to the home interface, so that it is available in a generated access bean. To create this method, do the following:

   a. Select the **BonusBean** class in the Types pane.

   b. Click **ejbCreate(Long)** in the Members pane.

   c. Modify the code so that it matches the following:

```
public void ejbCreate(java.lang.Long argMemberId,
  Integer argBonusPoint)
  throws javax.ejb.CreateException, java.rmi.RemoteException {
            _initLinks();
   // All CMP fields should be initialized here.
    memberId=argMemberId;
          bonusPoint=argBonusPoint;
   }
```

   d. Save the code and VisualAge for Java creates a new method, called **ejbCreate(Long, Integer)** in the Members pane.

   e. Right-click the original **ejbCreate(Long)** method and select **Delete**.

9. Add the new method to the home interface. This will make the method available in the access bean class. To do this, do the following:

   a. Right-click the **ejbCreate(Long, Integer)** method in the BonusBean class and select **Add To > EJB Home Interface**.

10. Update the getOwner method, by doing the following:

    a. Select the **BonusBean** class in the Types pane.

    b. Click the **getOwner()** method in the Members pane.

    Note: If you have selected to view the inherited methods, you will see two getOwner() methods. One is inherited from the

ECEntityBean class. This is not the one that you should select in this step. Ensure that you select the getOwner method that is specific to the BonusBean class.

c. The source code for the getOwner method appears as follows:

```
public Long getOwner()
   throws Exception, java.rmi.RemoteException
   {
      return null;
   }
```

You must change the value that the method returns. The portion of code that you should change is shown in bold in the following:

```
public Long getOwner()
   throws Exception, java.rmi.RemoteException
   {
      return getMemberId();
   }
```

Save your work.

d. Click the **fulfills(Long, String)** method in the Members pane.

Note: If you have selected to view the inherited methods, you will see two fulfills(Long, String) methods. One is inherited from the ECEntityBean class. This is not the one that you should select in this step. Ensure that you select the fulfills(Long, String) method that is specific to the BonusBean class.

e. The source code for the fulfills(Long, String) method appears as follows:

```
public boolean fulfills(Long member, String relationship)
  throws Exception, java.rmi.RemoteException
  {
      return false;
  }
```

You must specify the relationship that the user must fulfill. To do this, you must change the portion of code shown in bold in the following:

```
public boolean fulfills(Long member, String relationship)
  throws Exception, java.rmi.RemoteException
  {

      if (relationship.equalsIgnoreCase("creator"))
      {
         return member.equals(getMemberId());
      }
      return false;
  }
```

Save your work.

11. Map the BONUS database table to the BonusBean. The first step in mapping the database schema to the BonusBean entity involves using the tools in VisualAge for Java to create the database schema. Do the following to create the schema:
    a. In the Workspace window, from the **EJB** menu, select **Open To > Database Schemas**.
    b. From the **Schemas** menu, select **Import/Export Schema > Import Schema from Database**.
    The Information required window opens.
    c. In the **Schema Name** field, enter `WCSSamples`, and click **OK**.
    The Database Connection Info window opens.
    d. Fill in the following information:

Attribute	DB2 **DB2 value**	Oracle **Oracle value**
Connection Type	`COM.ibm.db2.jdbc.app.DB2Driver`	`Oracle.jdbc.driver.OracleDriver`
Data Source	`jdbc:db2:`*`wcs_db_name`*	`jdbc:oracle:thin:@`*`hostname`*`:`*`port`*`:`*`SID`*
User Name	*`wcs_db_user_name`*	*`wcs_db_user_name`*
Password	*`wcs_db_password`*	*`wcs_db_password`*

with values replaced as follows:

- DB2 *`wcs_database_name`* is the name of your WebSphere Commerce database
- Oracle *`hostname`* is the Oracle host name
- Oracle *`port`* is the port number of the Oracle database (for example, 1521).
- *`wcs_db_user_name`* is the user name for the database.
- *`wcs_db_password`* is the database password.

Click **OK**.
The Select Tables window opens.

    e. From the **Qualifiers** list, select your database qualifier (this may be your database user name or your machine name) and click **Build Table List**. A list of the available database tables is loaded.
    f. Select **Bonus** from the Tables panel and click **OK**. Wait a few moments.
    g. In the Schema Browser, click the newly added table to see that both columns from the table appear.
    h. Right-click the **Bonus** table and select **Edit Table**.
    The Table Editor opens.

i. Remove any entries from the **Qualifier** field. It is good practice to remove the qualifier information so that the code can be deployed to other machines using a different database.

j. ➤ Oracle Modify the column data types, as follows:

   1) Select the **MEMBERID** column, then click **Edit**. From the Type drop-down list, select **BIGINT** and click **OK**.

   2) Select the **BONUSPOINT** column, then click **Edit**. From the Type drop-down list, select **INTEGER** and click **OK**.

k. Click **OK** to exit the Table Editor.

l. From the **Schemas** menu, select **Save Schema**.
The Save Schema window opens.

m. Enter the following information:

Attribute	Value
Project	`_WCSamplesEntityBeansProject`
Package	`com.ibm.commerce.sample.objects`
Class Name	`WCSSamplesSchema`

then click **Finish** and close the Schema Browser.

12. Once the schema has been created, you can create the schema map. To create this map between the BONUS table and the BonusBean entity, do the following:

a. From the **EJB** menu, select **Open To > Schema Maps**. The Map Browser opens.

b. In the Map Browser, from the **Datastore Maps** menu, select **New EJB Group Map**.
The New Datastore Map window opens.

c. Fill in the following information:

Attribute	Value
Name	`WCS Samples`
EJB Group	`WCSSamplesEntityBeans`
Schema	`WCSSamples`

and click **OK**.

d. In the Datastore Maps panel, click **WCS Samples**.

e. In the Persistent Classes panel, click **Bonus**.

f. From the **Table Maps** menu, select **New Table Map > Add Table Map with No Inheritance**.

g. From the **Table** drop-down list, select **Bonus** and click **OK**.

h. In the Table Maps panel, select **Bonus** then right-click it and select **Edit Property Maps**.
The Property Map Editor opens.

i. Set the attributes as follows:

Class Attribute	Map Type	Table Column
memberId	Simple	MEMBERID
bonusPoint	Simple	BONUSPOINT

and click **OK**.

j. From the **Datastore Maps** menu, select **Save Datastore Map**.
The Save Datastore Map opens.

k. Enter the following information:

Attribute	Value
Project	`_WCSamplesEntityBeansProject`
Package	`com.ibm.commerce.sample.objects`
Class Name	`WCSSamplesMap`

then click **Finish** and close the Map Browser.

13. Once the BonusBean entity has been created and the schema is correctly mapped, you must create an access bean for the entity bean. This access bean makes it simpler for applications to access information contained in the Bonus entity bean. The tools in VisualAge for Java are used to generate this access bean, based upon the entity that you have already created (in particular, only methods that have been promoted to the remote interface will be used by the access bean). To create the access bean for your Bonus entity bean, do the following:

a. In the Workbench, with the EJB tab selected, right-click the **Bonus** enterprise bean and select **Add > Access Bean**.
The Create Access Bean SmartGuide window opens (this may take a moment).

b. Ensure the following information is entered:

Attribute	Value
EJB Group	`WCSSamplesEntityBeans`
Enterprise Bean	`Bonus`
Access Bean Name	`BonusAccessBean`
Access Bean Type	`Copy Helper for an Entity Bean`

and click **Next**.

c. From the **Select home method for zero argument constructor** drop-down list, select **findByMemberId(Long)**.
d. For **init_argMemberId**, (in the Initial Properties column), set **Converter** to `com.ibm.commerce.base.objects.WCSStringConverter` and click **Next**.
e. For bonusPoint, ensure that **CopyHelper** is selected, set the **Converter** value to `com.ibm.commerce.base.objects.WCSStringConverter`, and click **Finish**.

   Note: You do not need to make any modifications for the memberId field.
f. Click **OK** when the “Code Generation Complete” message is displayed.

You can view the newly generated code by switching to the **Projects** tab, expanding the **_WCSamplesEntityBeansProject** project and then expanding **com.ibm.commerce.sample.objects**. A new class called `BonusAccessBean` is displayed inside the package.

14. The next step is to generate deployed code.
The code generation utility analyzes the beans to ensure that Sun Microsystems’ EJB specifications are met and it ensures that rules specific to the EJB server are followed. In addition, for each selected enterprise bean, the code-generation tool generates the home and EJBObject (remote) implementations and implementation classes for the home and remote interfaces, as well as the JDBC persister and finder classes for CMP beans. It also generates the Java ORB, stubs, and tie classes required for RMI access over IIOP, as well as stubs for the home and remote interfaces.
If you selected an EJB group containing a CMP enterprise bean, or if you selected an individual CMP enterprise bean, the following items are also generated:
   - A create table string that is generated into the persister class.
   - A Persister implementation that maps to and from the table

   To generate the deployed code, do the following:
   a. With the EJB tab selected, in the Enterprise Beans panel, right-click the **Bonus** enterprise bean and select **Generate Deployed Code**. The generation of code takes a few minutes.
15. Before testing the Bonus enterprise bean, you must create a new EJB server that contains all of the WebSphere Commerce EJB groups and plus the new WCSSamplesEntityBeans EJB group. These groups must be running on the same server in order for transaction scope to be maintained. Once you have created the new server, you must start it.
To create and start the new EJB server, do the following:
   a. Ensure that all of the EJB groups are collapsed.

b. In the Enterprise Beans pane, select all of the WebSphere Commerce EJB groups and your new EJB group (that is, select all EJB groups beginning with `WCS`).

c. With these EJB groups selected, right-click and select **Add To > Server Configuration**.
The EJB Server Configuration window opens (this may take a moment).

d. Right-click the newly created EJB Server {for example **EJB Server (server2)**}, select **Properties** and fill in the following:

Attribute	DB2 **DB2 value**	Oracle **Oracle value**
Data Source	`WebSphere Commerce DB2 DataSource` *`instance_name`*	`WebSphere Commerce Oracle DataSource` *`instance_name`*
Connection Type	`<DataSource>`	`<DataSource>`
User Name	*`wcs_db_user_name`*	*`wcs_db_user_name`*
Password	*`wcs_db_password`*	*`wcs_db_password`*
Transaction timeout	`1200`	`1200`
Transaction inactivity timeout	`600000`	`600000`

and click **OK**.

Note: The value for Data Source must match the value for the data source that is specified in the *`instance_name`*`.xml` file.

Depending upon the hardware of your development machine (for example, processor speed) you may need to increase the values for the **Transaction Timeout** and **Transaction Inactivity** EJB server properties.

e. If you have the WebSphere Test Environment running, stop it as well as the persistent name server and other EJB server, as described in Appendix A, "Starting and stopping the WebSphere Test Environment" on page 333.

f. Start the persistent name server, as described in "Starting and stopping the persistent name server" on page 333.

g. Start the new EJB server, as described in "Starting and stopping the EJB server" on page 334.

16. Once the EJB server has been started, you can start the test client. Using the test client, you create a new record in the database. To start the test client and create this record, do the following:

    a. With the EJB tab selected, right-click the **Bonus** enterprise bean and select **Run test client**.

b. In the EJB Lookup window, click **Lookup**.
   The Bonus window opens.

c. Click **create(Long, Integer)**. In the Details pane, fill in the following:

Attribute	Value
Long	`-1000`
Integer	`100`

and click the Invoke icon in the EJB Test Client window.

Note: The first attribute must match the memberId for any registered user. You can determine the memberId's by looking in the USERS table.

17. Verify that the database record was created correctly, by directly querying the database.

DB2 If you are using a DB2 database, do the following:

a. Open the DB2 Command Center (**Start > Programs > IBM DB2 > Command Center**)
b. Select the **Interactive** tab.
c. Enter `connect to` *`wcs_database_name`* and click the Execute icon.
d. Enter `SELECT * FROM Bonus` and click the Execute icon.
   The following values should be returned

Column	Value
MEMBERID	`-1000`
BONUSPOINT	`100`

The record shown above was created by your EJB test client.

Oracle If you are using an Oracle database, do the following:

a. Open the Oracle SQL Plus command window (**Start > Programs > Oracle - OraHome81> Application Development > SQL Plus**).
b. In the **User Name** field, enter your Oracle user name.
c. In the **Password** field, enter you Oracle password.
d. In the **Host String** field, enter your connect string.
e. In the SQL Plus window, enter the following:

```
select * from BONUS;
```

and click Enter to run the SQL statement.
The following values should be returned

Column	Value
MEMBERID	`-1000`
BONUSPOINT	`100`

18. Use the test client to verify that the Bonus enterprise bean can successfully access the database record, by doing the following:
    a. In the Bonus window, select the **Home** tab.
    b. In the Methods panel, click **findByMemberId(Long)**.
    c. In the **Long** field, enter `–1000` and click the Invoke icon.
    d. With the **Remote** tab selected, expand **Methods**, click **getBonusPoint** and then click the Invoke icon.
       The Details panel shows an integer result of `100`.
    e. Close the Bonus and EJB Test Client windows.

Integrate the Bonus entity bean with MyNewControllerCmd

In the previous section, you tested the new Bonus entity bean using the test client that was generated within VisualAge for Java. In this section, you integrate the Bonus entity bean with the MyNewControllerCmd logic. Once the Java code is updated, the Sample.jsp template is updated to create an interface that allows updates to a shopper's balance of bonus points.

Integrating the Bonus entity bean involves the following high-level steps:
1. Modify the performExecute method of the MyNewTaskCmdImpl class to calculate the new bonus points and save the points to the BONUS table.
2. Add a getResources method to the MyNewControllerCmdImpl class to return a list of resources that the command uses. This method is included for access control purposes.
3. Create a new access control policy for the new resources.
4. Modify DataBeanSampleBean to extend from the access bean for the Bonus entity bean. By having the data bean extend from the access bean, all attributes from the access bean are inherited by the data bean.
5. Modify methods in the DataBeanSampleBean.
6. Modify the class path for the servlet engine in the WebSphere Test Environment to include the new _WCSamplesEntityBeansProject.
7. Modify the Sample.jsp template to allow users to enter bonus points and display results.

Modify MyNewTaskCmdImpl for bonus point calculation:
MyNewTaskCmdImpl is used as the point of integration for the Bonus entity bean and MyNewControllerCmd (since MyNewControllerCmd invokes MyNewTaskCmd).

To modify MyNewTaskCmdImpl to perform the bonus point calculation, do the following:

1. In the VisualAge for Java Workbench window, expand the **_WCSamples** project.
2. Expand the **com.ibm.commerce.sample.commands** package, then select the **MyNewTaskCmdImpl** class to view its source code.
3. Uncomment the following import statement:

   ```
   import com.ibm.commerce.sample.objects.*;
   ```

   Save your work (**Ctrl + S**).
4. Select the **performExecute** method of the **MyNewTaskCmdImpl** class.
5. In the source code for the performExecute method uncomment Section 3. This introduces the following code into the method:

   ```
   // use BonusAccessBean to update new bonus point

   String newBonusPoint = null;
   BonusAccessBean bb = new BonusAccessBean();
   try {
        if (refNum != null) {
             bb.setInit_argMemberId(refNum);
             bb.refreshCopyHelper();
             oldBonusPoint = bb.getBonusPoint();
        }
   } catch (javax.ejb.FinderException e) {
        try {
             bb = new BonusAccessBean(new Long(refNum),new Integer(0));
             oldBonusPoint = "0";
        } catch (javax.ejb.CreateException ec) {
             throw new ECSystemException(ECMessage._ERR_CREATE_EXCEPTION,
                  this.getClass().getName(), "performExecute");
        } catch (javax.naming.NamingException ec) {
             throw new ECSystemException(ECMessage._ERR_NAMING_EXCEPTION,
                  this.getClass().getName(), "performExecute");
        } catch (java.rmi.RemoteException ec) {
             throw new ECSystemException(ECMessage._ERR_REMOTE_EXCEPTION,
                  this.getClass().getName(), "performExecute");
        }
   } catch (javax.naming.NamingException e) {
        throw new ECSystemException(ECMessage._ERR_NAMING_EXCEPTION,
             this.getClass().getName(), "performExecute");
   } catch (java.rmi.RemoteException e) {
        throw new ECSystemException(ECMessage._ERR_REMOTE_EXCEPTION,
             this.getClass().getName(), "performExecute");
   } catch (javax.ejb.CreateException e) {
        throw new ECSystemException(ECMessage._ERR_CREATE_EXCEPTION,
             this.getClass().getName(), "performExecute");
   }


   try {
   ```

```
        if (oldBonusPoint != null) {
            int newBP =  Integer.parseInt(oldBonusPoint) + getTask_input2();
            newBonusPoint = Integer.toString( newBP );
            bb.setBonusPoint( newBonusPoint )   ;
            newBonusPoint=bb.getBonusPoint();
            bb.commitCopyHelper();
        }
    } catch (javax.ejb.FinderException e) {
        throw new ECSystemException(ECMessage._ERR_FINDER_EXCEPTION,
            this.getClass().getName(), "performExecute");
    } catch (javax.naming.NamingException e) {
        throw new ECSystemException(ECMessage._ERR_NAMING_EXCEPTION,
            this.getClass().getName(), "performExecute");
    } catch (java.rmi.RemoteException e) {
        throw new ECSystemException(ECMessage._ERR_REMOTE_EXCEPTION,
            this.getClass().getName(), "performExecute");
    } catch (javax.ejb.CreateException e) {
        throw new ECSystemException(ECMessage._ERR_CREATE_EXCEPTION,
            this.getClass().getName(), "performExecute");
    }
```

Save your work.

Add getResources method to MyNewControllerCmdImpl class: In this section, you add a new getResources method to the MyNewControllerCmdImpl class. This method returns a list of resouces that the command uses during processing. This method is required for resource level access control.

To add the getResources method, do the following:

1. With the **Projects** tab selected, expand the **_WCSamples** project.
2. Expand the **com.ibm.commerce.sample.commands** package.
3. Select the **MyNewControllerCmdImpl** class to view its source code.
4. In the source code, uncomment the access control section. This section appears as shown in the following code snippet:

```
public AccessVector getResources() throws ECException{

   // use UserRegistryAccessBean to check member reference number

   String refNum;
   String methodName="getResources";

   com.ibm.commerce.user.objects.UserRegistryAccessBean rrb =
      new com.ibm.commerce.user.objects.UserRegistryAccessBean();

   try {
      rrb = rrb.findByUserLogonId(getInputString());
      refNum = rrb.getUserId();

   }
   catch (javax.ejb.FinderException e) {
      throw new ECSystemException(ECMessage._ERR_BAD_USER_NAME,
         this.getClass().getName(),methodName);

   }
   catch (javax.naming.NamingException e) {
      throw new ECSystemException(ECMessage._ERR_NAMING_EXCEPTION,
         this.getClass().getName(), methodName);
   }
   catch (java.rmi.RemoteException e) {
      throw new ECSystemException(ECMessage._ERR_REMOTE_EXCEPTION,
         this.getClass().getName(), methodName);
   }
   catch (javax.ejb.CreateException e) {
      throw new ECSystemException(ECMessage._ERR_CREATE_EXCEPTION,
      this.getClass().getName(), methodName);
   }


   //find the Bonus bean for this user
   String newBonusPoint = null;
   com.ibm.commerce.sample.objects.BonusAccessBean bb =
      new com.ibm.commerce.sample.objects.BonusAccessBean();
   try {
      if (refNum != null) {
         bb.setInit_argMemberId(refNum);
         bb.refreshCopyHelper();
      }
   }
catch (javax.ejb.FinderException e) {

   // The user doesn't have a Bonus object so return the container that
   // will hold the bonus object when it's created

   return new AccessVector(rrb);

   }
   catch (javax.naming.NamingException e) {
      throw new ECSystemException(ECMessage._ERR_NAMING_EXCEPTION,
         this.getClass().getName(), methodName);
   }

   catch (java.rmi.RemoteException e) {
      throw new ECSystemException(ECMessage._ERR_REMOTE_EXCEPTION,
      this.getClass().getName(), methodName);
   }

   catch (javax.ejb.CreateException e) {
      throw new ECSystemException(ECMessage._ERR_CREATE_EXCEPTION,
      this.getClass().getName(), methodName);
   }
```

```
        return new AccessVector(bb);

    }
```

Save your work. (Ctrl+S).

Once you save the preceding section of code, VisualAge for Java separates the fields and accessors out of this particular view. Notice that they now appear under the MyNewControllerCmdImpl class and the method is marked with an M.

Note: For simplicity in this tutorial, the resource objects are created in this getResources method. In a real application, it is preferable to create the resource objects in the validateParameters method and save them as instance variables. As such, the objects could then be reused by the getResources and performExecute methods.

Setting up the access control policy for the new resource: A sample access control policy is provided. This policy creates the following access control objects:

An action
The action that is created is `com.ibm.commerce.sample.commands.MyNewControllerCmd`

An action group
The action group that is created is `MyNewControllerCmdActionGroup`. This action group contains only one action; `com.ibm.commerce.sample.commands.MyNewControllerCmd`

A resource category
The resource category that is created is `com.ibm.commerce.sample.objects.BonusResourceCategory`. This resource category is for the Bonus entity bean.

A resource group
The resource group that is created is `BonusResourceGroup`. This resource group only contains the preceding resource category.

A policy
The policy that is created is `AllUsersUpdateBonusResourceGroup`. This policy allows users to perform the MyNewControllerCmd action on the Bonus bean only if the user is the “owner” of the bonus object. For example, if the user is logged on as the `wcsadmin` user, the user can only modify the bonus points for `wcsadmin`.

Setting up the `AllUsersUpdateBonusResourceGroup` policy involves the following steps:

1. Loading the `SampleACPolicy.xml` file using the `acpload` command.
2. Loading the `SampleACPolicy_`***`locale`***`.xml` description using the `acpnlsload` command.
3. Refreshing the policy registry. Note that this step is only required if the servlet engine is running at the time the access control policy is loaded.

To set up the `AllUsersUpdateBonusResourceGroup` policy, do the following:

1. At a command prompt, switch to the following directory:
   `drive``:\WebSphere\CommerceServerDev\bin`
2. To load the `SampleACPolicy.xml` file, you must issue the `acpload` command, which has the following form:

   ```
   acpload db_name db_user db_password inputXMLFile
   ```

   where
   - *`db_name`* is the name of your database
   - *`db_user`* is your database user name
   - *`db_password`* is your database password
   - *`inputXMLFile`* is the name of the XML file containing the policy

   For example, you may issue the following command:

   ```
   acpload VAJ_Demo user password SampleACPolicy.xml
   ```
3. To load the policy description, you must issue the `acpnlsload` command, which has the following form:

   ```
   acpnlsload db_name db_user db_password inputXMLFile
   ```

   For example, you may issue the following command:

   ```
   acpnlsload VAJ_Demo user password SampleACPolicy_en_US.xml
   ```
4. If the servlet engine for the WebSphere Test Environment is currently running, stop and then restart it to refresh the policy registry.

Modify DataBeanSampleBean for bonus points: In this section you modify the DataBeanSampleBean to extend the BonusAccessBean, by doing the following:

1. In the Workbench, expand the **com.ibm.commerce.sample.databeans** package.
2. Add a new field to the data bean, by doing the following:
   a. Right-click the **DataBeanSampleBean** class and select **Add > Field**. Using the Create Field SmartGuide, create new fields as described in the following steps. If you require additional information about creating fields, refer to “Creating new fields” on page 213.

b. Add a new field to the data bean, using the following values:

Attribute Name	Value
Field Name	`task_output_oldBonusPoint`
Field Type	`String`
Initial Value	Leave blank.
Access Modifiers	`private`
Other Modifiers	Leave all unchecked.
Access with getter and setter methods	`checked`
Getter	`public`
Setter	`public`

Click **Finish**.

3. Click the **DataBeanSampleBean** class to view its source code. In the source code, uncomment the following import statement:

```
import com.ibm.commerce.sample.objects.*;
```

Save your work.

4. Still within the source code for the DataBeanSampleBean class, uncomment Section 1 and comment out Section 2. This changes the bean to extend from the BonusAccessBean. After making these modifications, the code appears as follows:

```
/// Section 1 ///////////////////////////////////////////////

// Extend the databean to BonusAccessBean

public class DataBeanSampleBean
   extends com.ibm.commerce.sample.objects.BonusAccessBean
   implements SmartDataBean {


/////////////////////////////////////////////////////////////

/// Section 2 ///////////////////////////////////////////////////
/*
// Extend the databean to BonusAccessBean

     public class DataBeanSampleBean implements SmartDataBean {
*/

//
/////////////////////////////////////////////////////////////
```

Save your work.

5. Select the **setTask_output_userId(String)** method to view its source code. Locate the following line of code:

```
public void setTask_output_userId(java.lang.String newTask_output_userId) {
  task_output_userId = newTask_output_userId;
```

After the preceding line, enter the following code to instantiate the new BonusAccessBean:

```
///////////////////////////////////////
// Section A : instantiate BonusAccessBean

if (task_output_userId != null)
     this.setInit_argMemberId(newTask_output_userId);

///////////////////////////////////////
```

Save your work.

6. Select the **populate()** method to view its source code. Uncomment Section 2 to instantiate the BonusAccessBean. This introduces the following code into the method:

```
setTask_output_oldBonusPoint(getRequestProperties().getString(
    "task_output_oldBonusPoint"));
```

Modify the class path: Before you can test the modified command in the WebSphere Test Environment, you must modify a class path to include the new project that was created for the new Bonus entity bean. To modify this class path, do the following:

1. From the **Workspace** menu in VisualAge for Java, select **Tools > WebSphere Test Environment**.
   The WebSphere Test Environment Control Center opens.
2. Click **Servlet Engine**.
3. If the Servlet Engine is running, click **Stop Servlet Engine** and then **Edit Class Path**.
4. Click **Select All** and then click **OK**.

Modify the Sample.jsp template for bonus points: To modify the display template, do the following:

1. Open `Sample.jsp` and `Sample_All.jsp` files in a text editor.
2. Copy Section 4 of the code from Sample_All.jsp into Sample.jsp between the `<!-- SECTION 4 -->` and `<!-- END OF SECTION 4 -->` markers. The following code is introduced into the JSP template

```
<!-- SECTION 4 -->

<h1>
Bonus Administration
</h1>

<%
if (userId != null) {
```

```
%>

<B>
<UL>
   <LI> The bonus point before update is
      <%=testBean.getTask_output_oldBonusPoint()%>
   <LI> The bonus point after update is
      <%=testBean.getBonusPoint()%>
</UL>
</B>

<%
}
%>

<br>
<B>Input to command:</B><P>

<FORM NAME=Bonus ACTION="MyNewControllerCmd">
<TABLE>
   <TR>
        <TD>
             <B>Logon ID </B>
         </TD>
        <TD>
             <input type=text name=input1
                 value='<%=testBean.getInput1()%>'>
        </TD>
   </TR>
   <TR>
         <TD>
             <B>Bonus Point</B>
         </TD>
         <TD>
              <input type=text name=input2>
         </TD>
     </TR>
     <TR>
          <TD COLSPAN=2>
               <input type=submit>
          </TD>
     </TR>
</TABLE>
</FORM>

<!-- END OF SECTION 4 -->
```

Save the `Sample.jsp` file.

3. Since the new Bonus bean is protected under access control and users can only execute the MyNewControllerCmd action on a bean that they own, the user must log in. As such, you will use the login feature in your sample store to allow the user to log in. This requires you to copy the Sample.jsp file into the store's directory structure, since once you log into

the store, the Web controller will search for the Sample.jsp file in the store's directory. Copy the Sample.jsp file from:
`vaj_drive:\VAJava\Ide\project_resources\IBM WebSphere Test Environment \hosts\default_host\default_app\web`
to
`vaj_drive:\VAJava\Ide\project_resources\IBM WebSphere Test Environment \hosts\default_host\default_app\web\store_directory`.

Note: For the sample store, the value for *store_directory* is InFashion.

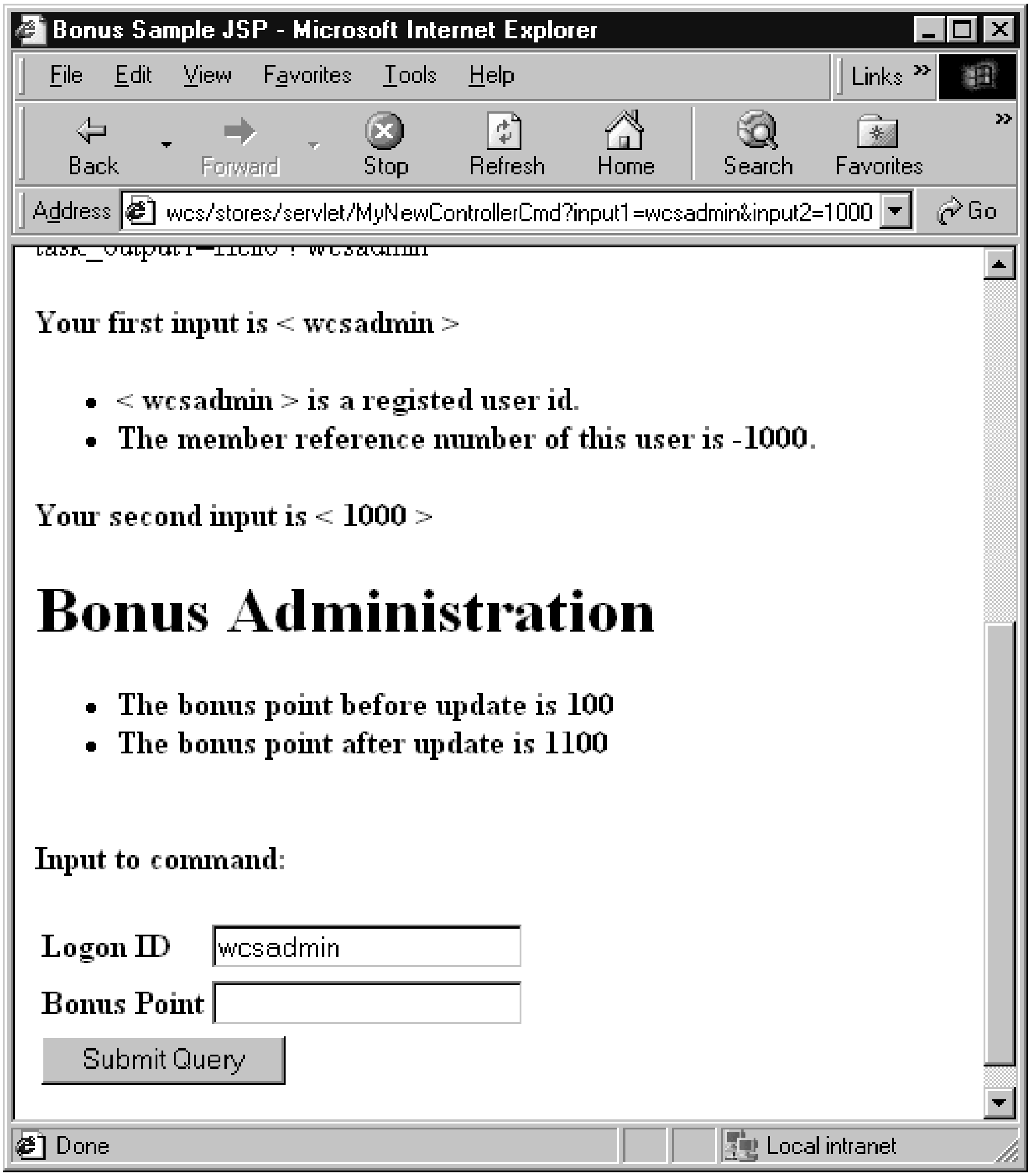
4. Ensure the WebSphere Test Environment is running (refer to Appendix A, "Starting and stopping the WebSphere Test Environment" on page 333).
5. Log in as the `wcsadmin` user, by doing the following:
   - Enter the following URL in a browser:
     ```
     http://localhost:8080/webapp/wcs/stores/servlet/StoreCatalogDisplay?
          storeId=store_Id&catalogId=catalog_Id&langId=-1
     ```
   - Click the **Register** link.
     The Register or Login page is displayed.
   - In the **E-mail address** field, enter `wcsadmin`.
   - In the **Password** field, enter the password for the `wcsadmin` user and then click **Login**.

     Note: The original password was `wcsadmin`, but this was changed when the `contractPublish` command was executed as part of the installation process. This step is described in the following document:

     – Business *WebSphere Commerce Studio Business Developer Edition Installation Guide*

     – Professional *WebSphere Commerce Studio Professional Developer Edition Installation Guide*
6. After the login is complete, enter the following URL in the same browser:
   ```
   http://localhost:8080/webapp/wcs/stores/servlet/MyNewControllerCmd?
        input1=wcsadmin&input2=1000
   ```

   You are presented with a page that contains all of the previous output parameters as well as a new form that allows you to update the bonus point balance for a user. The page displayed is similar to the following screen shot:


Bonus Sample JSP - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Links
Back Forward Stop Refresh Home Search Favorites
Address
Go
Done
Local intranet

wcs/stores/servlet/MyNewControllerCmd?input1=wcsadmin&input2=1000

Your first input is < wcsadmin >

- < wcsadmin > is a registed user id.
- The member reference number of this user is -1000.

Your second input is < 1000 >

Bonus Administration

- The bonus point before update is 100
- The bonus point after update is 1100

Input to command:

Logon ID	wcsadmin
Bonus Point	

Submit Query

Figure 40.

7. Create a version of your code in its current state. Name the version `mySample 1.7 Completed`. When versioning the code, be sure to select both of your projects. If you require detailed information about versioning your code, refer to step 12 on page 208.

(Optional) Using the Debugger in VisualAge for Java

This section shows you how to add a breakpoint into your code and launch the Debugger component of VisualAge for Java. This is included to introduce the Debugger. For details about this powerful feature, refer to the online help for VisualAge for Java.

This section is optional, because it does not introduce any new code that is required for the tutorial. Instead, you create a breakpoint, verify the values of some variables at the breakpoint and then remove the breakpoint.

Adding the breakpoint to your code

To add the breakpoint to your code, do the following:

1. Ensure that the use of breakpoints is enabled in your workspace, by doing the following:
   a. From the **Window** menu, select **Debug**. Ensure that **Global Enable Breakpoints** is selected.
2. Select the **Projects** tab.
3. Expand the **_WCSamples** project.
4. Expand the **com.ibm.commerce.sample.commands** package.
5. Expand the **MyNewTaskCmdImpl** class.
6. Select the **performExecute** method, to view its source code.
7. In the Source pane, place your cursor (and click) at the beginning of the following line of code:
   ```
   setTask_output1( "Hello ! " + getTask_input1() );
   ```
   Leave your cursor in this position.
8. From the **Edit** menu, select **Breakpoint**.
9. In the Configuring: Breakpoint #1 window, select **In Selected Thread** and click **OK**.
10. Ensure the WebSphere Test Environment is running (refer to Appendix A, “Starting and stopping the WebSphere Test Environment” on page 333).
11. Log in as the `wcsadmin` user, by doing the following:
    - Enter the following URL in a browser:
      ```
      http://localhost:8080/webapp/wcs/stores/servlet/StoreCatalogDisplay?
           storeId=store_Id&catalogId=catalog_Id&langId=-1
      ```
    - Click the **Register** link.
      The Register or Login page is displayed.

- In the **E-mail address** field, enter `wcsadmin`.
- In the **Password** field, enter the password for the wcsadmin user and then click **Login**.

12. After the login process is complete, enter the following URL:

```
http://localhost:8080/webapp/wcs/stores/servlet/MyNewControllerCmd?
     input1=wcsadmin&input2=1000
```

13. The Debugger window opens when the breakpoint in the code is reached.

Verifying the values of variables

This section shows how to verify the values of the `task_input1` and `task_output1` variables at various points during execution of the command.

When the Debugger opens because of the breakpoint in the performExecute method of the MyNewTaskCmdImpl, switch to that window and do the following:

1. In the Variable pane, expand **this**.
2. Click **task_input1**.
   In the Value pane, the value for this variable at this point during execution is displayed. It should display `wcsadmin`.

To verify that the value of the `task_output1` variable gets set as expected you must have the Debugger continue executing the performExecute method to reach the point in the code where that variable gets set. This can be done as follows:

1. Use the Step over function to move step by step through the code. This can be done in one of the following ways:
   - From the **Selected** menu, select **Step Over**.
   - Click F6.
   - Click the Step Over icon.

   For the purpose of this example, click F6 four times. This executes the code that sets the task_output1 variable.
2. In the Variable pane, click task_output1.
   In the Value pane, the value for this variable at this point during execution is displayed. It should display `Hello ! wcsadmin`.
3. You can finish executing the command by clicking the Resume icon.

Removing the breakpoint

To remove the breakpoint from your code, do the following:

1. Switch back to the Workbench window.
2. Ensure that you can view the source code for the `performExecute` method of the `MyNewTaskCmdImpl` class.

3. In the Source pane, navigate to the ollowing line of code:
   ```
   setTask_output1( "Hello ! " + getTask_input1() );
   ```
   Notice that in the left margin of the pane, there is a blue dot to mark the breakpoint.
4. Double-click the blue dot to remove the breakpoint.

The breakpoint is now removed from your code.

Integrating MyNewControllerCmd with the sample store in the WebSphere Test Environment

In this section, you add a link to the home page for the sample store that invokes MyNewControllerCmd. To perform this integration step, do the following:

1. In a text editor, open the `sidebar.jsp` file. This file is located in the following directory:
   `*vaj_drive*:\VAJava\ide\project_resources\IBM WebSphere Test Environment \hosts\default_host\default_app\web\*store_directory*\include`
   where *vaj_drive* is the drive on which you installed VisualAge for Java and *store_directory* is the name of the directory for the sample store.
2. Add another row with a link to MyNewControllerCmd into the table by inserting the following code before the final `</table>` tag:
   ```
   <tr>
   <td>
   <a href = "MyNewControllerCmd?input1=wcsadmin&input2=1000">
        MyNewControllerCmd</a>
   </td>
   </tr>
   ```
   Save your work.
3. Ensure the WebSphere Test Environment is running.
4. Test the integration by entering the following URL in a browser:
   ```
   http://localhost:8080/webapp/wcs/stores/servlet/StoreCatalogDisplay?
        storeId=store_Id&catalogId=catalog_Id&langId=-1
   ```
   Click the **Register** link.
   The Register or Login page is displayed.
5. In the **E-mail address** field, enter `wcsadmin`.
6. In the **Password** field, enter the password for the wcsadmin user and then click **Login**.
7. After the user is logged on, click the **MyNewControllerCmd** link in the side navigation pane. The Sample JSP is displayed.

Note: If the side navigation pane does not display the new link, the sidebar.jsp may be cached. Remove it from the cache and reload the page. Refer also to Appendix C, “Tips for VisualAge for Java” on page 369 for information about deleting compiled JSP files. You may need to restart your servlet engine after deleting compiled JSP files.

(Optional) Deploying new business logic to a remote WebSphere Commerce Server

This section describes how to deploy your new business logic into a store running on a remote WebSphere Commerce Server. You must have created a store (based upon the InFashion sample store) on the remote WebSphere Commerce Server before starting these deployment steps.

The deployment process includes steps that are performed on the development machine, as well as steps that are performed on the target WebSphere Commerce Server.

Create the JAR file for the new command logic

You must create a JAR file that contains the new command and data bean logic. To create this JAR file, do the following:

1. Stop the WebSphere Test Environment, as described in Appendix A, “Starting and stopping the WebSphere Test Environment” on page 333.
2. With the Projects tab selected, select the **_WCSamples** project.
3. With the project highlighted, right-click and select **Export**.
   The Export SmartGuide opens.
4. Select **Jar file** and click **Next.**
5. In the Jar file field, enter the following:
   `*drive*:\WebSphere\CommerceServerDev\mytemp\wcssamples_1.jar`
   where *`drive`* is the drive on which Commerce Studio is installed.
6. Select attributes as follows:

Attribute	Value
class	Checked
java	Unchecked
resource	Checked
beans	Checked
Include debug attributes in .class file	Checked

   Accept the default values for other attributes.
7. Click **Finish**.
8. If prompted, confirm the creation of the new directory.

Since the JAR file created does not contain complete package naming information, you must use another packaging utility (outside of VisualAge for Java) to repackage the JAR file. To repackage this file, do the following:

1. In a command window, navigate to the following directory:
   `drive:\WebSphere\CommerceServerDev\mytemp`
2. Enter `mkdir temp1`.
3. Enter `cd temp1`.
4. Set the path as follows:
   `set PATH=%PATH%;drive:\WebSphere\WebSphereStudio4\bin;`
   where *drive* is the drive on which WebSphere Studio is installed.
5. Enter `jar xvf ../wcssamples_1.jar`
6. Enter `jar cvf ../wcssamples.jar *` (note that the `_1` is removed from the name).

Creating the JAR file for the new EJB group

You must create a JAR file for your new EJB group. To create this file, do the following:

1. With the EJB tab selected, right-click the **WCSSamplesEntityBeans** EJB group and select **Export > EJB 1.1 JAR**.
   The Export to an EJB 1.1 JAR File SmartGuide opens.
2. In the **JAR file** field, enter
   `drive:\WebSphere\CommerceServerDev\mytemp\sampleEntityBeans_DT.jar`
3. Select attributes as follows:

Attribute	Value
class	Checked
java	Unchecked
resource	Checked
Target database	DB2 If you are deploying to a DB2 database, select **DB2 for NT, V7.1**. Oracle If you are deploying to an Oracle database, select **Oracle, V8**.
Include debug attributes in .class file	Checked

   Accept the default values for other attributes.
4. Click **Finish**.

The JAR file is created.

The JAR file has been named with the "_DT" suffix as a reminder that you must run this JAR file through the EJB Deploy Tool provided by WebSphere Application Server before deploying it into your WebSphere Commerce application.

Creating the implementation JAR file for the new enterprise bean

You must create a JAR file containing the implementation code for the Bonus enterprise bean. To create this file, do the following:

1. With the project tab selected, right-click the **_WCSSamplesEntityBeansProject** and select **Export**.
   The Export SmartGuide opens.
2. Select **Jar file** and click **Next.**
3. In the **JAR file** field, enter
   `*drive*:\WebSphere\CommerceServerDev\mytemp\sampleImpl_1.jar`
4. Select attributes as follows:

Attribute	Value
class	Checked
java	Unchecked
resource	Checked
beans	Checked
Include debug attributes in .class file	Checked

   Accept the default values for other attributes.
5. Click **Finish**.

The JAR file is created. Close VisualAge for Java.

Since the JAR file created does not contain complete package naming information, you must use another packaging utility (outside of VisualAge for Java) to repackage the JAR file. To repackage this file, do the following:

1. In a command window, navigate to the following directory:
   `*drive*:\WebSphere\CommerceServerDev\mytemp`
2. Enter `mkdir temp2`.
3. Enter `cd temp2`.
4. Set the path as follows:
   `set PATH=%PATH%;*drive*:\WebSphere\WebSphereStudio4\bin;`
   where *drive* is the drive on which WebSphere Studio is installed.
5. Enter `jar xvf ../sampleImpl_1.jar`

6. Enter `jar cvf ../sampleImpl.jar *` (note that the `_1` is removed from the name).

Copy the JSP files to the target WebSphere Commerce Server

You must copy the `Sample.jsp` and the updated `sidebar.jsp` files into the correct directories for the store to which you are deploying the code.

Before you copy the updated JSP templates onto the target WebSphere Commerce Server, you may want to make backup copies of the original JSP templates on that machine. That is, rename the existing file to `sidebar.jsp.bak`.

To copy these files, do the following:

1. On the development machine, navigate to the following directory: *vaj_drive*`:\VAJava\Ide\project_resources\IBM WebSphere Test Environment \hosts\default_host\default_app\web\`*store_directory*
   where *vaj_drive* is the drive on which you installed VisualAge for Java and *store_directory* is the name of the directory for the store.
   Copy the `Sample.jsp`.
2. Paste the `Sample.jsp` into the following directory on the target WebSphere Commerce Server:
   ```
   drive:\WebSphere\AppServer\installedApps\
      WC_Enterprise_App_instance_name.ear\
      wcstores.war\store_directory
   ```
   where *drive* is the drive on which WebSphere Commerce is installed, *store_directory* is the directory name for the store, and *instance_name* is the name of your WebSphere Commerce instance.
3. On the development machine, navigate to the following directory: *vaj_drive*`:\VAJava\Ide\project_resources\IBM WebSphere Test Environment`
   `\hosts\default_host\default_app\web\`*store_directory*`\include`
   Copy the `sidebar.jsp`
4. Paste the `sidebar.jsp` into the following directory on the target WebSphere Commerce Server:
   ```
   drive:\WebSphere\AppServer\installedApps\
      WC_Enterprise_App_instance_name.ear\
      wcstores.war\store_directory\include
   ```

Copy the JAR files to the target WebSphere Commerce Server

You must copy the JAR files from the development machine into the appropriate directory on the target WebSphere Commerce Server.

Additionally, there are two further processing steps that must be performed on the JAR file containing the new EJB group. First, you must run the EJB

Deploy Tool from WebSphere Application Server against the file. Then, you must run the modifyIsolationLevel command against the file. As a result, in this step of copying the JAR files onto the target WebSphere Commerce Server, this particular JAR file gets stored in a temporary directory to await those further steps.

To copy these files, do the following:

1. On the development machine, navigate to the following directory: `*drive*:\WebSphere\CommerceServerDev\mytemp` and locate the following files:
   - `wcssamples.jar`
   - `sampleImpl.jar`
   - `sampleEntityBeans_DT.jar`

   where *drive* is the drive onto which you installed WebSphere Commerce Studio, Business Developer Edition.

   Each of the preceding files must be copied into a particular directory on the target WebSphere Commerce Server. Read the following steps carefully to ensure that each file is stored in the correct location.
2. Copy the `wcssamples.jar` file into the following directory on the target WebSphere Commerce Server:

   ```
   drive:\WebSphere\AppServer\InstalledApps\
   WC_Enterprise_App_instance_name.ear\wcstores.war\WEB-INF\lib
   ```

   where *drive* is the drive onto which you installed WebSphere Commerce Business Edition and *instance_name* is the name of your instance (for example, `demo`).
3. Create a temporary library directory into which you will place the JAR file. That is, create the following diretory:

   ```
   drive:\WebSphere\CommerceServer\temp\lib
   ```
4. Copy the `sampleImpl.jar` file into the following directory the target WebSphere Commerce Server:

   ```
   drive:\WebSphere\CommerceServer\temp\lib
   ```
5. Copy the `sampleEntityBeans_DT.jar` file into the following directory on the target WebSphere Commerce Server:

   ```
   drive:\WebSphere\CommerceServer\temp
   ```

Running the EJB deploy tool

Before you can successfully run your enterprise beans on either a test or production server, you need to generate deployment code for the enterprise beans. The Deployment Tool for Enterprise JavaBeans (also referred to as the

EJB Deploy Tool) that is provided by WebSphere Application Server, provides a command-line interface that you can use to generate enterprise bean deployment code.

To run this tool, do the following:

1. At a command prompt, navigate to the following directory:

   ```
   drive:\WebSphere\CommerceServer\temp
   ```
2. Temporarily add the tool to the system path by entering the following command:

   ```
   PATH=drive:\WebSphere\AppServer\deploytool;%PATH%
   ```
3. Enter the `ejbdeploy` command as follows:

   ```
   ejbdeploy EJBGroupJARFile WorkingDir OutputJARFile -nowarn -keep -35 -cp
      ClassPathOfDepJARFiles
   ```

   where:

   - *`EJBGroupJARFile`* is the fully qualified name the JAR file of the enterprise beans for which you want to generate deployed code. In this case, this is `drive:\WebSphere\CommerceServer\temp\sampleEntityBeans_DT.jar`.
   - *`WorkingDir`* is name of the directory where temporary files that are required for code generation are stored.
   - *`OutputJARFile`* is the fully qualified name of the output JAR file. In this case, enter `drive:\WebSphere\CommerceServer\temp\sampleEntityBeans.jar`.
   - `-nowarn` is an optional parameter to suppress warning and information messages.
   - `-keep` is an optional parameter to retain the working directory after the ejbdeploy command has run.
   - `-35` is a mandatory parameter that will use the same top-down mapping rules for CMP entity beans that are used in the EJB Deploy Tool that was provided with the WebSphere Application Server, Version 3.5.
   - `-cp` *`ClassPathOfDepJARFiles`* is the class path of any dependent JAR files. In this case, enter

     ```
     "drive:\WebSphere\CommerceServer\temp\lib\sampleImpl.jar;
     drive:\WebSphere\AppServer\installedApps\
        WC_Enterprise_App_instanceName.ear\lib\wcsejbimpl.jar"
     ```

     Note: You must enclose the class path value in quotation marks (“”).

Modify transaction isolation level for the Bonus bean

In this step you use the `modifyIsolationLevel` command to modify the transaction isolation level of the Bonus bean. This tool also sets the isolation level of the bean to the required level for your specific type of database.

To run the `modifyIsolationLevel` command, do the following:

1. On the target WebSphere Commerce Server, open a command window.
2. Switch to the following directory:
   `drive:\WebSphere\CommerceServer\bin`
3. You must issue the `modifyIsolationLevel` command which has the following general syntax:

   ```
   modifyIsolationLevel -jarFile jar_file_name.jar
      -logFile log_file_name -dbType db_type
   ```

   where

   - *jar_file_name.jar* is the name of the JAR file that contains the customized code
   - *log_file_name* is the fully qualified file name where information should be logged
   - *db_type* is the type of database you are using. Enter either DB2 or ORACLE

   The following is an example of the `modifyIsolationLevel` command with all values specified:

   ```
   modifyIsolationLevel -jarFile
      D:\WebSphere\CommerceServer\temp\sampleEntityBeans.jar
      -logFile D:\WebSphere\CommerceServer\instances\demo\logs\output.log
      -dbType DB2
   ```

The command has run successfully if no exceptions are displayed in the command window. After completion, note that the timestamp on your deployed JAR file has changed.

Note: The parameter names are case sensitive. That is, jar**F**ile is not the same as jar**f**ile. Ensure that you enter the parameter names correctly.

Updating the target database

Since you are deploying the new business logic to a target WebSphere Commerce Server that uses a different database than that which is used by the WebSphere Test Environment, you must update the target database to reflect the changes that were made to the command registry and to create the BONUS table.

DB2 If you are using a DB2 database, do the following to update your target database:

1. Open the DB2 Command Center (**Start > Programs >IBM DB2 > Command Center**).
2. From the **Tools** menu, select **Tools Settings**.

3. Select the **Use statement termination character** checkbox and ensure the character specified is a semicolon (;)
4. With the Script tab selected, create the required entry in the URLREG table, by entering the following information in the script window:

   ```
   connect to your_database_name;
   insert into URLREG (URL, STOREENT_ID, INTERFACENAME, HTTPS,
       DESCRIPTION, AUTHENTICATED) values ('MyNewControllerCmd',0,
       'com.ibm.commerce.sample.commands.MyNewControllerCmd',0,
       'This is a new controller command for test/education purposes.',
       null)
   ```

   where *your_database_name* is the name of your database and click the Execute icon.
   This command is used by all merchants (indicated by the 0 value for STOREENT_ID).
5. Create an entry in the VIEWREG table, by entering the following in the script window:

   ```
   insert into VIEWREG (VIEWNAME, DEVICEFMT_ID, STOREENT_ID, INTERFACENAME,
      CLASSNAME, PROPERTIES, DESCRIPTION, HTTPS, LASTUPDATE) values
      ('SampleViewTask',-1, 0, 'com.ibm.commerce.command.ForwardViewCommand',
      'com.ibm.commerce.command.HttpForwardViewCommandImpl',
      'docname=Sample.jsp','This is a sample view for the Bonus Point
      exercise', 0, null)
   ```

   and click the Execute icon.
6. Create the BONUS table by entering the following in the script window:

   ```
   CREATE TABLE Bonus (MEMBERID BIGINT NOT NULL,
       BONUSPOINT INTEGER NOT NULL, constraint p_memberid primary key (MEMBERID),
       constraint f_memberid foreign key (MEMBERID)
       references users (users_id) on delete cascade)
   ```

   Click the Execute icon.

The preceding steps register MyNewControllerCmd and SampleViewTask in the command registry, as well as create the BONUS table.

Oracle If you are using an Oracle database, do the following to update your target database:

1. Open the Oracle SQL Plus command window (**Start > Programs > Oracle > Application Development > SQL Plus**).
2. In the **User Name** field, enter your Oracle user name.
3. In the **Password** field, enter you Oracle password.
4. In the **Host String** field, enter your connect string.
5. Create the required entry in the URLREG table, by entering the following information in the SQL Plus window:

```
insert into URLREG (URL, STOREENT_ID, INTERFACENAME, HTTPS,
    DESCRIPTION, AUTHENTICATED) values ('MyNewControllerCmd',0,
    'com.ibm.commerce.sample.commands.MyNewControllerCmd',0,
    'This is a new controller command for test/education purposes.',
    null);
```

and press Enter to run the SQL statement.

6. Create an entry in the VIEWREG table, by entering the following in the SQL Plus window:

   ```
   insert into VIEWREG (VIEWNAME, DEVICEFMT_ID, STOREENT_ID, INTERFACENAME,
      CLASSNAME, PROPERTIES, DESCRIPTION, HTTPS, LASTUPDATE) values
      ('SampleViewTask',-1, 0, 'com.ibm.commerce.command.ForwardViewCommand',
      'com.ibm.commerce.command.HttpForwardViewCommandImpl',
      'docname=Sample.jsp','This is a sample view for the Bonus Point
      exercise', 0, null);
   ```

   and press Enter to run the SQL statement.
7. Create the BONUS table by entering the following in the SQL Plus window:

   ```
   CREATE TABLE Bonus (MEMBERID NUMBER NOT NULL,
       BONUSPOINT INTEGER NOT NULL, constraint p_memberid primary key (MEMBERID),
       constraint f_memberid foreign key (MEMBERID)
       references users (users_id) on delete cascade);
   ```

   and press Enter to run the SQL statement.
8. Enter the following to commit your database changes:

   ```
   commit;
   ```

   and press Enter to run the SQL statement.

Loading the access control policies for the new resources

In the tutorial, you created a new enterprise bean (the Bonus bean) that is a protectable resource. As such, there is an access control policy related to this resource. You also created a new controller command that can be executed by all users. While working on the development machine, you loaded the access control policy information onto that machine. You must now load the same accress control policy information onto the target WebSphere Commerce Server.

To set up the access control policies, do the following:

1. Insert the following CD into your CD drive:
   - Business WebSphere Commerce Business Edition, V5.4 Disk 2 CD
   - Professional WebSphere Commerce Professional Edition, V5.4 Disk 2 CD
2. Switch to the following directory:

   CD_drive:\repository\samples\programguide\

3. Locate the following files in that directory:
   - `SampleCmdACPolicy.xml`
     This XML file contains the access control policy that is used by the new controller command.
   - `SampleACPolicy.xml`
     This XML file contains the access control policy that is used when you create a new enterprise bean.
   - `SampleACPolicy_`*`locale`*`.xml`
     where *`locale`* is the language identifier. This XML file contains the access control policy description.
4. Copy the preceding three files into the following directory:
   `drive``:\WebSphere\CommerceServer\xml\policies\xml`
5. To load the `SampleCmdACPolicy.xml` file, use a command prompt to switch to the following directory:
   `drive``:\WebSphere\CommerceServer\bin`
   You must issue the `acpload` command, which has the following form:
   ```
   acpload db_name db_user db_password inputXMLFile
   ```
   where
   - *`db_name`* is the name of your database
   - *`db_user`* is your database user name
   - *`db_password`* is your database password
   - *`inputXMLFile`* is the name of the XML file containing the policy.

   For example, you may issue the following command:
   ```
   acpload mall user password SampleCmdACPolicy.xml
   ```
6. Load the `SampleACPolicy.xml` file, by issuing the `acpload` command specifying the `SampleACPolicy.xml` file as the input file. For example, you may issue the following command:
   ```
   acpload mall user password SampleACPolicy.xml
   ```
7. To load the policy description, you must issue the `acpnlsload` command, which has the following form:
   ```
   acpnlsload db_name db_user db_password inputXMLFile
   ```
   For example, you may issue the following command:
   ```
   acpnlsload mall user password SampleACPolicy_en_US.xml
   ```

Note that normally a registry refresh would be required in order for the policy to take effect. In this case, you are not required to perform this step since you will be stopping and restarting the WebSphere Commerce Server application in WebSphere Application Server, as part of the deployment steps for the enterprise bean. If this were not the case, you could use the Administration console in WebSphere Commerce to update the registry. For more information about the Administration console, refer to the WebSphere Commerce online help.

400 If you were deploying to a WebSphere Commerce instance running on an iSeries machine, you use different commands to load access control policy information. Use the `LODWCSAC` command instead of the `acpload` command and the `LODWCSACD` command instead of the `acpnlsload` command.

The syntax for the `LODWCSAC` command is:

```
LODWCSAC DATABASE(dbName) SCHEMA(schemaName)
PASSWD(instancePassword) INSTROOT('instanceRoot')
INFILE('inputFile')
```

where

- *dbName* is the name of their relational database as defined in the WRKRDBDIRE command.
- *schemaName* is the name of the database schema for the instance (this is the same name as the instance name).
- *instancePassword* is the instance password.
- *instanceRoot* is the instance root. As example instance root is `/QIBM/UserData/WebCommerce/instances/instanceName`
- *inputFile* is the fully-qualified name of the input XML file that has the access policies

The syntax for the `LODWCSACD` command is:

```
LODWCSACD DATABASE(dbName) SCHEMA(schemaName)
PASSWD(instancePassword) INSTROOT('instanceRoot')
INFILE('inputFile')
```

You can store the XML files for your access control policies in the following directory:

`/QIBM/UserData/WebCommerce/instances/instanceName`

Additionally, within the XML files for your access control policies, you must use the full path to the access control DTD. The DTDs for access control policies are stored in the`/QIBM/ProdData/WebCommerce/xml/policies/dtd` directory.

As an example, if you deploy the access control policies for the tutorials to an Websphere Commerce instance running on an iSeries machine, you must modify the DTD specification in the XML files for the access control policies for the tutorials from:

```
<!DOCTYPE Policies SYSTEM "../dtd/accesscontrolpolicies.dtd">
```

to

```
<!DOCTYPE Policies SYSTEM "/QIBM/ProdData/WebCommerce/
xml/policies/dtd/accesscontrolpolicies.dtd">
```

For more information about the WebSphere Commerce access control model, refer to the *WebSphere Commerce Access Control Guide*.

Exporting the current enterprise application from WebSphere Application Server

In this step, you export the current enterprise application from WebSphere Application Server so that you can open it in the Application Assembly Tool.

To export the current enterprise application, do the following:

1. Create a directory into which the current enterprise application will be exported. Note that you do not call this a "temp" directory because you should not risk having the file deleted during routine system maintenance, until you are certain that you are satisfied with the way the customized code behaves once it has been deployed. To create this directory, do the following at a command prompt:
   a. Navigate to the following directory:
      ```
      drive:\WebSphere\CommerceServer\
      ```
   b. Enter the following commnad:
      ```
      mkdir working
      ```

      This creates the `drive:\WebSphere\CommerceServer\working` directory.
2. Open the WebSphere Application Server Administration Console.
3. Expand **WebSphere Administrative Domain**.
4. Expand **Enterprise Applications**.
5. Right-click your WebSphere Commerce application. For example, right-click the **demo** application and select **Export Application**.
6. In the **Export directory** field, enter `drive:\WebSphere\CommerceServer\working`.
   This exports the whole application, including all resources into the `WC_Enterprise_App_instanceName.ear` file (where `instanceName` is the name of your WebSphere Commerce instance).
7. Click **OK**. Exporting the application may take several minutes.

Exporting XML configuration information for the enterprise application

You must also export the XML configuration information for the enterprise application. To export this information, you use the `XMLConfig` command line utility provided by WebSphere Application Server.

To export this configuration information, do the following:

1. Copy the `was.export.app.xml` file from the following directory:
   ```
   drive:\WebSphere\CommerceServer\xml\config
   ```

   into the following directory:

```
drive:\WebSphere\CommerceServer\working
```

2. Open the `was.export.app.xml` file in a text editor. In this file replace all occurances of `$Enterprise_Application_Name$` with

   ```
   WebSphere Commerce Enterprise Application - instanceName
   ```

   where *instanceName* is the name of your WebSphere Commerce instance (for example, demo). Save this file.

   Note: The value that you are inserting must match the information for your instance that is displayed in the WebSphere Advanced Administration Console.
3. At a command prompt, navigate to the following directory:

   ```
   drive:\WebSphere\CommerceServer\working
   ```
4. Invoke the XMLConfig tool to perform a partial export by entering the following command:

   ```
   xmlConfig -export OutputFile.xml -partial was.export.app.xml
      -adminNodeName wasHostName
   ```

   where *wasHostName* is the name of the node in the WebSphere Application Server that contains your current enterprise application. Additionally, `OutputFile.xml` is the name of the file that is created as a result of running this command and `was.export.app.xml` is file that you modified in step 2.

After you have exported the information about the enterprise beans that are in the current enterprise application, you must add a new stanza to the XML file that describes the Bonus bean.

To add the new stanza that describes the Bonus bean, do the following:

1. Navigate to the following directory:

   ```
   drive:\WebSphere\CommerceServer\working
   ```
2. Open the `OutputFile.xml` file in a text editor.
3. Locate the `<ear-file-name>` tag and replace the value with the following:

   ```
   drive:\WebSphere\CommerceServer\working\
      WC_Enterprise_App_instanceName.ear
   ```
4. You must also add in a new stanza for the Bonus bean, as shown in the following sample:

   ```
   <ejb-module name="WCSSamplesEntityBeans">
      <jar-file>sampleEntityBeans.jar</jar-file>
      <module-install-info>
         <application-server-full-name>/NodeHome:$hostName$/EJBServerHome:
            WebSphere Commerce Server - demo/</application-server-full-name>
      </module-install-info>
      <ejb-module-binding>
         <data-source>
            <jndi-name>jdbc/WebSphere Commerce DB2 DataSource demo</jndi-name>
   ```

```
         <default-user>user</default-user>
         <default-password>password</default-password>
      </data-source>
      <enterprise-bean-binding name="Bonus_Binding">
         <jndi-name>democom/ibm/commerce/sample/objects/Bonus</jndi-name>
      </enterprise-bean-binding>
   </ejb-module-binding>
</ejb-module>
```

where

- *user* is your database user name.
- *password* is the password for your database user.

Notes:

a. The line breaks in the preceding example are for display purposes only.
b. Ensure that the **$hostName$** value matches the current admininstration node server name. In addition, ensure that there is no carriage return character in this line.
c. The `<application-server-full-name>` specification cannot span more than one line.
d. If you are using an Oracle database, you must modify the datasource information. Change the following line taken from the preceding code snip:

   ```
   <jndi-name>jdbc/WebSphere Commerce DB2 DataSource demo</jndi-name>
   ```

   to the following:

   ```
   <jndi-name>jdbc/WebSphere Commerce Oracle DataSource demo</jndi-name>
   ```
e. When you are deploying your own applications (for example, outside of this tutorial, ensure that the JNDI name for your enterprise beans that is specified in the XML file matches the JNDI name that is used in VisualAge for Java but has the WebSphere Commerce instance name prepended.

5. Save the `OutputFile.xml` file.

Assembling the new EJB group into the enterprise application

In this step, you open your enterprise application in the application assembler tool. Once it is open inside that tool, you can do the following to add the new Bonus bean to the enterprise application:

1. Import the new Bonus bean. The JAR file for the new EJB group is stored within the EJB Module section of the enterprise application.
2. Set the class path for the Bonus bean to include the implementation JAR file.
3. Add the implementation JAR file to the application. This JAR file is stored within the Files section of the enterprise application.

4. Set up WebSphere Application Server security for methods contained in the Bonus bean.

To assemble the new EJB group into the enterprise application, do the following:

1. Backup the current enterprise application, by doing the following:
   a. At a command prompt, navigate to the following directory:
      ```
      drive:\WebSphere\CommerceServer\working
      ```
   b. Enter the following command:
      ```
      copy WC_Enterprise_App_instanceName.ear
         WC_Enterprise_App_instanceName.ear.bak
      ```
2. Open the WebSphere Application Server administrative console.
3. From the **File** menu, select **Tools > Application Assembly Tool**.
4. If a Welcome window opens, select **Cancel** to close that window.
5. Open the enterprise application upon which you are going to work by doing the following:
   a. From the **File** menu, select **Open**.
   b. In the **File name** field, enter:
      ```
      drive:\WebSphere\CommerceServer\working\
         WC_Enterprise_App_instanceName.ear
      ```

      and click **Open**. Wait for the application to open before continuing to the next steps. This takes several minutes.
6. Right-click **EJB Modules** and select **Import**.
7. In the **File name** field, enter
   ```
   drive:\WebSphere\CommerceServer\temp\sampleEntityBeans.jar
   ```

   and click **Open**. In the Confirm Values window, click **OK**.
8. Once the `sampleEntityBeans.jar` file is imported, scroll to the **WCSSamplesEntityBeans** EJB group and select this group. Information about this group is shown in the pane on the right.
9. In the classpath field for the new enterprise bean, enter any dependent JAR files. In this case, enter
   ```
   lib/sampleImpl.jar lib\wcsejbimpl.jar
   ```
10. Click **Apply**.
11. Add the `sampleImpl.jar` file to the application, by doing the following:
    a. Right-click the **Files** node for the enterprise application and select **Add Files**. **Files** node for the enterprise application is located near the bottom of the hierarchical tree. Note that there are other Files nodes for components within the enterprise application, but you must select the Files node for the whole application.)

b. In the Add Files window, click **Browse**.
c. Navigate to the `drive:\WebSphere\CommerceServer\temp`.
d. With this directory highlighted, click **Select**.
e. Return to the Add Files window. Notice that the contents of the `drive:\WebSphere\CommerceServer\temp` directory are displayed. Highlight the `lib` directory.
The contents of the `lib` directory are displayed in the pane on the right.
f. In the pane on the right, select the `sampleImpl.jar` file and click **Add**. The file is then shown in the Selected Files pane.
g. Click **OK**.

12. Configure security for the Bonus bean, by doing the following:
    a. With the EJB Modules node expanded, locate and expand the **WCSSamplesEntityBeans** node.
    b. Expand **Entity Beans**.
    c. Expand **Bonus**
    d. Click **Method Extensions**, then in the pane on the right do the following:
        1) Click the **Advanced** tab.
        2) Ensure that **Security identity** is selected.
        3) For each method, ensure that **Use identity of EJB server** is selected.
        4) Click **Apply** (if you have made any modifications).
    e. In the left navigation pane, right-click **Security Roles** under the `WCSamplesEntityBeans` EJB group and select **New**, then do the following:
        1) In the **Name** field, enter `WCSecurityRole` and click **Apply**. Note, if this role exists already, you do not need to perform this step.
    f. In the left navigation pane, right-click **Method Permissions** under the `WCSamplesEntityBeans` EJB group and select **New**, then do the following:
        1) In the **Method Permission Name** field, enter `WCMethodPermission`
        2) In the **Methods** selection area, click **Add**.
        The Add Methods window opens.
        3) Expand **sampleEntityBeans.jar**, then **Bonus** and then expand each of the **Home** and **Remote** lists of methods.
        4) Hold the Shift key and select all of the home methods and click **OK**.
        5) Repeat the method selection process to add the remote methods as well (if there are any remote methods).

6) In the Roles selection area, click **Add**, select the `WCSecurityRole` and click **OK**.
7) Click **Apply** for each update.

13. From the **File** menu, select **Save**.
14. Close the Application Assembler Tool.

After this step has completed, you have created a new enterprise application that contains all of the previous logic as well as your new business logic. This is all contained in the newly modified `WC_Enterprise_App_`*`instanceName`*`.ear` file.

Importing the new enterprise application into WebSphere Application Server

The following are the high-level steps involved in importing the new enterprise application into WebSphere Application Server:

1. Stopping the enterprise application that is currently running in WebSphere Application Server and then removing it. These steps are performed in the WebSphere Application Server Administator's Console.
2. Importing the new application, using the XMLConfig command line utility.
3. Refreshing the WebSphere Application Server Administator's Console and then starting the new enterprise application.

Each of these steps is described in more detail in the following sections.

Stopping and removing the current enterprise application

To stop and remove your current enterprise application from WebSphere Application Server, do the following:

1. Open the WebSphere Application Server Administration Console.
2. Expand **WebSphere Administrative Domain**.
3. Expand **Nodes**.
4. Expand *nodeName* (where *nodeName* is the name of your node).
5. Expand **Application Servers**.
6. Right-click your WebSphere Commerce application. For example, right-click the **WebSphere Commerce Server -** *instanceName* and select **Stop**.
7. Expand **Enterprise Applications**.
8. Right-click your WebSphere Commerce application. For example, right-click the **WebSphere Commerce Enterprise Application - demo** application and select **Stop**.
9. Right-click your WebSphere Commerce application. For example, right-click the **WebSphere Commerce Enterprise Application - demo** application and select **Remove**.

10. When prompted to indicate if the application should be exported, select **No**.

Importing the new enterprise application using XMLConfig

To import the new enterprise application using the XMLConfig command line utility, do the following:

1. Navigate to the following directory:

   ```
   drive:\WebSphere\CommerceServer\working
   ```
2. At the command prompt, enter the following command to import the enterprise application into WebSphere Application Server:

   ```
   xmlConfig -import OutputFile.xml -adminNodeName was_hostname
   ```

   where *was_hostname* is the name of the node of the WebSphere Application Server containing the current application.

Note: 400 If you were deploying to a WebSphere Commerce instance running on iSeries, you would have to perform an additional step to modify directory permissions after you have imported the application. Refer to "Importing an enterprise application" on page 366 for details on how to modify these permissions.

Starting the new enterprise application

After you have imported the new enterprise application using the XMLConfig command line utility, you can use the WebSphere Application Server Administrator's Console to perform a refresh and then start the new application.

To refresh the console and start the new application, do the following:

1. Open the WebSphere Application Server Administration Console.
2. Expand **WebSphere Administrative Domain**
3. Highlight **Nodes**.
4. Click the **Refresh selected subtree** icon.
5. Start your WebSphere Commerce application by doing the following:
   - Expand **Application Servers**.
   - Right-click your WebSphere Commerce application. For example, right-click the **WebSphere Commerce Server -** *instanceName* and select **Start**.

Test MyNewControllerCmd

The next step is to test the new logic in your store running in the WebSphere Application Server environment. To test MyNewControllerCmd, do the following:

1. Test the integration by entering the following URL in a browser:

```
http://hostname/webapp/wcs/stores/servlet/StoreCatalogDisplay?
     storeId=store_Id&catalogId=catalog_Id&langId=-1
```

where `store_Id` is the identifier for your store and `catalog_Id` is the identifier for your store's catalog.

2. Click the **Register** link.
   The Register or Login page is displayed.
3. In the **E-mail address** field, enter `wcsadmin`.
4. In the **Password** field, enter the password for the wcsadmin ID used by this site and then click **Login**.
5. After the user is logged on, click the **MyNewControllerCmd** link in the side navigation pane. The Sample JSP is displayed.

If the updated `sidebar.jsp` file does not appear, do the following to clear your cache:

1. Delete the cached files from the following directory:
   `drive:\WebSphere\CommerceServer\instances\instanceName\cache`
2. Delete any relevent cached files from the following directories:

```
drive:\WebSphere\AppServer\temp\hostName\
WebSphere_Commerce_Server_instanceName\
WebSphere_Commerce_Enterprise_Application_-_instanceName\
wcstores.war\storeName
```

```
drive:\WebSphere\AppServer\temp\hostName\
WebSphere_Commerce_Server_instanceName\
WebSphere_Commerce_Enterprise_Application_-_instanceName\
wcstores.war
```

3. Clear your browser's cache.

If the JSP page compilation takes too long, the page may not be displayed. In this case, reload the page.

Chapter 10. Modifying and extending existing business logic

The following tutorials show how to extend or modify existing WebSphere Commerce business logic.

Extending an existing controller command

In this section, you extend the existing OrderProcess controller command so that the total bonus points that have been accumulated on the purchase are displayed on the order confirmation page.

Note: The goal of this tutorial is to show the process of modifying an existing controller command. It is not designed to show the best way to modify the order process step in the shopping flow. In fact, WebSphere Commerce provides the `ExtOrderProcess` task command that can be used to modify the order process step in the shopping flow.

Before starting this tutorial

You should have already completed the steps in Chapter 9, "Tutorial: Creating new business logic" on page 201.

The following list summarizes the steps involved in extending the OrderProcess command:

1. Creating a new package in which the customized code is stored. Recall that all customized code (for commands and data beans) must be stored in projects and packages that are separate from the WebSphere Commerce code.
2. Creating a new `OrderProcessCmdBonusImpl` class that extends the existing `OrderProcessCmdImpl` command.
3. Adding fields and methods to the `OrderProcessCmdBonusImpl` class.
4. Modifying the command registry to use the `OrderProcessCmdBonusImpl` class
5. Modifying the confirmation.jsp template to display the new business logic.
6. Testing the new business logic within the WebSphere Test Environment.
7. (Optional) Deploying the new business logic to a store on a remote WebSphere Commerce Server.

Creating the new package for `OrderProcessCmdBonusImpl`

To create the new package in which the `OrderProcessCmdBonusImpl` command is stored, do the following:

1. In the VisualAge for Java Workbench window, ensure that you have the Projects tab selected.
2. Right-click the **_WCSamples** project and select **Add > Package**.
   The Add Package SmartGuide opens.
3. Ensure that the **Create a new package named** radio button is enabled and enter `com.ibm.commerce.sample.order`.
4. Click **Finish**.

Creating the `OrderProcessCmdBonusImpl` class

To create the new `OrderProcessCmdBonusImpl` class, do the following:

1. Right-click the `com.ibm.commerce.sample.order` package and select **Add > Class**.
   The Create Class SmartGuide opens.
2. Ensure that the **Create a new class** radio button is selected.
3. In the **Class name** field, enter `OrderProcessCmdBonusImpl`.
4. To specify the superclass, click **Browse**, then in the **Pattern** field, enter `com.ibm.commerce.order.commands.OrderProcessCmdImpl` and click **OK**.
5. Click **Next**.
6. To specify the packages that should be imported, click **Add Package**. In the **Pattern** field, enter the following packages:
   - `com.ibm.commerce.datatype` and click **Add**
   - `com.ibm.commerce.exception` and click **Add**
   - `com.ibm.commerce.order.commands` and click **Add**
   - `com.ibm.commerce.order.objects` and click **Add**
   - `com.ibm.commerce.ras` and click **Add**
   - `com.ibm.commerce.server` and click **Add**
   - `com.ibm.commerce.sample.objects` and click **Add**
   - `javax.ejb` and click **Add**
   - `java.io` and click **Add**
   - `java.math` and click **Add**, then **Close**.
7. To specify the interfaces that the class should implement, click **Add**. In the Pattern field, enter the following interface:
   - `OrderProcessCmd` and click **Add**, then **Close**.
8. Click **Finish**.

Adding fields and methods to `OrderProcessCmdBonusImpl`

You must add two fields and a performExecute method to the class.

To add the `theOrder` field to the `OrderProcessCmdBonusImpl` class, do the following:

1. Right-click the `OrderProcessCmdBonusImpl` class and select **Add > Field**. The Create Field SmartGuide opens.
2. Add a field to the class using the following attributes. For detailed steps on how to create a new field, refer to "Creating new fields" on page 213

Attribute Name	Value
Field Name	`theOrder`
Field Type	`OrderAccessBean`
Initial Value	Leave blank.
Access Modifiers	`private`
Other Modifiers	Leave all unchecked.
Access with getter and setter methods	`checked`
Getter	`public`
Setter	`public`

and click **Finish**.

To add the `bonusPercentAmount` field to the `OrderProcessCmdBonusImpl` class, do the following:

1. Right-click the `OrderProcessCmdBonusImpl` class and select **Add > Field** again.
2. Add a field to the class using the following attributes. For detailed steps on how to create a new field, refer to "Creating new fields" on page 213

Attribute Name	Value
Field Name	`bonusPercentAmount`
Field Type	`double`
Initial Value	`1000`
Access Modifiers	`private`
Other Modifiers	Leave all unchecked.
Access with getter and setter methods	`checked`
Getter	`public`
Setter	`public`

and click **Finish**.

To add the `performExecute` method to the `OrderProcessCmdBonusImpl` class, do the following:

1. Right-click the **`OrderProcessCmdBonusImpl`** class and select **Add > Method**. The Create Method SmartGuide opens.
2. Ensure that the **Create a new method** radio button is selected and click **Next**.
3. In the **Method Name** field, enter `performExecute`.
4. From the **Return Type** drop-down list, select `void`. Click **Next**.
5. To specify the exception that the method may throw, click **Add**. In the **Pattern** field, enter `ECException`, click **Add**, and then **Close**.
6. Click **Finish**.
   The method is generated and source code for the method is displayed.
7. You must modify the source code. Locate the following line in the source code of the performExecute method:
   ```
   public void performExecute() throws
      com.ibm.commerce.exception.ECException {
   ```
   After the preceding line, enter the following code:

You can cut and paste this code from the PDF version of the Programmer's Guide. It is recommended that you initially copy the code in the Scrapbook window in VisualAge for Java (refer to the VisualAge for Java online help for more information) and inspect the code to ensure no characters were lost or modified during the cut and paste operation. Then, after validating the code, copy it into the target location. Note that copying the text into another editor may cause some characters to be modified.

```
final String methodName = "performExecute";
ECTrace.entry(ECTraceIdentifiers.COMPONENT_ORDER,
   this.getClass().toString(), methodName);

// do all order processing as normal
super.performExecute();

// *** updating Bonus Point Information ***

// fetch order info
theOrder = new OrderAccessBean();
theOrder.setInitKey_orderId(getOrderRn().toString());

int bonusPt;  // bonus points for this order
int bonusTotal;  // total bonus points
BigDecimal subtotal;  // subtotal
BigDecimal bonusdeter;  // bonus determinant
BigDecimal ans;

// determine bonus points = subtotal * bonus determinant
try {
     subtotal = theOrder.getTotalProductPriceInEJBType();
     bonusdeter = new BigDecimal(bonusPercentAmount);
```

```
      ans = subtotal.multiply(bonusdeter);
      bonusPt = Math.round(ans.floatValue());

      System.out.println("subtotal is: " + subtotal +
         " bonus deter is: " + bonusdeter + " ans is: " + ans);
      System.out.println("Bonus Percent amount = " +
         bonusPercentAmount);
      System.out.println("Bonus calculated is: "+ bonusPt);
     }


   // Various Exceptions
   catch (Exception ex) {
   throw new ECSystemException(ECMessage._ERR_GENERIC,
      this.getClass().toString(),methodName,
      ECMessageHelper.generateMsgParms(ex.getMessage()), ex);
   }

// *** Updating bonus points in BONUS table using bean
//     created in previous example ***

BonusAccessBean bonusBean = new BonusAccessBean();
bonusBean.setInit_argMemberId(
   getCommandContext().getUserId().toString());
try {
    //new bonus value = this order bonus points + Old Bonus Points
    bonusTotal = bonusPt + Integer.parseInt(bonusBean.getBonusPoint());
    bonusBean.setBonusPoint(String.valueOf(bonusTotal));
    bonusBean.commitCopyHelper();

    System.out.println("In try, BonusTotal calculated is: "+
           bonusTotal);

    }

// Various exceptions
catch (FinderException e) // user does not have points setup yet
{
   // create a row in table bonus
   bonusTotal = bonusPt;

   try {
       BonusAccessBean bonusBeanNew = new
          BonusAccessBean(getCommandContext().getUserId(),
             new Integer(bonusTotal));

          System.out.println("In catch, BonusTotal calculated is: "+
             bonusTotal);

}
catch (Exception ex) {
   throw new ECSystemException(ECMessage._ERR_GENERIC,
      this.getClass().toString(), methodName,
      ECMessageHelper.generateMsgParms(ex.getMessage()), ex);
   }
```

```
        }
           catch (Exception ex) {
               throw new ECSystemException(ECMessage._ERR_GENERIC,
                 this.getClass().toString(), methodName,
                 ECMessageHelper.generateMsgParms(ex.getMessage()), ex);
           }

           // *** setting view details ***

           // Fetch setResponse properties and add bonus parameters
          //  needed by the JSP page
           TypedProperty resp = getResponseProperties();
           resp.put("bonus", new Integer(bonusPt).toString());
           setResponseProperties(resp);
           ECTrace.exit(ECTraceIdentifiers.COMPONENT_ORDER,
              this.getClass().toString(), methodName);
```

8. Save your work.

Modifying the command registry to use `OrderProcessCmdBonusImpl`

In this example, you want to use the new implementation class for order processing whenever order processing is required. In order for this to happen, you must update the command registry to associate the original OrderProcess interface with the new `OrderProcessCmdBonusImpl` implementation class.

DB2 If you are using a DB2 database, do the following to update the command registry:

1. Open the DB2 Command Center (**Start > Programs >IBM DB2 > Command Center**).
2. With the Script tab selected, create the required entry in the CMDREG table, by entering the following information in the script window:

   ```
   connect to your_database_name;
   update CMDREG
   set CLASSNAME='com.ibm.commerce.sample.order.OrderProcessCmdBonusImpl'
   WHERE INTERFACENAME='com.ibm.commerce.order.commands.OrderProcessCmd'
   and storeent_Id=0;
   ```

   where *`your_database_name`* is the name of your database and click the Execute icon
   This command is used by all merchants (indicated by the 0 value for STOREENT_ID).

Oracle If you are using an Oracle database, do the following to update the command registry:

1. Open the Oracle SQL Plus command window (**Start > Programs > Oracle > Application Development > SQL Plus**).
2. In the **User Name** field, enter your Oracle user name.
3. In the **Password** field, enter your Oracle password.

4. In the **Host String** field, enter your connect string.
5. Create the required entry in the URLREG table, by entering the following information in the SQL Plus window:

   ```
   update CMDREG
   set CLASSNAME='com.ibm.commerce.sample.order.OrderProcessCmdBonusImpl'
   WHERE INTERFACENAME='com.ibm.commerce.order.commands.OrderProcessCmd'
   and storeent_Id=0;
   ```

   Press Enter to run the SQL statement.
   This command is used by all merchants (indicated by the 0 value for STOREENT_ID)
6. Enter the following to commit your database changes:

   ```
   commit;
   ```

   and press Enter to run the SQL statement.

Modifying the `confirmation.jsp` template

You must modify the `confirmation.jsp` template to display the new business logic that you have added to the order process business process. To modify the display template, do the following:

1. Navigate to the following directory:
   `vaj_drive:\VAJava\ide\project_resources\IBM WebSphere Test Environment\hosts\default_host\default_app\web\`*`store_directory`*.
2. Make a copy of the `confirmation.jsp` and call it `confirmation.jsp.bak`
3. Open the `confirmation.jsp` in a text editor.
4. After the existing import statements, add the following:

   ```
   <%@ page import="com.ibm.commerce.datatype.*"   %>
   ```
5. Immediately after the following line in the JSP template:

   ```
   String orderRn = jhelper.getParameter("orderId");
   ```

   add the following:

   ```
   String bonus = ((TypedProperty)request.getAttribute(
      ECConstants.EC_REQUESTPROPERTIES)).getString("bonus");
   ```
6. Locate the following section in the JSP template:

   ```
   <tr>
   <td align="left" valign="middle">
   <font class="product"><%=infashiontext.getString("GRAND_TOTAL")%>
      </font></td>
   <td align="right" valign="middle">
   <font class="strongprice"><%=orderBean.getGrandTotal()  %></font></td>
   ```

   and then add the following:

   ```
   </tr>
   <tr>
   <td align="left" valign="middle">
   ```

```
<font class="text">Bonus Points</font></td>
<td align="right" valign="middle">
<font class="strongtext"><%=bonus  %></font></td>
```

7. Save your work.

Testing `OrderProcessCmdBonusImpl` within the WebSphere Test Environment

You can now use the WebSphere Test Environment to test your new business logic. To test the `OrderProcessCmdBonusImpl` command, do the following:

1. Start the WebSphere Test Environment, as described in Appendix A, "Starting and stopping the WebSphere Test Environment" on page 333.
2. Open a browser and enter the URL for your store. For example, enter the following URL:

   ```
   http://localhost:8080/webapp/wcs/stores/servlet/StoreCatalogDisplay?
        storeId=10001&catalogId=10001&langId=-1
   ```
3. Select and purchase a product.
4. After you have purchased a product, the order confirmation displays the number of bonus points that were earned on the order.

(Optional) Deploying the customized business logic to a remote WebSphere Commerce Server

After you have completed testing your business logic in the WebSphere Test Environment and are satisfied with your code, you can deploy the code to a store on a remote WebSphere Commerce Server. In this tutorial, the customizations include the following:

- The new OrderProcessCmdBonusImpl class.
- The updated confirmation.jsp template file.
- The updated command registry.

As such, code deployment includes the following steps:

1. Creating a JAR file for the command logic, using the tools in VisualAge for Java.
2. Copying the JAR file and JSP template to the appropriate directories on the target WebSphere Commerce Server.
3. Updating the command registry on the target WebSphere Commerce Server.

Notes about the test payment method

The sample store running within the WebSphere Test Environment by default uses a test payment method. This test payment method is used so that you can complete the shopping flow within WebSphere Test Environment, without requiring a call out to a Payment Manager. This test payment method only lets you complete a purchase, it does not enable orders submitted with this payment method to be available for further processing. As such, the test payment method should only be used within the WebSphere Test Environment.

Ensure that you can complete a purchase in the store to which you are deploying this customized code. Payment processing can be done using either a local or remote Payment Manager.

For more information about the test payment method, refer to “Test payment method” on page 196.

Creating the JAR file for the command logic

You must package the command logic into a JAR file in order for it to be deployed to the target WebSphere Commerce Server. Since the `OrderProcessCmdBonusImpl` is stored in the _WCSamples project, you create a JAR file for that project.

To create the JAR file, do the following:

1. Stop the WebSphere Test Environment, as described in Appendix A, “Starting and stopping the WebSphere Test Environment” on page 333.
2. Right-click the **_WCSamples** project and select **Export**.
   The Export SmartGuide opens.
3. Select **Jar file** and click **Next.**
4. In the Jar file field, enter the following:
   `*drive*:\WebSphere\CommerceServerDev\mytemp_b\wcssamplesb_1.jar`
   where *drive* is the drive on which Commerce Studio is installed.
5. Select attributes as follows:

Attribute	Value
class	Checked
java	Unchecked
resource	Checked
beans	Checked
Include debug attributes in .class file	Checked

   Accept the default values for other attributes.
6. Click **Finish**.

Since the JAR file created does not contain complete package naming information, you must use another packaging utility (outside of VisualAge for Java) to repackage the JAR file. To repackage this file, do the following:

1. In a command window, navigate to the following directory: `drive:\WebSphere\CommerceServerDev\mytemp_b`
2. Enter `mkdir temp1`.
3. Enter `cd temp1`.
4. Set the path as follows: `set PATH=%PATH%;drive:\WebSphere\WebSphereStudio4\bin;` where *drive* is the drive on which WebSphere Studio is installed.
5. Enter `jar xvf ../wcssamplesb_1.jar`
6. Enter `jar cvf ../wcssamplesb.jar *`

Storing assets on the target WebSphere Commerce Server

The JAR file for the command logic and the modified `confirmation.jsp` template must be placed in the appropriate directories on the target WebSphere Commerce Server.

To store the JAR file in the appropriate directory on a remote WebSphere Commerce Server, do the following:

1. On the development machine, open a command window and navigate to the following directory: `drive:\WebSphere\CommerceServerDev\mytemp_b` and locate the `wcssamplesb.jar` file.
2. Copy this file into the following directory on the target WebSphere Commerce Server:

```
drive:\WebSphere\AppServer\installedApps\
   WC_Enterprise_App_instanceName.ear\wcstores.war\WEB-INF\lib
```

To store the `confirmation.jsp` template in the appropriate directory on the target WebSphere Commerce Server, do the following:

1. On the development machine, navigate to the following directory: `vaj_drive:\VAJava\ide\project_resources\IBM WebSphere Test Environment\hosts\default_host\default_app\web\store_directory`
2. Copy the `confirmation.jsp` template to the following directory, on the target WebSphere Commerce Server:

```
drive:\WebSphere\AppServer\installedApps\
   WC_Enterprise_App_instanceName.ear\wcstores.war\storeDir
```

Updating the command registry

If you are deploying the OrderProcessCmdBonusImpl command to a target WebSphere Commerce Server that uses a different database than the WebSphere Test Environment, you must update the target database to reflect the changes that you made to the command registry.

▶ DB2 If you are using a DB2 database, do the following to update the database of the target WebSphere Commerce Server:

1. Open the DB2 Command Center (**Start > Programs >IBM DB2 > Command Center**).
2. With the Script tab selected, create the required entry in the CMDREG table, by entering the following information in the script window:
   ```
   connect to your_target_database_name;
   update CMDREG
   set CLASSNAME='com.ibm.commerce.sample.order.OrderProcessCmdBonusImpl'
   WHERE INTERFACENAME='com.ibm.commerce.order.commands.OrderProcessCmd'
   and storeent_Id=0
   ```
   where *your_target_database_name* is the name of your database and click the Execute icon

Oracle If you are using an Oracle database, do the following to update the command registry:

1. Open the Oracle SQL Plus command window (**Start > Programs > Oracle > Application Development > SQL Plus**).
2. In the **User Name** field, enter your Oracle user name.
3. In the **Password** field, enter your Oracle password.
4. In the **Host String** field, enter your connect string.
5. Create the required entry in the CMDREG table, by entering the following information in the SQL Plus window:
   ```
   update CMDREG
   set CLASSNAME='com.ibm.commerce.sample.order.OrderProcessCmdBonusImpl'
   WHERE INTERFACENAME='com.ibm.commerce.order.commands.OrderProcessCmd'
   and storeent_Id=0;
   ```
   Click Enter to run the SQL statement.
   This command is used by all merchants (indicated by the 0 value for STOREENT_ID)
6. Enter the following to commit your database changes:
   ```
   commit;
   ```
   and press Enter to run the SQL statement.

Restarting your enterprise application in WebSphere Application Server

After you have added the command logic to your enterprise application by placing the file assets into the appropriate directories and updating the command registry, you must stop and restart your enterprise application in order for the change to take effect.

To stop and restart your enterprise application, do the following:

1. Open the WebSphere Application Server Administration Console.
2. Expand **WebSphere Administrative Domain**.

3. Expand **Nodes**.
4. Expand ***nodeName*** (where ***nodeName*** is the name of your node).
5. Expand **Application Servers**.
6. Right-click your WebSphere Commerce application. For example, right-click the **WebSphere Commerce Server - demo** application and select **Stop**.
7. Right-click your WebSphere Commerce application. For example, right-click the **WebSphere Commerce Server - demo** application and select **Start**.

Testing your new logic in InFashion running in the WebSphere Application Server

You can now verify your new business logic in the InFashion store running in the WebSphere Application Server.

To perform this final verification, do the following:

1. After the WebSphere Commerce Server instance has started, open a browser and enter the URL for your store's home page. For example, enter the following URL:

   ```
   http://hostname/webapp/wcs/stores/servlet/StoreCatalogDisplay?
        storeId=store_Id&catalogId=catalog_Id&langId=-1
   ```

   where `store_Id` is the identifier for your store and `catalog_Id` is the identifier for your store's catalog.
2. Select and purchase a product.
3. After you have purchased a product, the order confirmation displays the number of bonus points that were earned on the order.

Modifying an existing entity bean and extending an existing task command

Note: The goal of this tutorial is to show the process used to modify existing entity beans and extend existing task commands. It is not designed to show the best way to modify product pricing. For information about discounting prices, refer to the *WebSphere Commerce Calculation Framework Guide*.

In Chapter 9, "Tutorial: Creating new business logic", an entirely new set of business logic was created. This included the creation of a new controller command, a new JSP template, a new database table, a new enterprise bean for accessing the table, a corresponding access bean, as well as a data access bean. All of this logic fit together to create a simplified bonus points application in which a user's balance of bonus points can be updated using a JSP template that is launched by the new controller command.

The manner in which the BONUS table is used in Chapter 9, "Tutorial: Creating new business logic" is depicted in the following diagram:

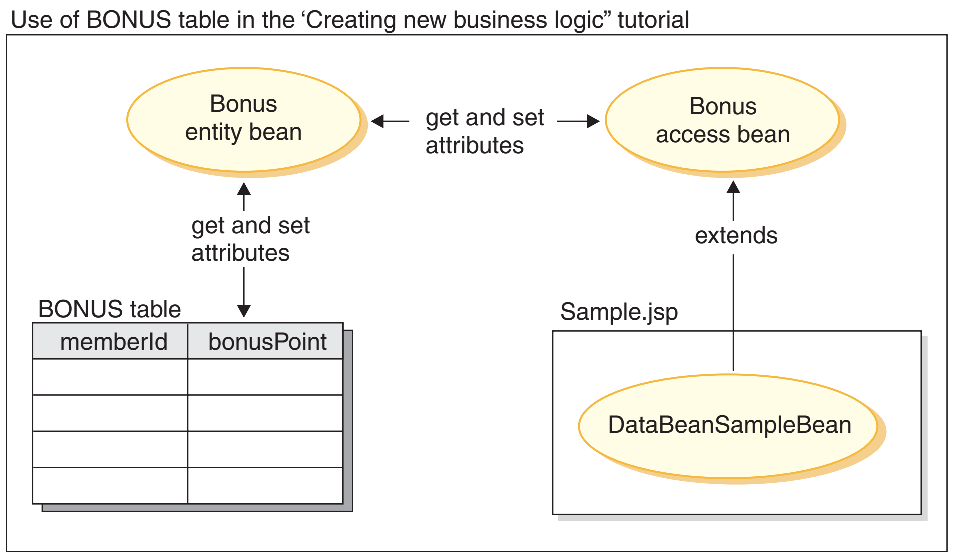


Figure 41.

In the next tutorial, the BONUS table is used in a different manner. Specifically, the User entity bean is customized in a manner such that it appears to applications that the BONUSPOINT column of the BONUS table is actually a column in the USERS table. When a new record is created in the USERS table, a corresponding record is automatically inserted into the BONUS table.

In order to create this *table join*, a new CMP field must be added to the User entity bean. This CMP field maps to the BONUSPOINT column in the BONUS table using the Secondary Table Map feature in the VisualAge for Java Map Browser.

To integrate the modified User entity bean into the shopping flow, a new price for products is created. This new price takes the shopper's current balance of bonus points into consideration. You create the new price by extending the GetProductContractUnitPriceCmd task command. When extending this command, you create a new interface that extends the GetProductContractUnitPriceCmd interface. The new interface adds an additional attribute (for the bonus price). You also create a new implementation class that extends the GetContractUnitPriceCmdImpl implementation class. This new implementation class is called

"GetNewContractUnitPriceCmdImpl". The new implementation class calls the performExecute method of its super class and then adds the business logic to determine the new bonus price.

The following diagram shows how the BONUS table is used in the next tutorial.

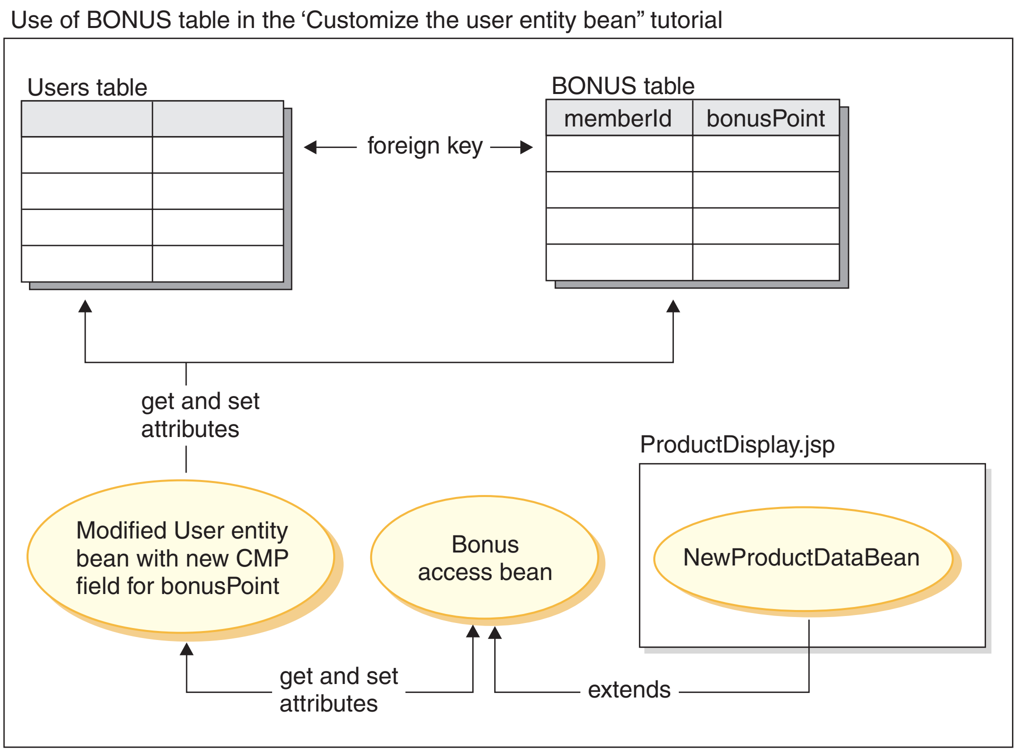


Figure 42.

This tutorial includes the following steps:

1. Add a new CMP field to the User entity bean.
2. Create and populate the BONUS table.
3. Update the WCSUser database schema and table mapping to include the new BONUS table.
4. Create the foreign key relationship between the BONUS and USER tables.
5. Create the BONUS table map.
6. Generate the deployed code and access bean for the User entity bean.
7. Use the test client for preliminary testing of the updated entity bean.
8. Create a new task command interface and implementation class. The new task command extends `GetBaseUnitProductPriceCmd`. The most important

new feature in this command is the logic in the performExecute() method of the implementation class. This method calculates a new *bonus price* for the product. This bonus price is created such that the price is reduced by the shopper's bonus point balance, up to a maximum reduction of 20% of the calculated price. The following table shows examples of this price reduction formula in use:

Calculated price	**Bonus point balance**	**Maximum deduction (20% of calculated price)**	**New bonus price**
$1000	100	$200	$900 Since the bonus point balance is less than the maximum deduction, the list price is reduced by the bonus points.
$1000	300	$200	$800 Since the bonus point balance exceeds the maximum deduction, the list price is reduced by the maximum deduction of $200.

9. Create the `NewProductDataBean` that extends `ProductDataBean`. Add a new method to this bean so that the new bonus price can easily be used in a product display page.
10. Update the product display JSP template for the InFashion store to show the bonus price.
11. Test the business logic in the InFashion store running within the WebSphere Test Environment.
12. (Optional) Deploy the updated business logic to a remote WebSphere Commerce Server and test it outside of the WebSphere Test Environment.

Before starting this tutorial

If you have not completed Chapter 9, "Tutorial: Creating new business logic" on page 201, you must perform the steps described in "Preparing the sample project" on page 202 before starting this tutorial.

Adding a new bonusPoint field to the User entity bean

In this section, you use the VisualAge for Java's EJB tools to add the new CMP field to the entity bean. The new field is called *bonusPoint* and is eventually be mapped to the BONUSPOINT column of the BONUS table.

To add the new CMP field to the User entity bean, do the following:

1. If they are running, stop the servlet engine, the EJB server, and the persistent name server, as described in Appendix A, "Starting and stopping the WebSphere Test Environment" on page 333.
2. In the Workbench, click the **EJB** tab.
3. Expand the **WCSUser** EJB group.
4. Right-click the **User** bean and select **Add > CMP Field**.
   The Create CMP Field SmartGuide opens.
5. Create a new CMP field with the following properties:

Property	**Value**
Field Name	`bonusPoint`
Field Type	`int`
Initial Value	`0`
Access with getter and setter methods	`enable`
Promote getter and setter methods to remote interface	`enable`
Getter	`public`
Setter	`public`

   and click **Finish**.

The bonusPoint field is displayed in the Properties pane. There may be some warnings displayed, but these are fixed when you regenerate the entity bean's code.

Creating and populating the BONUS table

A database table that records a user's bonus points is used in this tutorial.

If you completed Chapter 9, "Tutorial: Creating new business logic" on page 201, you are already familiar with this table. In this case, you must update the table to add a row for each user in the USERS table. If you did not complete Chapter 9, "Tutorial: Creating new business logic" on page 201, you must create and populate the table. Instructions for each of these scenarios is provided.

▶ DB2 If you are using a DB2 database and need to *update* the BONUS table, do the following:

1. Open the DB2 Command Center (**Start > Programs > IBM DB2 > Command Center**) and click the **Script** tab.
2. In the **Command** field, enter

   ```
   connect to your_database_name
   ```

   where *your_database_name* is the name of your database and click the Execute icon.
3. In the **Command** field, enter the following, then click the Execute icon:

   ```
   INSERT INTO BONUS
   (SELECT USERS_ID, 0
   FROM USERS
   WHERE USERS_ID NOT IN (SELECT MEMBERID FROM BONUS))
   ```

   The BONUS table has now been updated.

▶ DB2 If you are using a DB2 database and need to *create and populate* the BONUS table, do the following:

1. Open the DB2 Command Center (**Start > Programs > IBM DB2 > Command Center**) and click the **Script** tab.
2. In the **Command** field, enter

   ```
   connect to your_database_name
   ```

   where *your_database_name* is the name of your database and click the Execute icon.
3. In the **Command** field, enter the following, then click the Execute icon:

   ```
   CREATE TABLE BONUS (MEMBERID BIGINT NOT NULL,
       BONUSPOINT INTEGER NOT NULL, constraint p_memberid primary key (MEMBERID),
       constraint f_memberid foreign key (MEMBERID)
       references users (users_id) on delete cascade)
   ```

   The BONUS table has now been created.
4. To populate the table, enter the following in the **Command** field, then click the Execute icon:

   ```
   insert into BONUS (select USERS_ID, 0 from USERS)
   ```
5. Close the DB2 Command Center.

▶ Oracle If you are using an Oracle database, and need to *update* the BONUS table, do the following:

1. Open the Oracle SQL Plus command window (**Start > Programs > Oracle > Application Development > SQL Plus**).

2. In the **User Name** field, enter your Oracle user name.
3. In the **Password** field, enter your Oracle password.
4. In the **Host String** field, enter your connect string.
5. Update the BONUS table, by entering the following information in the SQL Plus window:
   ```
   INSERT INTO BONUS
   (SELECT USERS_ID, 0
   FROM USERS
   WHERE USERS_ID NOT IN (SELECT MEMBERID FROM BONUS));
   ```
   Click Enter to run the SQL statement.
   The BONUS table has now been updated.
6. Enter the following to commit your database changes:
   ```
   commit;
   ```
   and press Enter to run the SQL statement.

Oracle If you are using an Oracle database, and need to *create and populate* the BONUS table, do the following:

1. Open the Oracle SQL Plus command window (**Start** > **Programs** > **Oracle** > **Application Development** > **SQL Plus**).
2. In the **User Name** field, enter your Oracle user name.
3. In the **Password** field, enter your Oracle password.
4. In the **Host String** field, enter your connect string.
5. Create the BONUS table, by entering the following information in the SQL Plus window:
   ```
   CREATE TABLE Bonus (MEMBERID NUMBER NOT NULL,
       BONUSPOINT INTEGER NOT NULL, constraint p_memberid primary key (MEMBERID),
       constraint f_memberid foreign key (MEMBERID)
       references users (users_id) on delete cascade);
   ```
   Click Enter to run the SQL statement.
   The BONUS table has now been created.
6. Populate the BONUS table, by entering the following:
   ```
   insert into BONUS (select USERS_ID, 0 from USERS);
   ```
7. Enter the following to commit your database changes:
   ```
   commit;
   ```
   and press Enter to run the SQL statement.

Updating the schema and table mapping

In the following sections, you update the WCSUser schema with the new BONUS table, create the foreign key relationship for the new table, and create a table map between the fields of the User entity bean and the columns of the BONUS table.

Creating the BONUS table schema

To create the table schema, do the following:

1. Right-click the **User** entity bean and select **Open To > Database Schemas**.
   The Schema Browser window opens.
2. From the **Schemas** list, select **WCS User**.
3. From the **Tables** menu, select **New Table**.
   The Table Editor opens.
4. In the **Name** field, enter `BONUS`.
5. Leave the **Qualifier** and **Physical name** fields blank.
6. In the Table Columns section, click **New**. The Column Editor opens. Create a column, as follows:

Attribute	**Value**
Name	`memberId`
Physical name	Leave this field blank.
Type	`BIGINT`
Type Details	`VapConverter`
Allow nulls	unchecked

   and click **OK**.
7. Select **memberId** in the Table columns list and click **>>** to use this column as the primary key.
8. Create another column (click **New** again) as follows:

Attribute	**Value**
Name	`bonusPoint`
Physical name	Leave this field blank.
Type	`INTEGER`
Type Details	`VapConverter`
Allow nulls	checked

   and click **OK**.

9. Click **OK** in the Table Editor.
   After a few moments, **BONUS** is listed in the **Tables** list of the Schema Browser window.
10. For verification, click **BONUS** in the **Tables** list and ensure that its two columns are displayed in the **Columns** list.

Creating the foreign key relationship

In this section, you establish the foreign key relationship between the BONUS and USERS tables.

To create the foreign key relationship, do the following:

1. In the Schema Browser window, click the **WCS User** schema.
   The tables and foreign key relationships for this schema are displayed.
2. From the **Foreign Keys** menu, select **New Foreign Key Relationship**.
   The Foreign Key Relationship Editor opens.
3. In the **Name** field, `F_User_Bonus`.
4. Ensure that the **Constraint exists in Database** checkbox is checked.
5. From the **Primary key table** drop-down list, select `USERS`
6. From the **Foreign key table** drop-down list, select `BONUS`
7. Click in the empty field under the **Foreign Key** heading, so that a drop-down list is displayed. From this list, select **memberId** and click **OK**.
   `F_User_Bonus` is displayed in the list of foreign key relationships.
8. From the **Schemas** menu, select **Save Schema**, and then click **Finish**.
9. Close the Schema Browser.

Creating the BONUS table map

In this section, you create the mapping between the `BONUSPOINT` column in the BONUS table and the `bonusPoint` field in the User entity bean.

To create the BONUS table map, do the following:

1. Ensure that you have the Workbench open with the EJB tab selected.
2. From the **EJB** menu, select **Open To > Schema Maps**.
   The Map Browser opens.
3. From the Datastore Maps list, select **WCS User**.
4. In the **Persistent Classes** list, do the following:
   a. Double-click **Member**.
   b. Select **User** (Note that **User** is only displayed below **Member** after you expand Member in step 4a.)
5. From the **Table Maps** menu, select **New Table Map > Add Secondary Table Map**.
   The Secondary Table Map window opens.
6. From the **Table** drop-down list, select **BONUS**.

7. From the Foreign key relationship drop-down list, select **F_User_Bonus** and click **OK**.
   After a few moments, the `BONUS (secondary)` map is displayed in the Table Maps list.
8. Highlight, then right-click the **BONUS (secondary)** table map and select **Edit Property Maps**.
   The Property Map Editor opens.
9. Scroll down to the row displaying the bonusPoint class attribute. Select **bonusPoint**.
10. Click in the corresponding entry in the **Map Type** column and select **Simple** from the drop-down list.
11. Click in the corresponding entry in the **Table Column** column and select **bonusPoint** from the drop-down list.
12. Leave all other fields unchanged and click **OK**.
    The new property map is generated, and after a few moments
    ```
    (a) bonusPoint (bonusPoint)
    ```
    is displayed in the **Property Maps** column of the Map Browser.
13. Right-click the **WCS User** datastore map, select **Save Datastore Map**, then click **Finish**.
14. Close the Map Browser.

At this point, the User bean contains errors indicating that some abstract classes are not implemented. These errors are fixed when deployed code is generated.

Generating the deployed code and access bean

Since you have modified the code for the User entity bean, you must regenerate its deployed code, as well as its access bean. The tools in VisualAge for Java make this code generation step very simple.

To perform this step, do the following:

1. Ensure that you have the Workbench open with the EJB tab selected.
2. Expand the **WCSUser** EJB group.
3. Right-click the **User** bean and select **Generate Deployed Code**.
   If a message window questioning whether or not to create an edition of the package is displayed, click **Yes**.
   Code generation takes a few minutes.
4. Once the deployed code has been generated, right-click the **User** bean again and select **Add > Access Bean**. The Create Access Bean SmartGuide opens.
5. Accept the default values on the Select Access Bean Properties page and click **Next**.

6. Accept the default values on the Define Zero Argument Constructor page and click **Next**.
7. On the Select and Customize Bean Properties for Copy Helper page, scroll down to **bonusPoint** in the Enterprise Bean column. Check **Copy Helper** for the bean and set the converter to `com.ibm.commerce.base.objects.WCSStringConverter`.
8. Click **Finish**.
9. Click **OK** when the "Code generation completed" message is displayed.

Testing the modification using the test client

A test client can be used to test the newly customized User entity bean. To start the test client and test the entity bean, do the following:

1. Start the persistent name server, as described in "Starting and stopping the persistent name server" on page 333.
2. Start the EJB server that has the WCSUser EJB group on it, as described in "Starting and stopping the EJB server" on page 334.
3. In the Enterprise Beans pane, expand the **WCSUser** EJB group. Right-click the **User** entity bean and select **Run Test Client**.
4. In the EJB Lookup window, click **Lookup**.
5. From the list of methods, select **findByPrimaryKey(MemberKey)**.
6. In the Details pane, click **`<null>`**, then enter `-1000` in the argument box, and click the Invoke icon in the EJB Test Client window.

   Note that the value entered here must match a value in the USERS_ID column of the USERS table.

   When the execution of this method is complete, information for the user with the ID of -1000 is displayed in a tree view in the Methods pane. Within this view, there is a node called *methods*.
7. Expand the **Methods** node.
8. Click **getBonusPoint()**, then the Invoke icon.
   This method retrieves the customer's balance of bonus points. The current balance of bonus points is returned.
9. Click **setBonusPoint(int)**, input `100` in the Details pane and then click the Invoke icon.
10. Click **getBonusPoint()**, then the Invoke icon.
    A value of 100 is returned. This shows that the database has been successfully updated by the User enterprise bean.
11. Close the User and EJB Test Client windows.

Creating the `GetNewProductContractUnitPriceCmd` interface

As part of this customization exercise, you create new business logic to calculate a discounted price, based upon the customer's balance of bonus points. This calculation is performed by a new task command. In this section, you create the interface for this new task command.

To create the interface for your new task command, do the following:

1. In the Workbench, with Projects tab selected, expand the **_WCSamples** project.
2. Right-click the **com.ibm.commerce.sample.commands** package and select **Add > Interface**.
3. Ensure that **Create a new interface** is selected and in the **Interface name** field, enter `GetNewProductContractUnitPriceCmd`.
4. Select the interface that should be extended by clicking **Add**, then in the **Pattern** field, enter `com.ibm.commerce.price.commands.GetProductContractUnitPriceCmd`, click **Add**, then **Close**.
5. Click **Next**.
6. To add the appropriate import statements, click **Add Package**, and then do the following:
   a. In the **Pattern** field, enter `com.ibm.commerce.price.utils` and click **Add**.
   b. In the **Pattern** field, enter `com.ibm.commerce.exception` and click **Add**.
   c. Click **Close**.
7. Ensure that **Make the interface public** is selected.
8. Click **Finish**.
   The source code for the new interface is displayed in the Source pane.
9. Create a method signature for the `getBonusPrice()` method, by doing the following:
   a. Right-click the **GetNewProductContractUnitPriceCmd** interface, and select **Add > Method**.
      The Create Method SmartGuide opens.
   b. Ensure that **Create a new method** is selected and click **Next**.
   c. In the Method Name field, enter `getBonusPrice`.
   d. To select the method's return type, click **Browse**. In the **Pattern** field, enter `MonetaryAmount` and click **OK**, then click **Next**.
   e. To select the exception that the method may throw, click **Add**. In the **Pattern** field, enter `ECSystemException` and click **Add**, then **Close**.
   f. Click **Finish**.
10. Add a new field to the interface that specifies the default implementation class for the command, by doing the following:
    a. Right-click the **GetNewProductContractUnitPriceCmd** interface, and select **Add > Field**.
       The Create Field SmartGuide opens.

b. In the **Field Name** field, enter `defaultCommandClassName`.
c. From the **Field Type** drop-down list, select **String**.
d. In the **Initial Value** field, enter

```
"com.ibm.commerce.sample.commands.
   GetNewContractUnitPriceCmdImpl"
```

and click **Finish**.

Notes:

1) You must include the double quotation marks when entering this value.
2) If a warning message indicating that a field in the superclass will be hidden by the creation of this new field, click **Yes** to proceed.

11. Add a new field to the interface that specifies the name of the command, by doing the following:
   a. Right-click the **GetNewProductContractUnitPriceCmd** interface, and select **Add > Field**.
   The Create Field SmartGuide opens.
   b. In the **Field Name** field, enter `NAME`.
   c. From the **Field Type** drop-down list, select **String**.
   d. In the **Initial Value** field, enter

```
"com.ibm.commerce.sample.commands.
   GetNewProductContractUnitPriceCmd"
```

   and click **Finish**.

Creating the `GetNewContractUnitPriceCmdImpl` implementation class

You must create the implementation class for the new task command. This implementation class implements your newly created interface and contains the business logic for the task command.

To create the `GetNewContractUnitPriceCmdImpl` implementation class, do the following:

1. In the Workbench, with Projects tab selected, expand the **_WCSamples** project.
2. Right-click the **com.ibm.commerce.sample.commands** package and select **Add > Class**.
   The Create Class SmartGuide opens.
3. Ensure that **Create a new class** is selected and create the class as follows:
   a. In the **Class name** field, enter `GetNewContractUnitPriceCmdImpl`.
   b. To specify the superclass, click **Browse**, then in the **Pattern** field, enter `com.ibm.commerce.price.commands.GetContractUnitPriceCmdImpl`, and click **OK**.

c. Click **Next**.
d. To specify the packages that should be imported, click **Add Package**. In the **Pattern** field, enter the following packages:
   - `com.ibm.commerce.command` and click **Add**
   - `com.ibm.commerce.exception` and click **Add**
   - `com.ibm.commerce.price.commands` and click **Add**
   - `com.ibm.commerce.price.utils` and click **Add**
   - `com.ibm.commerce.ras` and click **Add**
   - `com.ibm.commerce.server` and click **Add**
   - `java.math` and click **Add**, then **Close**.
e. To specify the interfaces that the class should implement, click **Add**. In the Pattern field, enter the following interfaces:
   - `GetContractSpecialPriceCmd` and click **Add**.
   - `GetContractUnitPriceCmd` and click **Add**.
   - `GetProductContractUnitPriceCmd` and click **Add**.
   - `GetNewProductContractUnitPriceCmd` and click **Add**, then **Close**.
f. Click **Finish**.

4. Right-click the **GetNewContractUnitPriceCmdImpl** class and select **Add > Field**. The Create Field SmartGuide opens. Enter information as follows:
   a. In the **Field Name** field, enter `bonusPrice`.
   b. To select the field type, click **Browse**. In the Pattern field, enter `MonetaryAmount` and click **OK**.
   c. Click **Finish**.
5. Add a new `performExecute()` method to the class. This method does the following:
   - calls the `performExecute()` method of the superclass (GetContractUnitPriceCmdImpl)
   - sets the `thisClass` and `methodName` values to be used by the exception handling mechanism
   - instantiates a StoreAccessBean
   - instantiates a UserAccessBean
   - gets the original product price
   - calculates the maximum applicable discount
   - calculates the new bonus price (this price is of the type *double*)
   - gets the currency type for the store
   - rounds off the bonus price and stores it as a monetary amount in the correct currency

To add this method, right-click the **GetNewContractUnitPriceCmdImpl** class and select **Add > Method**. The Create Method SmartGuide opens. Enter information as follows:

a. Ensure that **Create a new method** is selected and click **Next**.

b. In the **Method Name** field, enter `performExecute`.

c. From the **Return Type** drop-down list, select **void** and click **Next**.

d. To select the exception that the method may throw, click **Add**. In the **Pattern** field, enter `ECException` and click **Add**, then **Close**.

e. Click **Finish**.

f. After this line in the source code:

```
public void performExecute()
   throws com.ibm.commerce.exception.ECException
{
```

add the following code:

You can cut and paste this code from the PDF version of the Programmer's Guide. It is recommended that you initially copy the code in the Scrapbook window in VisualAge for Java (refer to the VisualAge for Java online help for more information) and inspect the code to ensure no characters were lost during the cut and paste operation. Then, after validating the code, copy it into the target location. Note that copying the text into another editor may cause some characters to be modified.

```
super.performExecute();

// Get and set this class name and method
// for use when exceptions occur.
final String thisClass =
   GetContractUnitPriceCmdImpl.class.getName();
final String methodName = "performExecute";

//get the store access bean
Integer storeId = getStoreId();
com.ibm.commerce.common.objects.StoreAccessBean storeAB =
     getCommandContext().getStore(storeId);

//get the user access bean
com.ibm.commerce.user.objects.UserAccessBean bonusAB =
     new com.ibm.commerce.user.objects.UserAccessBean();

// get the calculated price from the GetContractUnitPriceCmdImpl
MonetaryAmount priceOrg = super.getPrice();
double dblPriceOrg = priceOrg.getValue().doubleValue();

//calculate the maximum bonus that can apply to this product
double dblBonusPrice; // = dblPriceOrg;
double dblMaxBonusPoint = 0;
```

```
try {
bonusAB.setInitKey_MemberId(super.getUserId().toString());
bonusAB.refreshCopyHelper();
double dblMaxDed = dblPriceOrg * 0.2;
dblMaxBonusPoint =
     (new java.math.BigDecimal(
        bonusAB.getBonusPoint()).doubleValue());
if (dblMaxBonusPoint > dblMaxDed) dblMaxBonusPoint = dblMaxDed;
} catch (javax.ejb.CreateException ex) {
throw new ECSystemException(
     ECMessage._ERR_CREATE_EXCEPTION, thisClass, methodName, ex);
} catch (javax.ejb.FinderException ex) {

} catch (javax.naming.NamingException ex) {
throw new ECSystemException(
     ECMessage._ERR_GENERIC, thisClass, methodName, ex);
} catch (java.rmi.RemoteException ex) {
throw new ECSystemException(
     ECMessage._ERR_REMOTE_EXCEPTION, thisClass, methodName, ex);
}


//apply the maximum applicable bonus to this product price
dblBonusPrice = dblPriceOrg - dblMaxBonusPoint ;

//get the currency of this store
CommandContext context = getCommandContext();
String requestedCurrency = Helper.getCurrency( context, storeAB );

//round off and return the bonus price in MonetoryAmount type
bonusPrice = new MonetaryAmount(
     new BigDecimal(dblBonusPrice), requestedCurrency);
CurrencyManager.getInstance().roundCustomized(bonusPrice, storeAB);
```

Save your work.

6. Select the **getBonusPrice()** method in the **GetNewContractUnitPriceCmdImpl** class to view its source code. In the source code, change

   ```
   return null;
   ```

   to

   ```
   return bonusPrice;
   ```

   Save your work.

Creating the `NewProductDataBean` data bean

The `getCalculatedBonusPrice()` method must be added to the existing WebSphere Commerce ProductDataBean. Since you must not actually modify the code for the ProductDataBean, you must create a new data bean that extends the ProductDataBean and then add the method to the new data bean.

To create the new data bean, do the following:

1. In the Workbench, with the Projects tab selected, expand the **_WCSamples** project.
2. Right-click the **com.ibm.commerce.sample.databeans** package and select **Add > Class**. The Create Class SmartGuide opens. Enter information as follows:
   a. In the **Class name** field, enter `NewProductDataBean`.
   b. To specify the superclass, click **Browse**, then in the **Pattern** field enter `com.ibm.commerce.catalog.beans.ProductDataBean`. Click **OK**, then **Next**.
   c. Add the appropriate import statements to the class by clicking **Add Package**, then do the following:
      - Enter `com.ibm.commerce.beans` and click **Add**.
      - Enter `com.ibm.commerce.catalog.objects` and click **Add**.
      - Enter `com.ibm.commerce.command` and click **Add**.
      - Enter `com.ibm.commerce.datatype` and click **Add**.
      - Enter `com.ibm.commerce.exception` and click **Add**.
      - Enter `com.ibm.commerce.price.beans` and click **Add**.
      - Enter `com.ibm.commerce.ras` and click **Add**.
      - Enter `com.ibm.commerce.sample.commands` and click **Add**.
      - Enter `com.ibm.commerce.server` and click **Add**.
      - Enter `java.util` and click **Add**, then **Close**.
   d. Click **Finish**.
3. Right-click the **NewProductDataBean** class and select **Add > Method**. The Add Method SmartGuide opens.
4. Ensure that **Create new method** is selected and click **Next**.
5. In the **Method Name** field, enter `getCalculatedBonusPrice`.
6. To select the return type, click **Browse**. In the **Pattern** field, enter `PriceDataBean`, click **OK**, then **Next**.
7. To select the exception that the method may throw, click **Add**. In the **Pattern** field, enter `ECSystemException` and click **Add**, then **Close**.
8. Click **Finish**.
9. In the source code, replace `return null;` with the following:

```
PriceDataBean ibnPrice = null;
try {
     GetNewProductContractUnitPriceCmd comm =
        (GetNewProductContractUnitPriceCmd) CommandFactory.createCommand
        (GetNewProductContractUnitPriceCmd.NAME,
             getCommandContext().getStoreId());

     ECTrace.trace(ECTraceIdentifiers.COMPONENT_CATALOG,
     this.getClass().getName(), "getCalculatedBonusPrice",
     "Getting Price for CatalogEntry: " + getProductID());

     comm.setCatEntryId(new Long(getProductID()));
     comm.setCommandContext(getCommandContext());
     comm.execute();
     ibnPrice = new PriceDataBean(comm.getBonusPrice(),
          getCommandContext().getStore(),
          getCommandContext().getLanguageId());

} catch (Exception e) {
     throw new ECSystemException(ECMessage._ERR_RETRIEVE_PRICE,
          this.getClass().getName(), "getCalculatedBonusPrice",e);
}

return ibnPrice;
```

Save your work.

Adding the new bonus price to the product display template

The next step is to add the new bonus price to the product display template, so that shoppers will be able to see the customized price. Once you have updated the display template, the new discounted price is displayed.

The sample store uses the `ProductDisplay.jsp` template for displaying products. Therefore, you must update this template with information to display the new price.

To update the display template, do the following:

1. Navigate to the following directory:
   vaj_drive:\VAJava\ide\project_resources\IBM WebSphere Test Environment\hosts\default_host\default_app\web\store_name
2. Make a copy of the `ProductDisplay.jsp` file and name it `ProductDisplay.jsp.bak`.
3. Open `ProductDisplay.jsp` in a text editor.
4. After the `<%@ page import="com.ibm.commerce.common.beans.*" %>`, add the following import statements:
   ```
   <%@ page import="com.ibm.commerce.sample.commands.*" %>
   <%@ page import="com.ibm.commerce.sample.databeans.*" %>
   ```
5. Replace all occurrences of `ProductDataBean` with `NewProductDataBean`.
6. Locate the following line:
   ```
   <font class="price"><%=product.getCalculatedContractPrice()%></font>
      <br><br>
   ```

   After this line, insert the following line to retrieve and display the bonus price for the product

```
<font class="price"><%=product.getCalculatedBonusPrice()%>
      Bonus Price </font>
<br><br>
```

7. Save this file.

Note: If you cut and paste sections into the display template from the PDF version of the *WebSphere Commerce Programmer's Guide*, ensure that no characters are modified during that process.

Testing the enterprise bean extension

In this section, you can test the extension that you have made to the enterprise bean, by viewing a product in the InFashion sample store. The new bonus price is displayed.

Note, for the simplicity of this sample, the bonus price is viewable to all shoppers (registered or guest shoppers). For shoppers that have no bonus points, the bonus price is the same as the regular price.

To test the enterprise bean extension and see the bonus price displayed, do the following:

1. Verify that the `_WCSamples` project is included in the path for the Servlet Engine, by doing the following:
   a. From the **Workspace** menu in VisualAge for Java, select **Tools > WebSphere Test Environment**.
      The WebSphere Test Environment Control Center opens.
   b. Click **Servlet Engine**.
   c. If the Servlet Engine is running, click **Stop Servlet Engine** and then **Edit Class Path**.
   d. If **_WCSamples** is not already selected, select it now and click **OK**.
2. Start the WebSphere Test Environment as described in Appendix A, "Starting and stopping the WebSphere Test Environment" on page 333. The persistent name server and EJB server may already be running, in this case, you need only start the servlet engine.
3. Open a browser and enter the following URL

   ```
   http://localhost:8080/webapp/wcs/stores/servlet/StoreCatalogDisplay?
        storeId=store_Id&catalogId=catalog_Id&langId=-1
   ```
4. Click the **Register** link under the Services heading and then click **Register** under the New Customer heading.
   Register a new customer using the e-mail address of `wctester@wc` and a password of `wctester1`. Fill in other fields with test values and click **Submit**. Leave the browser open.
5. ▶ DB2 Open the DB2 Command Center and do the following:
   a. Click the **Interactive** tab

b. In the **Command** field, do the following:

   1) Enter

      ```
      connect to your_database_name
      ```

      where *your_database_name* is the name of your WebSphere Commerce database and click the Execute icon.

   2) Enter `select users_id from userreg where logonid = 'wctester@wc'`and click the Execute icon.

c. The Query Results tab displays the entry for the customer you registered in step 4. Make note of the customer's USERS_ID value here: ______________

d. Update the newly registered customer's balance of bonus points. Click the Interactive tab and in the **Command** field, enter the following:

   ```
   update BONUS set BONUSPOINT = 1000 where MEMBERID = users_id
   ```

   where *users_id* is the value from step 5c. Click the Execute icon.

6. Oracle Update the test user's bonus points balance, by doing the following:

   a. Open the Oracle SQL Plus command window (**Start > Programs > Oracle > Application Development > SQL Plus**).

   b. In the **User Name** field, enter your Oracle user name.

   c. In the **Password** field, enter your Oracle password.

   d. In the **Host String** field, enter your connect string.

   e. Enter `select users_id from userreg where logonid = 'wctester@wc';`

   f. The entry for the customer you registered in step 4 is displayed. Make note of the customer's USERS_ID value here: ______________

   g. Update the newly registered customer's balance of bonus points by entering the following:

      ```
      update BONUS set BONUSPOINT = 1000 where MEMBERID = users_id;
      ```

      where *users_id* is the value from step 6f.

   h. Enter the following to commit your database changes:

      ```
      commit;
      ```

      and press Enter to run the SQL statement.

7. In the browser, click the **Men's** link to view the Men's fashions section of the store.

8. Click the link for the featured special to view the product page. The page displays the regular price, and the discounted price based upon the customer's balance of bonus points.

Note: If a stack trace is displayed instead of the bonus price, you may need to disable caching. This can be done by setting the CacheDaemon component value to false in the *instance_name*.xml file, as shown below:

```
<component compClassName=
      "com.ibm.commerce.cache.daemon.CacheDaemonComponent"
               enable="false"
               name="CacheDaemon" />
```

After changing the value in the *instance_name*.xml file, you must stop and restart the servlet engine in the WebSphere Test Environment.

(Optional) Deploying the customized business logic to a remote WebSphere Commerce Server

This section describes how to deploy the modified entity bean and new task command to store that is running outside of the WebSphere Test Environment.

Deployment involves creating JAR files for the WebSphere Commerce public enterprise beans and the command and data bean logic, placing the JAR files in the appropriate directories on the target server, stopping the WebSphere Commerce instance, modifying class paths, deploying the enterprise beans using the XMLConfig utility and restarting the instance.

Creating the JAR file for the new price command

You must create a JAR file for the **_WCSamples** project so the new task command gets deployed. To create this JAR file, do the following on the development machine:

1. Stop the WebSphere Test Environment, as described in Appendix A, “Starting and stopping the WebSphere Test Environment” on page 333.
2. With the Projects tab selected, select the **_WCSamples** project.
3. With the project highlighted, right-click and select **Export**.
   The Export SmartGuide opens.
4. Select **Jar file** and click **Next.**
5. In the **Jar file** field, enter the following:
   `*drive*:\WebSphere\CommerceServerDev\mytemp_c\wcssamplesc_1.jar`
   where *drive* is the drive on which WebSphere Commerce is installed.
6. Select attributes as follows:

Attribute	**Value**
class	Checked
java	Unchecked
resource	Checked
beans	Checked

Attribute	Value
Include debug attributes in .class file	Checked

Accept the default values for other attributes.

7. Click **Finish**.

Since the JAR file created does not contain complete package naming information, you must use another packaging utility (outside of VisualAge for Java) to repackage the JAR file. To repackage this file, do the following:

1. In a command window, navigate to the following directory:`*drive*:\WebSphere\CommerceServerDev\mytemp_c`
2. Enter `mkdir temp3`.
3. Enter `cd temp3`.
4. Set the path as follows:
   `set PATH=%PATH%;*drive*:\WebSphere\WebSphereStudio4\bin;`
   where *drive* is the drive on which WebSphere Studio is installed.
5. Enter `jar xvf ../wcssamplesc_1.jar`.
6. Enter `jar cvf ../wcssamplesc.jar *` (note that the `_1` is removed from the name).

Creating a JAR file for the WCSUser EJB group

You must create an EJB 1.1 Export JAR file containing for the EJB group that contains the modified enterprise bean. As such, you must select the following group when creating the JAR file:

- WCSUser

To create the JAR file for the WCSUser EJB group, do the following:

1. With the EJB tab selected, highlight the **WCSUser** EJB group.
2. Right-click the **WCSUser** EJB group and select **Export > EJB 1.1 JAR**. The Export to an EJB 1.1 JAR File SmartGuide opens.
3. In the **JAR file** field, enter
   `*drive*:\WebSphere\CommerceServerDev\mytemp_c\`
   `CustomizedWCSUserDeployed_DT.jar`
4. Select attributes as follows:

Attribute	Value
class	Checked
java	Unchecked
resource	Checked

Attribute	Value
Target database	DB2 If you are deploying to a DB2 database, select **DB2 for NT, V7.1**. Oracle If you are deploying to an Oracle database, select **Oracle, V8**.
Include debug attributes in .class file	Checked

Accept the default values for other attributes.

5. Click **Finish**.

The JAR file is created.

The JAR file has been named with the "_DT" suffix as a reminder that you must run this JAR file through the EJB Deploy Tool provided by WebSphere Application Server before deploying it into your WebSphere Commerce application.

Creating the `wcsejsclient.jar` file

To create the client JAR file, do the following:

1. With the EJB tab selected, highlight the all of the WebSphere Commerce EJB groups (the name begin with `WCS`). With all of these groups highlighted, right-click and select **Export > Client JAR**.
   The Export SmartGuide opens.
2. In the **JAR file** field, enter
   `*drive*:\WebSphere\CommerceServerDev\mytemp_c\wcsejsclient.jar`
3. Select attributes as follows:

Attribute	Value
beans	Checked
class	Checked
java	Unchecked
resource	Checked
Include debug attributes in .class file	Checked

Accept the default values for other attributes.

4. Click **Finish**.

The JAR file is created.

Copying the updated JSP template to the target store directory

In "Adding the new bonus price to the product display template" on page 313, you updated the display template to reflect the newly created price. In this step you copy the updated JSP template from the WebSphere Test Environment directory structure into the directory used by the store when running outside of the WebSphere Test Environment.

1. On the development machine, navigate to the following directory: `vaj_drive:\VAJava\Ide\project_resources\IBM WebSphere Test Environment \hosts\default_host\default_app\web\store_directory` where *vaj_drive* is the drive on which you installed VisualAge for Java and *store_directory* is the name of the directory for the sample store. Copy the `ProductDisplay.jsp` file.
2. Paste the `ProductDisplay.jsp` file into the following directory:

   ```
   drive:\WebSphere\AppServer\installedApps\
      WC_Enterprise_App_instance_name.ear\
      wcstores.war\store_directory
   ```

   where *drive* is the drive on which WebSphere Commerce is installed, *store_directory* is the directory name for the store, and *instance_name* is the name of your WebSphere Commerce instance.

Copy the JAR files to the target WebSphere Commerce Server

You must copy the JAR files from the development machine into the appropriate directory on the target WebSphere Commerce Server. To copy these files, do the following:

1. On the development machine, navigate to the following directory: `drive:\WebSphere\CommerceServerDev\mytemp_c` and locate the following files:
   - `wcssamplesc.jar`
   - `CustomizedWCSUserDeployed_DT.jar`
   - `wcsejsclient.jar`

   where *drive* is the drive onto which you installed WebSphere Commerce Studio, Business Developer Edition.

   Each of the preceding files must be copied into a particular directory on the target WebSphere Commerce Server. Read the following steps carefully to ensure that each file is stored in the correct location.
2. Copy the `wcssamplesc.jar` file into the following directory on the target WebSphere Commerce Server:

   ```
   drive:\WebSphere\AppServer\installedApps\
      WC_Enterprise_App_instance_name.ear\
      wcstores.war\WEB-INF\lib
   ```

where *drive* is the drive onto which you installed WebSphere Commerce Business Edition and *instance_name* is the name of your instance (for example, demo).

3. Copy the wcsejsclient.jar file into the following directory the target WebSphere Commerce Server:

   ```
   drive:\WebSphere\CommerceServer\temp\lib
   ```

4. Copy the CustomizedWCSUserDeployed_DT.jar file into the following directory on the target WebSphere Commerce Server:

   ```
   drive:\WebSphere\CommerceServer\temp
   ```

Running the EJB deploy tool

You must run the EJB deploy tool against the JAR file containing the new EJB group. This tool is included with WebSphere Application Server.

To run this tool, do the following:

1. At a command prompt, navigate to the following directory:

   ```
   drive:\WebSphere\CommerceServer\temp
   ```

2. Temporarily add the tool to the system path by entering the following command:

   ```
   PATH=drive:\WebSphere\AppServer\deploytool;%PATH%
   ```

3. Enter the ejbdeploy command as follows:

   ```
   ejbdeploy EJBGroupJARFile WorkingDir OutputJARFile -nowarn -keep -35 -cp
      ClassPathOfDepJARFiles
   ```

   where:

   - *EJBGroupJARFile* is the name the JAR file for your EJB group. In this case, this is CustomizedWCSUserDeployed_DT.jar.
   - *WorkingDir* is the working directory.
   - *OutputJARFile* is the name of the output JAR file. In this case, enter CustomizedWCSUserDeployed.jar.
   - -nowarn is an optional parameter to suppress warning and information messages.
   - -keep is an optional parameter to retain the working directory after the ejbdeploy command has run.
   - -35 is a mandatory parameter that will use the same top-down mapping rules for CMP entity beans that are used in the EJB Deploy Tool that was provided with the WebSphere Application Server, Version 3.5.
   - -cp *ClassPathOfDepJARFiles* is the class path of any dependent JAR files. When you have modified an existing WebSphere Commerce enterprise bean you must include the wcsejsclient.jar, wcsejbimpl.jar, and xml4j.jar files in the class path of dependent JAR files. As such, enter the following:

```
"drive:\WebSphere\CommerceServer\temp\lib\wcsejsclient.jar;
drive:\WebSphere\AppServer\InstalledApps\
WC_Enterprise_App_instanceName.ear\lib\wcsejbimpl.jar;
drive:\WebSphere\AppServer\InstalledApps\
WC_Enterprise_App_instanceName.ear\lib\xml4j.jar;"
```

Modify transaction isolation level for the entity beans

In this step you use the `modifyIsolationLevel` command to modify the transaction isolation level of the entity beans to the required level for your specific type of database.

To run the `modifyIsolationLevel` command, do the following:

1. On the target WebSphere Commerce Server, use a command prompt to navigate to the following directory:

   ```
   drive:\WebSphere\CommerceServer\bin
   ```

2. You must issue the `modifyIsolationLevel` command which has the following general syntax:

   ```
   modifyIsolationLevel -jarFile jar_file_name.jar
      -logFile log_file_name -dbType db_type
   ```

   where

   - *`jar_file_name.jar`* is the name of the JAR file that contains the customized code
   - *`log_file_name`* is the fully qualified file name where information should be logged
   - *`db_type`* is the type of database you are using. Enter either DB2 or ORACLE

   The following is an example of the `modifyIsolationLevel` command with all values specified:

   ```
   modifyIsolationLevel -jarFile
      D:\WebSphere\CommerceServer\temp\CustomizedWCSUserDeployed.jar
      -logFile D:\WebSphere\CommerceServer\instances\demo\logs\output.log
      -dbType DB2
   ```

Note: The parameter names are case sensitive. That is, `jarFile` is not the same as `jarfile`. Ensure that you enter the parameter names correctly.

The command has run successfully if no exceptions are displayed in the command window. After completion, note that the timestamp on your deployed JAR file has changed.

Updating the target database

If you are deploying to a target WebSphere Commerce Server that uses a different database than the one used by the WebSphere Test Environment, you must update the target database, as follows:

- If you completed Chapter 9, “Tutorial: Creating new business logic” on page 201, you must update the table to add a row for each user in the USERS table.
- If you did not complete Chapter 9, “Tutorial: Creating new business logic” on page 201, you must create and populate the table.

If you did not complete Chapter 9, “Tutorial: Creating new business logic” on page 201, you must create and populate the table. Instructions for each of these scenarios is provided.

DB2 If you are using a DB2 database and need to *update* the BONUS table, do the following:

1. Open the DB2 Command Center (**Start > Programs > IBM DB2 > Command Center**) and click the **Script** tab.
2. In the **Script** field, enter

   ```
   connect to your_database_name
   ```

   where *your_database_name* is the name of your database and click the Execute icon.
3. In the **Script** field, enter the following, then click the Execute icon:

   ```
   INSERT INTO BONUS
   (SELECT USERS_ID, 0
   FROM USERS
   WHERE USERS_ID NOT IN (SELECT MEMBERID FROM BONUS))
   ```

   The BONUS table has now been updated.

DB2 If you are using a DB2 database and need to *create and populate* the BONUS table, do the following:

1. Open the DB2 Command Center (**Start > Programs > IBM DB2 > Command Center**) and click the **Script** tab.
2. In the **Script** field, enter

   ```
   connect to your_database_name
   ```

   where *your_database_name* is the name of your database and click the Execute icon.
3. In the **Script** field, enter the following, then click the Execute icon:

   ```
   CREATE TABLE BONUS (MEMBERID BIGINT NOT NULL,
       BONUSPOINT INTEGER NOT NULL, constraint p_memberid primary key (MEMBERID),
       constraint f_memberid foreign key (MEMBERID)
       references users (users_id) on delete cascade)
   ```

   The BONUS table has now been created.

4. To populate the table, enter the following in the **Script** field, then click the Execute icon:

   ```
   insert into BONUS (select USERS_ID, 0 from USERS)
   ```
5. Close the DB2 Command Center.

Oracle If you are using an Oracle database, and need to *update* the BONUS table, do the following:

1. Open the Oracle SQL Plus command window (**Start > Programs > Oracle > Application Development > SQL Plus**).
2. In the **User Name** field, enter your Oracle user name.
3. In the **Password** field, enter your Oracle password.
4. In the **Host String** field, enter your connect string.
5. Update the BONUS table, by entering the following information in the SQL Plus window:

   ```
   INSERT INTO BONUS
   (SELECT USERS_ID, 0
   FROM USERS
   WHERE USERS_ID NOT IN (SELECT MEMBERID FROM BONUS));
   ```

   Click Enter to run the SQL statement.
   The BONUS table has now been updated.
6. Enter the following to commit your database changes:

   ```
   commit;
   ```

   and press Enter to run the SQL statement.

Oracle If you are using an Oracle database, and need to *create and populate* the BONUS table, do the following:

1. Open the Oracle SQL Plus command window (**Start > Programs > Oracle > Application Development > SQL Plus**).
2. In the **User Name** field, enter your Oracle user name.
3. In the **Password** field, enter your Oracle password.
4. In the **Host String** field, enter your connect string.
5. Create the BONUS table, by entering the following information in the SQL Plus window:

   ```
   CREATE TABLE Bonus (MEMBERID NUMBER NOT NULL,
       BONUSPOINT INTEGER NOT NULL, constraint p_memberid primary key (MEMBERID),
       constraint f_memberid foreign key (MEMBERID)
       references users (users_id) on delete cascade);
   ```

   Click Enter to run the SQL statement.
   The BONUS table has now been created.

6. Populate the BONUS table, by entering the following:

   ```
   insert into BONUS (select USERS_ID, 0 from USERS);
   ```

7. Enter the following to commit your database changes:

   ```
   commit;
   ```

   and press Enter to run the SQL statement.

Exporting the current enterprise application from WebSphere Application Server

In this step, you export the current enterprise application from WebSphere Application Server so that you can later open it in the Application Assembly Tool.

To export the current enterprise application from WebSphere Application Server, do the following:

1. Open the WebSphere Application Server Administration Console.
2. Expand **WebSphere Administrative Domain**.
3. Expand **Enterprise Application**.
4. Right-click your WebSphere Commerce application. For example, expand the **demo** application and select **Export Application**.
5. In the **Export directory** field, enter `*drive*:\WebSphere\CommerceServer\working`.
   This exports the whole application, including all resources into the `WC_Enterprise_App_*instanceName*.ear` file (where *instanceName* is the name of your WebSphere Commerce instance).

   Note: If you already have an existing `WC_Enterprise_App_*instanceName*.ear` file, you can rename the old file or overwrite it.

Exporting XML configuration information for the enterprise application

You must also export the XML configuration information for the enterprise application. To export this information, you use the `XMLConfig` command line utility provided by WebSphere Application Server.

To export this configuration information, do the following:

1. At a command prompt, navigate to the following directory:

   ```
   drive:\WebSphere\CommerceServer\working
   ```

2. Invoke the XMLConfig tool to perform a partial export by entering the following command:

   ```
   xmlConfig -export OutputFileB.xml -partial was.export.app.xml
      -adminNodeName wasHostName
   ```

where `wasHostName` is the name of the node in the WebSphere Application Server that contains your current enterprise application. Additionally, `OutputFileB.xml` is the name of the file that is created as a result of running this command.

After you have exported the information about the enterprise beans in the current enterprise application, you must update the `OutputFileB.xml` file so that it points to the JAR file that contains the code for the modified User bean.

To update the `OutputFileB.xml` file, do the following:

1. Navigate to the following directory:
   ```
   drive:\WebSphere\CommerceServer\working
   ```
2. Open the `OutputFileB.xml` file in a text editor.
3. Locate the `<ear-file-name>` tag and replace the value with the following:
   ```
   drive:\WebSphere\CommerceServer\working\
      WC_Enterprise_App_instanceName.ear
   ```
4. Locate the `<ejb-module>` stanza for the WCSUser EJB group and change the value contained in the `<jar-file>` tags to `CustomizedWCSUserDeployed.jar`
5. Save the `OutputFileB.xml` file.

Assembling the modified EJB group into the enterprise application

In this step, you open your enterprise application in the application assembler tool. Once it is open inside that tool, you can do the following to include the modified User bean to the enterprise application:

1. Make a copy of the class path for the existing version of the WCSUser EJB group.
2. Remove the existing version of the WCSUser EJB group.
3. Import the new version of the WCSUser EJB group. The JAR file for the new EJB group is stored within the EJB Module section of the enterprise application.
4. Set the class path for the WCSUser EJB group.

To assemble the new EJB group into the enterprise application, do the following:

1. Backup the current enterprise application, by doing the following:
   a. At a command prompt, navigate to the following directory:
      ```
      drive:\WebSphere\CommerceServer\working
      ```
   b. Enter the following command:
      ```
      copy WC_Enterprise_App_instanceName.ear
         WC_Enterprise_App_instanceName.ear.bak2
      ```
2. Open the WebSphere Application Server Administration Console.

3. From the **Tools** menu, select **Application Assembly Tool**. (If a Welcome window opens, select **Cancel** to access the console.)
4. Open the enterprise application upon which you are going to work, by doing the following:
   a. From the **File** menu, select **Open**.
   b. In the **File name** field, enter:

      ```
      drive:\WebSphere\CommerceServer\working\
         WC_Enterprise_App_instanceName.ear
      ```

      and click **Open**. Wait for the application to open before continuing to the next steps. This takes several minutes.
5. Click **EJB Modules**. The pane on the right displays the EJB modules in the enterprise application.
6. Click the **WCSUser** EJB module.
7. Click the General tab to view the class path information for the existing WCSUser EJB module. Copy this existing class path information into a text file (for example, `WCSUser_path.txt`).
8. Right-click the **WCSUser** EJB module and select **Delete**.
9. Right-click **EJB Modules** and select **Import**.
10. In the **File name** field, enter

    ```
    drive:\WebSphere\CommerceServer\temp\CustomizedWCSUserDeployed.jar
    ```

    and click **Open**. In the Confirm Values window, click **OK**.
11. Once the `CustomizedWCSUserDeployed.jar` file is imported, scroll to the **WCSUser** EJB group and select this group.
    Information about this group is shown in the pane on the right.
12. Open the text file containing the class path information for the previous version of the WCSUser EJB group. Select and copy the class path.
13. In the classpath field for the new WCSUser EJB group paste this class path information.
14. Click **Apply**.
15. From the **File** menu, select **Close**.
16. Wait for the file to close, then from the **File** menu, select **Open** and reopen the `drive:\WebSphere\CommerceServer\working\ WC_Enterprise_App_instanceName.ear` file.
17. Configure security for the User bean, by doing the following:
    a. With the EJB Modules node expanded, locate and expand the **WCSUser** node.
    b. Expand **Entity Beans**.
    c. Expand **User**

   d. Click **Method Extensions**, then in the pane on the right do the following:
      1) Click the **Advanced** tab.
      2) Ensure that **Security identity** is selected.
      3) For each method, ensure that **Use identity of EJB server** is selected.
      4) Click **Apply** (if you have made any modifications).
   e. In the left navigation pane, right-click **Security Roles** under the `WCSUser` EJB group and select **New**, then do the following:
      1) In the **Name** field, enter `WCSecurityRole` and click **Apply**. Note, if this role exists already, you do not need to perform this step.
   f. In the left navigation pane, right-click **Method Permissions** under the `WCSUser` EJB group and select **New**, then do the following:
      1) In the **Method Permission Name** field, enter `WCMethodPermission`
      2) In the **Methods** selection area, click **Add**.
         The Add Methods window opens.
      3) Expand **CustomizedWCSUserDeployed.jar** and select all of the enterprise beans (hold the Shift key while selecting). Click **OK**. All enterprise beans are then displayed under the Enterprise bean column and **All methods** is displayed under the Types column.
      4) In the Roles selection area, click **Add**, select the `WCSecurityRole` and click **OK**.
      5) Click **Apply**, then click **OK**.

18. From the **File** menu, select **Save**.
19. Close the Application Assembler Tool.

After this step has completed, you have created a new enterprise application that contains all of the previous logic as well as your new business logic. This is all contained in the newly modified `WC_Enterprise_App_`*`instanceName`*`.ear` file.

Importing the new enterprise application into WebSphere Application Server

The following are the high-level steps involved in importing the new enterprise application into WebSphere Application Server:

1. Stopping the enterprise application that is currently running in WebSphere Application Server and then removing it. These steps are performed in the WebSphere Application Server Administration Console.
2. Importing the new application, using the XMLConfig command line utility.
3. Refreshing the WebSphere Application Server Administator's Console and then starting the new enterprise application.

Each of these steps is described in more detail in the following sections.

Stopping and removing the current enterprise application: To stop and remove your current enterprise application from WebSphere Application Server, do the following:

1. Open the WebSphere Application Server Administration Console.
2. Expand **WebSphere Administrative Domain**.
3. Expand **Nodes**.
4. Expand *nodeName* (where *nodeName* is the name of your node).
5. Expand **Application Servers**.
6. Right-click your WebSphere Commerce application. For example, right-click the **WebSphere Commerce Server -** *instanceName* and select **Stop**.
7. Expand **Enterprise Applications**.
8. Right-click your WebSphere Commerce application. For example, right-click the **WebSphere Commerce Enterprise Server - demo** application and select **Stop**.
9. Right-click your WebSphere Commerce application. For example, right-click the **WebSphere Commerce Enterprise Server - demo** application and select **Remove**.
10. When prompted to indicate if the application should be exported, select **No**.

Importing the new enterprise application using XMLConfig: To import the new enterprise application by using the XMLConfig command line utility, do the following:

1. Navigate to the following directory:
   ```
   drive:\WebSphere\CommerceServer\working
   ```
2. At the command prompt, enter the following command to import the enterprise application into WebSphere Application Server:
   ```
   xmlConfig -import OutputFileB.xml -adminNodeName was_hostname
   ```

   where `was_hostname` is the name of the node of the WebSphere Application Server containing the current application.

Note: ▶ 400 If you were deploying to a WebSphere Commerce instance running on iSeries, you would have to perform an additional step to modify directory permissions after you have imported the application. Refer to “Importing an enterprise application” on page 366 for details on how to modify these permissions.

Starting the new enterprise application: After you have imported the new enterprise application using the XMLConfig command line utility, you can use the WebSphere Application Server Administrator's Console to perform a refresh and then start the new application.

To refresh the console and start the new application, do the following:

1. Open the WebSphere Application Server Administration Console.
2. Expand **WebSphere Administrative Domain**
3. Highlight **Nodes**.
4. Click the **Refresh selected subtree** icon.
5. Start your WebSphere Commerce application by doing the following:
   - Expand **Application Servers**.
   - Right-click your WebSphere Commerce application. For example, right-click the **WebSphere Commerce Server -** *instanceName* and select **Start**.

Testing the new code in the target store

To test the modified entity bean and new task command in the store running in WebSphere Application Server, do the following:

1. Open a browser and enter the following URL:

   ```
   http://hostname/webapp/wcs/stores/servlet/StoreCatalogDisplay?
        storeId=store_Id&catalogId=catalog_Id&langId=-1
   ```

   where *store_Id* is the identifier for your store and *catalog_Id* is the identifier for your store's catalog.

   Note: If you receive an Error 500, you may need to restart WebSphere Application Server in order to refresh your enterprise application.
2. Click the **Register** link under the Services heading and then click **Register** under the New Customer heading.
   Register a new customer using the e-mail address of `wctester@wc` and a password of `wctester1`. Fill in other fields with test values and click **Submit**. Leave the browser open.
3. ▶ DB2 Open the DB2 Command Center and do the following:
   a. Click the **Interactive** tab
   b. In the **Command** field, do the following:
      1) Enter

         ```
         connect to your_database_name
         ```

         where *your_database_name* is the name of your WebSphere Commerce database and click the Execute icon.
      2) Enter `select users_id from USERREG where LOGONID = 'wctester@wc'` and click the Execute icon.

c. The Query Results tab displays the entry for the customer you registered in step 2. Make note of the customer's USERS_ID value here: _______________

d. Update the newly registered customer's balance of bonus points. Click the Interactive tab and in the **Command** field, enter the following:

```
update BONUS set BONUSPOINT = 1000 where MEMBERID = users_id
```

where *users_id* is the value from step 3c. Click the Execute icon.

4. Oracle Update the test user's bonus points balance, by doing the following:

   a. Open the Oracle SQL Plus command window (**Start > Programs > Oracle > Application Development > SQL Plus**).
   b. In the **User Name** field, enter your Oracle user name.
   c. In the **Password** field, enter your Oracle password.
   d. In the **Host String** field, enter your connect string.
   e. Enter `select users_id from USERREG where LOGONID = 'wctester@wc';` to determine the user's ID.
   f. The entry for the customer you registered in step 2 is displayed. Make note of the customer's USERS_ID value here: _______________
   g. Update the newly registered customer's balance of bonus points by entering the following:

   ```
   update BONUS set BONUSPOINT = 1000 where MEMBERID = users_id;
   ```

   where *users_id* is the value from step 4f.
   h. Enter the following to commit your database changes:

   ```
   commit;
   ```

   and press Enter to run the SQL statement.
5. In the browser, click the **Men's fashions** link to view the Men's fashions section of the store.
6. Click the link for the featured special to view the product page. The page displays the regular price, and the discounted price based upon the customer's balance of bonus points.

Part 5. Appendixes

Appendix A. Starting and stopping the WebSphere Test Environment

This section describes the steps involved to start the WebSphere Test Environment within VisualAge for Java. In general, starting this environment requires you to perform the following steps:

1. Start the persistent name server.
2. Start the EJB server.
3. Start the Servlet engine.

Each of these steps is described in the subsequent sections.

To stop the test environment, reverse the order of steps.

Note: In order to use all of the features of the WebSphere Test Environment, you should ensure that the WebSphere Application Server is not running, before starting the WebSphere Test Environment. For information on stopping the WebSphere Application Server, refer to the *WebSphere Commerce Installation Guide*.

Starting and stopping the persistent name server

The persistent name server receives lookup requests from clients for enterprise beans. All enterprise beans are registered with the persistent name server.

To start or stop the persistent name server, do the following:

1. From the **Workspace** menu in VisualAge for Java, select **Tools > WebSphere Test Environment**.
   The WebSphere Test Environment Control Center opens.
2. Click **Persistent Name Server** and then click one of the following:
   - **Start Name Server**.
     Wait for the “Server open for business” message in the Console window. Starting this server may take several minutes.
   - **Stop Name Server**.

Starting and stopping the EJB server

The EJB server is a high-level process or application that provides a run-time environment to support the execution of server applications that use enterprise beans.

To start or stop the EJB server, do the following:

1. Open the EJB server configuration window (click the EJB tab, then from the **EJB** menu, select **Open To > Server Configuration**).
2. Right-click the EJB server that you want to start or stop (for example, **EJB Server {Server 1}**) and do one of the following:
   - Select **Start Server**
     Wait for the “Server open for business” message in the Console window (you may need to select **EJB Server** in the Console to view the status for this server). Starting this may take 10 or 15 minutes, depending upon your computer.
   - Select **Stop Server**

The Console is the standard input and output device for Java programs in the IDE. To see output for a particular program, first select the program in the All Programs pane, then its output is displayed in the Output pane.

Starting and stopping the servlet engine

The servlet engine integrates the Web server and WebSphere Application Server functionality to create an environment for testing purposes.

To start or stop the servlet engine, do the following

1. From the **Workspace** menu in VisualAge for Java, select **Tools > WebSphere Test Environment**.
   The WebSphere Test Environment Control Center opens.
2. Click **Servlet Engine** and then one of the following:
   - **Start Servlet Engine**.
     When the Servlet Engine has started, the console displays the `***Servlet Engine is started ***` message.
   - **Stop Servlet Engine**.

Appendix B. Deployment details

After you have created customized code in VisualAge for Java and have tested it within the WebSphere Test Environment, you must deploy it to a WebSphere Commerce instance running outside of the WebSphere Test Environment. This WebSphere Commerce instance can run locally on your development machine, or it can be on another machine (using the same, or a different operating system).

This appendix describes the steps required for deployment of customized code to a WebSphere Commerce instance running outside of the WebSphere Test Environment. You should refer to "Code deployment" on page 187 to gain an understanding of the high-level steps of the deployment process and then refer to this appendix for the details.

Mapping to the integrated file system (iSeries)

▶ 400 This section only applies if your target WebSphere Commerce Server is running on the iSeries platform.

If your target WebSphere Commerce Server is running on the iSeries platform, then you must map a local drive on your development machine to the Integrated File System (IFS) on your iSeries server. In subsequent sections in this document, `iSeries_drive` is used to refer to this local drive that is mapped to the IFS. Additionally, `drive` is used to a local drive on your development machine (one that has not been mapped to the IFS).

JAR files for customized commands and data beans

Customized command and data bean code must be stored in a project that is separate from WebSphere Commerce code. When you have completed testing within the WebSphere Test Environment, you must create a JAR file for the project that contains the customized command and data bean code, and then place the JAR file in the appropriate directory on the target WebSphere Commerce Server.

To create a JAR file for customized commands and data beans, do the following:

1. In the Workbench in VisualAge for Java, select the **Project** tab.
2. Right-click the project that contains your customized command and data bean code and select **Export**.
   The Export SmartGuide opens.

3. Select **Jar file** and click **Next**.
4. In the **Jar file** field, enter the following:
   `drive:\WebSphere\CommerceServerDev\temp_directory\jar_file_name_1.jar`
   where `drive:\WebSphere\CommerceServerDev\temp_directory\` is a temporary directory that has enough free space for your JAR file and `jar_file_name_1.jar` is the name of your JAR file with _1 appended.
5. Select the attributes for the JAR file, as follows:

Attribute	Value
class	Checked
java	Unchecked
resource	Checked
beans	Checked
Include debug attributes in .class file	Checked

   Accept the default values for other attributes.
6. Click **Finish**.

Since the implementation JAR file created does not contain complete package naming information, you must use another packaging utility (outside of VisualAge for Java) to repackage the JAR file. To repackage this file, do the following:

1. In a command window, navigate to the directory where you stored `jar_file_name_1.jar` in the preceding steps.
2. Enter `mkdir temp1`.
3. Enter `cd temp1`.
4. Set the path as follows:
   `set PATH=%PATH%;drive:\WebSphere\WebSphereStudio4\bin;`
   where *drive* is the drive on which WebSphere Studio is installed.
5. Enter `jar xvf ../jar_file_name_1.jar`.
6. Enter `jar cvf ../jar_file_name.jar *` (note that the _1 is removed from the name).
7. Switch to the `drive:\WebSphere\CommerceDev\temp_directory` directory.
8. If you are deploying to a local WebSphere Commerce instance, copy `jar_file_name.jar` to the following directory:

```
drive:\WebSphere\AppServer\installedApps\
   WC_Enterprise_App_instanceName.ear\wcstores.war\WEB-INF\lib
```

directory. Otherwise, leave the JAR file in the temporary directory until you need to transfer all file assets to the target WebSphere Commerce Server.

Creating JAR files for new entity beans

When you create new entity beans, you must store the code in a project that is separate from all WebSphere Commerce code. In addition, it must be separate from any of your customized command and data bean code. A new entity bean must be created in an EJB group that is separate from the EJB groups that contain the WebSphere Commerce public entity beans.

Deploying new entity beans involves creating two JAR files. The first JAR file is the EJB 1.1 Export JAR, and it is created using tools that are available when the EJB tab selected. The second JAR file contains the implementation code for the entity bean, and is created from the project that contains the entity bean code. This second JAR file is created using tools that are available when the Projects tab selected in the VisualAge for Java Workbench.

Creating the EJB 1.1 Export JAR file

To create the EJB 1.1Export JAR file for new entity beans, do the following:

1. In the Workbench in VisualAge for Java, select the **EJB** tab.
2. Right-click the EJB group that contains your customized entity bean code and select **Export > EJB 1.1 JAR**.
   The Export to an EJB 1.1 JAR file SmartGuide opens.
3. In the **Jar file** field, enter the following:
   `*drive*:\WebSphere\CommerceServerDev\temp_directory\*jarFileName*_DT.jar`
   where `*drive*:\WebSphere\CommerceServerDev\temp_directory` is a temporary directory that has enough free space for your JAR file and `*jarFileName*_DT.jar` is the name of your JAR file with the suffix of _DT added on.

The JAR file has been named with the "_DT" suffix as a reminder that you must run this JAR file through the EJB Deploy Tool provided by WebSphere Application Server before deploying it into your WebSphere Commerce application.

4. Select the attributes for the JAR file, as follows:

Attribute	**Value**
class	Checked
java	Unchecked
resource	Checked

Attribute	Value
Target database	DB2 If you are deploying to a DB2 database, select one of the following: • Windows AIX Solaris Linux **DB2 for NT, V7.1** • 400 **DB2 for AS/400, V4** Oracle If you are deploying to an Oracle database, select **Oracle, V8**
Include debug attributes in .class file	Checked

Accept the default values for other attributes.

5. Click **Finish**.

The JAR file is created.

Creating the implementation JAR file

To create the implementation JAR file for new entity beans, do the following:

1. In the Workbench in VisualAge for Java, select the **Project** tab.
2. Right-click the project that contains your customized entity bean code and select **Export**.
   The Export SmartGuide opens.
3. Select **Jar file** and click **Next**.
4. In the **Jar file** field, enter the following:
   `*drive*:\WebSphere\CommerceServerDev\*temp_directory*\*jarFileName*_1.jar`
   where *drive*`:\WebSphere\CommerceServerDev\`*temp_directory* is a temporary directory that has enough free space for your JAR file and *jarFileName*`_1.jar` is the name of your JAR file with _1 appended.
5. Select the attributes for the JAR file, as follows:

Attribute	Value
class	Checked
java	Unchecked
resource	Checked
beans	Checked
Include debug attributes in .class file	Checked

Accept the default values for other attributes.

6. Click **Finish**.

Since the implementation JAR file created does not contain complete package naming information, you must use another packaging utility (outside of VisualAge for Java) to repackage the JAR file. To repackage this file, do the following:

1. In a command window, navigate to the directory where you stored `*jarFileName*_1.jar` in the preceding steps.
2. Enter `mkdir temp1`.
3. Enter `cd temp1`.
4. Set the path as follows:
   `set PATH=%PATH%;drive:\WebSphere\WebSphereStudio4\bin;`
   where *drive* is the drive on which WebSphere Studio is installed.
5. Enter `jar xvf ../jarFileName_1.jar`.
6. Enter `jar cvf ../jarFileName.jar *` (note that the `_1` is removed from the name).
7. Switch to the `drive:\WebSphere\CommerceServerDev\temp_directory` directory.
8. If you are deploying to a local WebSphere Commerce instance, copy `jarFileName_1.jar` to the following directory:
   `drive:\WebSphere\CommerceServer\temp\lib`
   Otherwise, leave the JAR file in the temporary directory until you need to transfer all file assets to the target WebSphere Commerce Server.

Creating JAR files for customized WebSphere Commerce entity beans

You are permitted to extend any of the public WebSphere Commerce entity beans. These beans are found in the following EJB groups:

- WCSActrlEJBGroup
- WCSApproval
- WCSAuction
- WCSCatalog
- WCSCommon
- WCSContract
- WCSCoupon
- WCSFulfillment
- WCSInventory
- WCSMessageExtensions
- WCSOrder
- WCSOrderManagement
- WCSOrderStatus

- WCSPayment
- WCSPVCDevices
- WCSTaxation
- WCSUserTraffic
- WCSUser
- WCSUTF

If you extend any of the public WebSphere Commerce entity beans, you must create an EJB 1.1 Export JAR file for the EJB group that contains the modified WebSphere Commerce public entity bean.

Creating the EJB 1.1 Export JAR file

To create the EJB 1.1 Export JAR file for the EJB group that contains the modified entity bean, do the following:

1. In the Workbench in VisualAge for Java, select the **EJB** tab.
2. Right-click the EJB group that contains your customized entity bean code and select **Export > EJB 1.1 JAR**.
   The Export to an EJB 1.1 JAR file SmartGuide opens.
3. In the **Jar file** field, enter the following:
   `*drive*:\WebSphere\CommerceServerDev\*temp_directory*\`
   `Cust_*EJBGroupName*-ejb_DT.jar`
   where `*drive*:\WebSphere\CommerceServerDev\*temp_directory*` is a temporary directory that has enough free space for your JAR file and `Cust_*EJBGroupName*-ejb_DT.jar` is the name of your JAR file with the suffix of _DT added on.

The JAR file has been named with the "_DT" suffix as a reminder that you must run this JAR file through the EJB Deploy Tool provided by WebSphere Application Server before deploying it into your WebSphere Commerce application.

4. Select the attributes for the JAR file, as follows:

Attribute	Value
class	Checked
java	Unchecked
resource	Checked

Attribute	Value
Target database	DB2 If you are deploying to a DB2 database, select one of the following: • Windows AIX Solaris Linux **DB2 for NT, V7.1** • 400 **DB2 for AS/400, V4** Oracle If you are deploying to an Oracle database, select **Oracle, V8**
Include debug attributes in .class file	Checked

Accept the default values for other attributes.

5. Click **Finish**.

Creating the client JAR file

To create the client JAR file, do the following:

1. With the EJB tab selected, highlight the all of the WebSphere Commerce EJB groups (the name begin with `WCS`). With all of these groups highlighted, right-click and select **Export > Client JAR**.
   The Export SmartGuide opens.
2. In the **JAR file** field, enter the following:
   drive`:\WebSphere\CommerceServerDev\`*temp_directory*`\wcsejsclient.jar`
3. Select attributes as follows:

Attribute	Value
beans	Checked
class	Checked
java	Unchecked
resource	Checked
Include debug attributes in .class file	Checked

Accept the default values for other attributes.

4. Click **Finish**.

The JAR file is created.

Storing assets on the target WebSphere Commerce Server

Assets related to your customized code must be copied to the target WebSphere Commerce Server. These assets include JAR files for customized commands, data beans and entity beans. You may also have new JSP templates and graphics that support the customization.

AIX Solaris Linux You should perform all of the deployment steps on the target WebSphere Commerce Server using the user that was created when the steps in the *"Running the postinstall script"* section of the *WebSphere Commerce Installation Guide* were performed. By default, that is the `wasuser`. In addition, ensure that your file assets (for example, JAR files) and directories into which these assets are placed have read, write and execute file permissions granted for this user.

400 Ensure that the authorities for the `/QIBM/UserData/WebCommerce/instances/`*`instanceName`* and `/QIBM/UserData/WebCommerce/instances/`*`instanceName`*`/working` directories include the user `QEJB`. Add this user to both directories, setting the Data Authority to `*RWX`.

The following table summarizes the standard directories in which assets are stored on a target WebSphere Commerce Server that is running Windows NT or Windows 2000.

Table 12.

Asset Type	Directory location on target WebSphere Commerce Server
JAR files for command and databean logic	*`drive`*`:\WebSphere\AppServer\installedApps\WC_Enterprise_App_`*`instanceName`*`.ear\wcstores.war\WEB-INF\lib`
EJB 1.1 Export JAR created using VisualAge for Java	*`drive`*`:\WebSphere\CommerceServer\temp` **Note:** This file is then passed as an input to the EJBDeploy tool to generate deployed code that can be used in WebSphere Application Server V4.0.
JAR file of EJB client code	*`drive`*`:\WebSphere\CommerceServer\temp\lib` **Note:** This file is only used in the class path for the EJBDeploy tool.
JAR file of EJB implementation code	*`drive`*`:\WebSphere\CommerceServer\temp\lib` **Note:** This file is then added into the enterprise application as a file asset.
JSP templates	*`drive`*`:\WebSphere\AppServer\installedApps\WC_Enterprise_App_`*`instanceName`*`.ear\wcstores.war\`*`storeDir`*

Table 12. (continued)

Asset Type	Directory location on target WebSphere Commerce Server
Images	`*drive*:\WebSphere\AppServer\installedApps\WC_Enterprise_App_*instanceName*.ear\wcstores.war\*storeDir*\images`

To store assets on the target WebSphere Commerce Server, do the following:

1. Locate the JAR files for your command and data bean code. These should be in one of the following directories on the development machine:
   - Windows *drive*`:\WebSphere\AppServer\installedApps\WC_Enterprise_App_`*instanceName*`.ear\wcstores.war\WEB-INF\lib`
   - Windows *drive*`:\WebSphere\CommerceServerDev\`*temp_directory*
2. Copy the JAR files for your command and data bean code one of the following directories on the target WebSphere Commerce Server:
   - Windows *drive*`:\WebSphere\AppServer\installedApps\WC_Enterprise_App_`*instanceName*`.ear\wcstores.war\WEB-INF\lib`
   - AIX `/usr/WebSphere/AppServer/installedApps/WC_Enterprise_App_`*instanceName*`.ear/wcstores.war/WEB-INF/lib`
   - Solaris `/opt/WebSphere/AppServer/installedApps/WC_Enterprise_App_`*instanceName*`.ear/wcstores.war/WEB-INF/lib`
   - Linux `/opt/WebSphere/AppServer/installedApps/WC_Enterprise_App_`*instanceName*`.ear/wcstores.war/WEB-INF/lib`
   - 400 `/QIBM/UserData/WebASAdv4/`*WAS_AdminInstanceName*`/installedApps/WC_Enterprise_App_`*instanceName*`.ear/wcstores.war/WEB-INF/lib`
3. Locate the EJB 1.1 Export JAR files for any new EJB groups as well as any modified WebSphere Commerce entity beans in the following directory on the development machine:
   Windows *drive*`:\WebSphere\CommerceServerDev\`*temp_directory*
4. Copy the EJB 1.1 Export JAR files into one of the following directories on the target WebSphere Commerce Server:
   - Windows *drive*`:\WebSphere\CommerceServer\temp`
   - AIX `/usr/WebSphere/CommerceServer/temp`
   - Solaris `/opt/WebSphere/CommerceServer/temp`
   - Linux `/opt/WebSphere/CommerceServer/temp`
   - 400 `/QIBM/UserData/WebCommerce/instances/`*instanceName*`/temp`

5. If you have created new enterprise beans, locate the implementation JAR file for these beans in the following directory on the development machine:
   Windows `drive:\WebSphere\CommerceServerDev\temp_directory`
6. Copy the implementation JAR file for your new enterprise beans into the following directory on the target WebSphere Commerce Server:
   - Windows *drive*`:\WebSphere\CommerceServer\temp\lib`
   - AIX `/usr/WebSphere/CommerceServer/temp/lib`
   - Solaris `/opt/WebSphere/CommerceServer/temp/lib`
   - Linux `/opt/WebSphere/CommerceServer/temp/lib`
   - 400 `/QIBM/UserData/WebCommerce/instances/`*instanceName*`/temp/lib`
7. If you have modified existing WebSphere Commerce entity beans, locate the client JAR file in the following directory on the development machine:
   Windows *drive*`:\WebSphere\CommerceServerDev\`*temp_directory*
8. Copy the client JAR file to one of the following directories on the target WebSphere Commerce Server:
   - Windows *drive*`:\WebSphere\CommerceServer\temp\lib`
   - AIX `/usr/WebSphere/CommerceServer/temp/lib`
   - Solaris `/opt/WebSphere/CommerceServer/temp/lib`
   - Linux `/opt/WebSphere/CommerceServer/temp/lib`
   - 400 `/QIBM/UserData/WebCommerce/instances/`*instanceName*`/temp/lib`
9. Copy any JSP templates into the following directories on the target WebSphere Commerce Server:
   - Windows *drive*`:\WebSphere\AppServer\installedApps\WC_Enterprise_App_`*instanceName*`.ear\wcstores.war\`*store_directory*
   - AIX `/usr/WebSphere/AppServer/installedApps/WC_Enterprise_App_`*instanceName*`.ear/wcstores.war/`*store_directory*
   - Solaris `/opt/WebSphere/AppServer/installedApps/WC_Enterprise_App_`*instanceName*`.ear/wcstores.war/`*store_directory*
   - Linux `/opt/WebSphere/AppServer/installedApps/WC_Enterprise_App_`*instanceName*`.ear/wcstores.war/`*store_directory*

- 400 /QIBM/UserData/WebASAdv4/*WAS_AdminInstanceName*/installedApps/WC_Enterprise_App_*instanceName*.ear/wcstores.war/*store_directory*

  where *store_directory* is the directory for your store.
10. Copy any images into the following directories on the target WebSphere Commerce Server:
    - Windows *drive*:\WebSphere\AppServer\installedApps\WC_Enterprise_App_*instanceName*.ear\wcstores.war\store_directory\images
    - AIX /usr/WebSphere/AppServer/installedApps/WC_Enterprise_App_*instanceName*.ear/wcstores.war/*store_directory*/images
    - Solaris /opt/WebSphere/AppServer/installedApps/WC_Enterprise_App_*instanceName*.ear/wcstores.war/*store_directory*/images
    - Linux /opt/WebSphere/AppServer/installedApps/WC_Enterprise_App_*instanceName*.ear/wcstores.war/*store_directory*/images
    - 400 /QIBM/UserData/WebASAdv4/*WAS_AdminInstanceName*/installedApps/WC_Enterprise_App_*instanceName*.ear/wcstores.war/*store_directory*/images

    where *store_directory* is the directory for your store.

Updating the target database

When your target WebSphere Commerce Server uses a different database than your development machine, you must perform all updates that were done to the development database on the database that is used by the target WebSphere Commerce Server. This includes any updates for the registration of new or modified commnads, additional tables that have been created and the creation of access control policies for any new resources that have been created.

For information about loading access control policies (including command syntax for various platforms and directory permission requirements), refer to the *WebSphere Commerce Access Control Guide*.

400 You are reponsible for having a utility for running SQL statements. One way to do this is to use Client Access Express V5R1 (full installation). To open this utility, do the following:

1. Open the **Operations Navigator**.

2. After the operations navigator opens, you are required to sign onto a particular system. Ensure that you select the target iSeries machine and use the WebSphere Commerce instance user profile and password. This ensures that the WebSphere Commerce instance user profile owns all new tables that are created.
3. Click on your system in the panel on the left, then right-click on **DATABASE** and select **Run SQL Scripts** from the drop-down list.
   The Run SQL Scripts window opens. Using this window, you can cut and paste in SQL statements or open a SQL script. You can set your default schema by using the Connection/JDBC setup options.

Generating deployed code

This section describes how to use the EJB Deploy Tool to generate the deployed code for the enterprise beans contained in the EJB 1.1 Export JAR file. This tool is provided by WebSphere Application Server.

To generate the deployed code, do the following:

1. At a command prompt on the target WebSphere Commerce Server, navigate to the following directory:
   - Windows `drive:\WebSphere\CommerceServer\temp`
   - AIX `/usr/WebSphere/CommerceServer/temp`
   - Solaris `/opt/WebSphere/CommerceServer/temp`
   - Linux `/opt/WebSphere/CommerceServer/temp`
2. 400 At a command prompt on the target WebSphere Commerce Server, enter the following commands:
   ```
   STRQSH
   cd /QIBM/UserData/WebCommerce/instances/instanceName/temp
   ```
3. Temporarily add the tool to the system path by entering the following command:
   - Windows `PATH=drive:\WebSphere\AppServer\deploytool;%PATH%`
   - AIX `PATH=/usr/WebSphere/AppServer/deploytool:$PATH`
   - Solaris `PATH=/opt/WebSphere/AppServer/deploytool:$PATH`
   - Linux `PATH=/opt/WebSphere/AppServer/deploytool:$PATH`
   - 400 `PATH=/QIBM/ProdData/WebASAdv4/bin:$PATH`

     ```
     CP=ClassPathOfDepJARFiles
     ```

   where *ClassPathOfDepJARFiles* is the class path of any dependent JAR files. Note that if you have modified an existing WebSphere Commerce enterprise bean you must include the `wcsejsclient.jar`, `wcsejbimpl.jar`, and `xml4j.jar` files in the class path of dependent JAR files. As such, the following provides an example class path for a case where an existing WebSphere Commerce enterprise bean has been modified:

```
CP=/QIBM/UserData/WebCommerce/instances/instanceName/temp/lib/
wcsejsclient.jar:
/QIBM/UserData/WebASAdv4/WAS_AdminInstanceName/installedApps/
WC_Enterprise_App_instanceName.ear/lib/wcsejbimpl.jar:
/QIBM/UserData/WebASAdv4/WAS_AdminInstanceName/installedApps/
WC_Enterprise_App_instanceName.ear/lib/xml4j.jar
```

Note: The line breaks in the preceding class path example are for display purposes only. You should enter class path information on a single line.

4. Solaris Linux In order to avoid exceding the character limit in the command line interface, a script is used to invoke the `ejbdeploy` command. Do the following to create this script and invoke the command:
   a. Navigate to the following directory:
   ```
   /opt/WebSphere/AppServer/bin
   ```
   b. Create a new file and name it appropriately for the script. For example, name the file `myejbd.sh`.
   c. Include the following information in the `myejbd.sh` file:
   ```
   #!/bin/sh
   ./ejbdeploy.sh EJBGroupJARFile WorkingDir OutputJARFile
   -nowarn -keep -35 -cp ClassPathOfDepJARFiles
   ```
   where the variables have the same definitions as those in step 5(note, that you do not perform that step). The following is an example of what to include in the script file, with values for the variables included:
   ```
   #!/bin/sh
   ./ejbdeploy.sh
   /opt/WebSphere/CommerceServer/temp/CustomizedWCSUserDeployed_DT.jar
   . /opt/WebSphere/CommerceServer/temp/CustomizedWCSUserDeployed.jar
   -nowarn -keep -35 -cp "/opt/WebSphere/CommerceServer/temp/lib/
   wcsejsclient.jar:/opt/WebSphere/AppServer/installedApps/
   WC_Enterprise_App_demo.ear/lib/wcsejbimpl.jar:/opt/WebSphere/
   AppServer/installedApps/WC_Enterprise_App_demo.ear/lib/xml4j.jar"
   ```
   Note: All line breaks after the `./ejbdeploy.sh` command are for display purposes only. You should not have breaks after the `./ejbdeploy.sh` command in your file.

   Save the script and make it executable.
   d. Invoke the script by entering the following:
   ```
   ./myejbd.sh
   ```

5. Windows AIX 400 Enter the ejbdeploy command as follows:

Windows 400

```
ejbdeploy EJBGroupJARFile_DT WorkingDir OutputJARFile -nowarn -keep -35 -cp
   ClassPathOfDepJARFiles
```

AIX

```
/usr/WebSphere/AppServer/bin/ejbdeploy.sh EJBGroupJARFile_DT WorkingDir
OutputJARFile -nowarn -keep -35
-cp ClassPathOfDepJARFiles
```

where:

- `EJBGroupJARFile_DT` is the name the JAR file for your EJB group. For example, if you modified a bean in the WCSUser EJB group, an example usage on a Windows-based platform is

  `drive:\WebSphere\CommerceServer\temp\CustomizedWCSUser_DT.jar`

  400 An example usage for the iSeries platform is

  ```
  /QIBM/UserData/WebCommerce/instances/instanceName/temp/
  CustomizedWCSUser_DT.jar
  ```
- `WorkingDir` the name of the directory where temporary files that are required for code generation are stored.
- `OutputJARFile` is the fully qualified name of the output JAR file. For example, you might enter `CustomizedWCSUserDeployed.jar`.
- `-nowarn` is an optional parameter to suppress warning and information messages.
- `-keep` is an optional parameter to retain the working directory after the ejbdeploy command has run.
- `-35` is a mandatory parameter that will use the same top-down mapping rules for CMP entity beans that are used in the EJB Deploy Tool that was provided with the WebSphere Application Server, Version 3.5.
- `-cp ClassPathOfDepJARFiles` is the class path of any dependent JAR files. When you have modified an existing WebSphere Commerce enterprise bean you must include the `wcsejsclient.jar`, `wcsejbimpl.jar`, and `xml4j.jar` files in the class path of dependent JAR files. As such, the following provides an example class path for a case where an existing WebSphere Commerce enterprise bean has been modified:

  ```
  "drive:\WebSphere\CommerceServer\temp\lib\wcsejsclient.jar;
  drive:\WebSphere\AppServer\installedApps\
  WC_Enterprise_App_instanceName.ear \lib\wcsejbimpl.jar;
  drive:\WebSphere\AppServer\installedApps\
  WC_Enterprise_App_instanceName.ear\lib\xml4j.jar;"
  ```

Note: 400 For the class path values, enter `-cp $CP` where `CP` has been previously defined with the appropriate class path entries. Additionally, the quotation marks are not required.

Modifying transaction isolation level of entity beans

This section describes how to use the `modifyIsolationLevel` command line utilty to set the transaction isolation level of your entity beans to the appropriate level for your database type.

To run the `modifyIsolationLevel` command, do the following:

1. On the target WebSphere Commerce Server, use a command prompt to navigate to the following directory:
   - Windows `*drive*:\WebSphere\CommerceServer\bin`
   - AIX `/usr/WebSphere/CommerceServer/bin`
   - Solaris `/opt/WebSphere/CommerceServer/bin`
   - Linux `/opt/WebSphere/CommerceServer/bin`
2. 400 Enter the following commands:
   ```
   STRQSH
   cd /QIBM/ProdData/WebCommerce/bin
   ```
3. You must issue the `modifyIsolationLevel` command, which has the following general syntax:

   Windows AIX 400
   ```
   modifyIsolationLevel -jarFile jar_file_name.jar
      -logFile log_file_name -dbType db_type
   ```
   Solaris Linux
   ```
   ./modifyIsolationLevel.sh -jarFile jar_file_name.jar
      -logFile log_file_name -dbType db_type
   ```
   where
   - *`jar_file_name.jar`* is the name of the JAR file that contains the customized code
   - *`log_file_name`* is the fully qualified file name where information should be logged
   - *`db_type`* is the type of database you are using. Enter either DB2 or ORACLE

   The following is an example of the `modifyIsolationLevel` command with all values specified for usage on a Windows platform:

```
modifyIsolationLevel -jarFile
   D:\WebSphere\CommerceServer\temp\CustomizedWCSUserDeployed.jar
   -logFile D:\WebSphere\CommerceServer\instances\demo\logs\output.log
   -dbType DB2
```

400 The following is an example of the `modifyIsolationLevel` command with all values specified for usage on the iSeries:

```
modifyIsolationLevel -jarFile
   /QIBM/UserData/WebCommerce/instances/instanceName/temp/
   CustomizedWCSUserDeployed.jar -logFile /QIBM/UserData/WebCommerce/
   instances/instanceName/logs/output.log -dbType DB2
```

Note that in the preceding examples, the line breaks are for display purposes only.

Note: The parameter names are case sensitive. That is, `jarFile` is not the same as `jarfile`. Ensure that you enter the parameter names correctly.

The command has run successfully if no exceptions are displayed in the command window. After completion, note that the timestamp on your deployed JAR file has changed.

Exporting the current WebSphere Commerce enterprise application

This section describes how to use the WebSphere Application Server Administration Console to export the current WebSphere Commerce enterprise application.

To export the current enterprise application from the WebSphere Application Server, do the following:

1. Ensure that you have the following directory on the target WebSphere Commerce Server:
   - Windows *drive*`:\WebSphere\CommerceServer\working`
   - AIX `/usr/WebSphere/CommerceServer/working`
   - Solaris `/opt/WebSphere/CommerceServer/working`
   - Linux `/opt/WebSphere/CommerceServer/working`
   - 400 `/QIBM/UserData/WebCommerce/instances/`*instanceName*`/working`

   If you do not already have this directory, create it now.

   Note: AIX Solaris Linux Ensure that the permission for the `/working` directory is set to the user that was created when the steps in the *"Running the postinstall script"* section of the *WebSphere Commerce Installation Guide* were performed.

400 Ensure that the authorities for the `/QIBM/UserData/WebCommerce/instances/`*`instanceName`* and `/QIBM/UserData/WebCommerce/instances/`*`instanceName`*`/working` directories include the user `QEJB`. Add this user to both directories, setting the Data Authority to `*RWX`.

2. Open the WebSphere Application Server Administration Console.
3. Expand **WebSphere Administrative Domain**.
4. Expand **Enterprise Application**.
5. Right-click your WebSphere Commerce application. For example, expand the **demo** application and select **Export Application**.
6. In the **Export directory** field, enter the following:
   - Windows *`drive`*`:\WebSphere\CommerceServer\working`
   - AIX `/usr/WebSphere/CommerceServer/working`
   - Solaris `/opt/WebSphere/CommerceServer/working`
   - Linux `/opt/WebSphere/CommerceServer/working`
   - 400 `/QIBM/UserData/WebCommerce/instances/`*`instanceName`*`/working`

   This exports the whole application, including all resources into the `WC_Enterprise_App_`*`instanceName`*`.ear` file (where *`instanceName`* is the name of your WebSphere Commerce instance).

Exporting configuration information for enterprise beans

This section describes how to use the `-export` option of the `XMLConfig` command line utility to export the configuration information for the enterprise beans that are contained in the existing enterprise application.

After the information for the beans in the existing application has been exported, you must manually add information for any new beans that you are adding to the application.

To export this configuration information, do the following:

1. Copy the `was.export.app.xml` file from the following directory on the target WebSphere Commerce Server:
   - Windows *`drive`*`:\WebSphere\CommerceServer\xml\config`
   - AIX `/usr/WebSphere/CommerceServer/xml/config`
   - Solaris `/opt/WebSphere/CommerceServer/xml/config`
   - Linux `/opt/WebSphere/CommerceServer/xml/config`
   - 400 `/QIBM/ProdData/WebCommerce/xml/config`

into the following directory on the target WebSphere Commerce Server:

- Windows *drive*:\WebSphere\CommerceServer\working
- AIX /usr/WebSphere/CommerceServer/working
- Solaris /opt/WebSphere/CommerceServer/working
- Linux /opt/WebSphere/CommerceServer/working
- 400 /QIBM/UserData/WebCommerce/instances/*instanceName*/working

  where
  – *instanceName* is the name of your WebSphere Commerce instance.

2. Windows AIX Solaris Linux Open the was.export.app.xml file in a text editor. In this file replace all occurances of $Enterprise_Application_Name$ with

   ```
   WebSphere Commerce Enterprise Application - instanceName
   ```

   where *instanceName* is the name of your WebSphere Commerce instance (for example, demo). Save this file.

   Note: The value that you are inserting must match the information for your instance that is displayed in the WebSphere Application Server Administration Console.

3. 400 Open the was.export.app.xml file in a text editor. In this file replace all occurances of $Enterprise_Application_Name$ with

   ```
   instanceName - WebSphere Commerce Enterprise Application
   ```

   where *instanceName* is the name of your WebSphere Commerce instance (for example, demo). Save this file.

   Note: The value that you are inserting must match the information for your instance that is displayed in the WebSphere Application Server Administration Console.

4. Windows AIX Solaris Linux At a command prompt, navigate to the following directory:
   - Windows *drive*:\WebSphere\CommerceServer\working
   - AIX /usr/WebSphere/CommerceServer/working
   - Solaris /opt/WebSphere/CommerceServer/working
   - Linux /opt/WebSphere/CommerceServer/working

5. 400 At a command prompt, enter the following:

```
STRQSH
PATH=/QIBM/ProdData/WebASAdv4/bin:$PATH
cd /QIBM/UserData/WebCommerce/instances/instanceName/working
```

6. Windows AIX Solaris Linux Invoke the XMLConfig tool to perform a partial export by entering the following command:

   Windows

   ```
   xmlConfig -export OutputFile.xml -partial was.export.app.xml
      -adminNodeName wasHostName
   ```

   AIX

   ```
   /usr/WebSphere/AppServer/bin/XMLConfig.sh  -export OutputFile.xml
      -partial was.export.app.xml -adminNodeName wasHostName
      -nameServiceHost wasHostName -nameServicePort wasAdminPort
   ```

   Solaris Linux

   ```
   /opt/WebSphere/AppServer/bin/XMLConfig.sh  -export OutputFile.xml
      -partial was.export.app.xml -adminNodeName wasHostName
      -nameServiceHost wasHostName -nameServicePort wasAdminPort
   ```

   where

   - *wasHostName* is the name of the node in the WebSphere Application Server that contains your current enterprise application.
   - `OutputFile.xml` is the name of the file that is created as a result of running this command and `was.export.app.xml` is file that you modified in step 2.
   - *wasAdminPort* is the WebSphere Application Server administration port.

7. 400 Invoke the XMLConfig tool to perform a partial export by entering the following command:

   ```
   xmlConfig -export OutputFile.xml -partial was.export.app.xml
   -adminNodeName wasHostName -nameServiceHost wasHostName
   -nameServicePort wasAdminPort -instance wasInstanceName
   ```

   where

   - *wasHostName* is the name of the node in the WebSphere Application Server that contains your current enterprise application.

     Note: The value for *wasHostName* is case-sensitive and must match the value that is in the TCP/IP configuration. (Use a command-line processor to access CFGTCP, option 12 to verify host name.)
   - `OutputFile.xml` is the name of the file that is created as a result of running this command and `was.export.app.xml` is file that you modified in step 3.

- *wasAdminPort* is the WebSphere Application Server administration port.
- *wasInstanceName* is the WebSphere Application Server instance name.

After you have exported the configuration information for each of the beans contained in the current enterprise application, you must add in a new stanza that describes each enterprise bean that you will add to your application. For example, if you had a new entity bean called "Bonus", you must add a stanza that describes this Bonus bean. In addition, you must replace a variable in the configuration file to specify the exact name of the .ear file for your enterprise application.

To make these updates to the `OutputFile.xml` file, do the following:

1. Navigate to the following directory on the target WebSphere Commerce Server:
   - Windows `*drive*:\WebSphere\CommerceServer\working`
   - AIX `/usr/WebSphere/CommerceServer/working`
   - Solaris `/opt/WebSphere/CommerceServer/working`
   - Linux `/opt/WebSphere/CommerceServer/working`
   - 400 `/QIBM/UserData/WebCommerce/instances/*instanceName*/working`
2. Open the `OutputFile.xml` file in a text editor.
3. Locate the `<ear-file-name>` tag and replace the value with the following:
   - Windows `*drive*:\WebSphere\CommerceServer\working\WC_Enterprise_App_*instanceName*.ear`
   - AIX `/usr/WebSphere/CommerceServer/working/WC_Enterprise_App_*instanceName*.ear`
   - Solaris `/opt/WebSphere/CommerceServer/working/WC_Enterprise_App_*instanceName*.ear`
   - Linux `/opt/WebSphere/CommerceServer/working/WC_Enterprise_App_*instanceName*.ear`
   - 400 `/QIBM/UserData/WebCommerce/instances/*instanceName*/working/WC_Enterprise_App_*instanceName*.ear`
4. You must also add in a new stanza for the each new bean that you are adding to the enterprise application. The following example shows how to add a new bean, called "Bonus".

   Windows AIX Solaris Linux

```
<ejb-module name="yourEJBGroup">
   <jar-file>yourDeployedJarFile.jar</jar-file>
   <module-install-info>
```

```
      <application-server-full-name>/NodeHome:$hostName$/EJBServerHome:
         WebSphere Commerce Server - instanceName/
      </application-server-full-name>
   </module-install-info>
   <ejb-module-binding>
      <data-source>
         <jndi-name>jdbc/WebSphere Commerce DB2 DataSource instanceName
            </jndi-name>
         <default-user>user</default-user>
         <default-password>password</default-password>
      </data-source>
      <enterprise-bean-binding name="BeanBindingName">
         <jndi-name>instanceNameJNDINameOfBean</jndi-name>
      </enterprise-bean-binding>
   </ejb-module-binding>
</ejb-module>
```

400

```
<ejb-module name="yourEJBGroup">
   <jar-file>yourDeployedJarFile.jar</jar-file>
   <module-install-info>
      <application-server-full-name>/NodeHome:$hostName$/EJBServerHome:
         instanceName - WebSphere Commerce Server/
      </application-server-full-name>
   </module-install-info>
   <ejb-module-binding>
      <data-source>
         <jndi-name>jdbc/instanceName WebSphere Commerce DB2 DataSource
            </jndi-name>
         <default-user>user</default-user>
         <default-password>password</default-password>
      </data-source>
      <enterprise-bean-binding name="BeanBindingName">
         <jndi-name>instanceNameJNDINameOfBean</jndi-name>
      </enterprise-bean-binding>
   </ejb-module-binding>
</ejb-module>
```

where

- *yourEJBGroup* is the name of the EJB group that contains the bean that you are adding to the enterprise application.
- *yourDeployedJarFile* is the name of the JAR file that contains the deployed code for your EJB group.
- *instanceName* is the name of your WebSphere Commerce instance to which you are deploying the code.
- *user* is your database user name.
- *password* is the password for your database user.
- *BeanBindingName* is the binding name for your enterprise bean. For example, for bean named Bonus, this is `Bonus_Binding`.

- *`instanceNameJNDINameOfBean`* is the JNDI name of the enterprise bean with the WebSphere Commerce instance prepended. This JNDI name must exactly match that of the enterprise bean. An example value is `democom/ibm/commerce/sample/objects/Bonus`. In this example, "`demo`" is the instance name and "`com/ibm/commerce/sample/objects/Bonus`" is the JNDI name of the enterprise bean. This value must be entered on a single line.
  You can verify the JNDI name using either VisualAge for Java or the Application Assembly Tool.
  To verify the JNDI name using VisualAge for Java, do the following:
  a. Right-click the bean and select **Properties**.
     The JNDI name is displayed.

The JNDI name can be verified using the Application Assembly Tool, however, this requires that you have already assembled the new enterprise bean into the enterprise application. The Application Assembly Tool is accessed from the Tools menu in the WebSphere Application Server Administration Console.

To verify the JNDI name using the Application Assembly Tool, do the following:

a. Open the .ear file that contains the bean.
b. Expand the EJB module that contains the bean.
c. Select the bean.
d. Click the **Bindings** tab.
   The JNDI name is displayed.

Note that using either of these methods shows the JNDI name, you must still prepend the name of the WebSphere Commerce instance when adding the information to the XML file.

Notes:

a. The line breaks in the preceding stanzas are for display purposes only.
b. Ensure that the **`$hostName$`** value matches the current admininstration node server name. In addition, ensure that there is no carriage return character in this line.
c. The `<application-server-full-name>` specification cannot span more than one line.
d. If you are using an Oracle database, you must modify the datasource information. Change the following line taken from the preceding code snip:

```
<jndi-name>jdbc/WebSphere Commerce DB2 DataSource instanceName
   </jndi-name>
```

   to the following:

```
<jndi-name>jdbc/WebSphere Commerce Oracle DataSource instanceName
   </jndi-name>
```
5. If you are deploying modified WebSphere Commerce entity beans, you must update the stanzas for those beans to reflect the names of JAR files that contain the deployed code for the modified beans.
6. Save the `OutputFile.xml` file.

Assembling new enterprise beans into an enterprise application

This section describes how to use the Application Assembly Tool to assemble new enterprise beans into an existing enterprise application.

400 Since the Application Assembly Tool runs on the Windows platform, you must reference the drive that you have mapped to your iSeries IFS when prompted for fully-qualified path names. This is referred to as the *`iSeries_drive`*.

AIX Solaris Linux It is advisable to increase the memory heap size value in the `assembly.sh` file to avoid running out of memory when saving new or modified `.ear` files.

This file is located in the following directory:

- AIX `/usr/WebSphere/AppServer/bin`
- Solaris `/opt/WebSphere/AppServer/bin`
- Linux `/opt/WebSphere/AppServer/bin`

To increase the memory heap size, modify the following line:

```
$JAVA_HOME/jre/bin/java
```

so that it appears as follows:

```
$JAVA_HOME/jre/bin/java -mx512M
```

In this step, you open the `.ear` file your enterprise application that was created in section Exporting the current WebSphere Commerce enterprise application in the application assembler tool. Once it is open inside that tool, perform the following tasks to add the new entity bean to the enterprise application:

1. Import the EJB group containing the new entity bean. The JAR file for the new EJB group will be stored within the EJB Module section of the enterprise application.
2. Set the class path for the new entity bean to include the implementation JAR file.

3. Add the implementation JAR file to the application. This JAR file will be stored within the Files section of the enterprise application.
4. Set up WebSphere Application Server security for methods contained in the new entity bean.

To assemble the new EJB group into the enterprise application, do the following:

1. Windows AIX Solaris Linux Backup the current enterprise application, by doing the following on the target WebSphere Commerce Server:
   a. At a command prompt, navigate to the following directory:
      - Windows `*drive*:\WebSphere\CommerceServer\working`
      - AIX `/usr/WebSphere/CommerceServer/working`
      - Solaris `/opt/WebSphere/CommerceServer/working`
      - Linux `/opt/WebSphere/CommerceServer/working`
   b. Make a copy of the existing WC_Enterprise_App_*instanceName*.ear file and name it WC_Enterprise_App_*instanceName*.ear.bak.
2. 400 Backup the current enterprise application, by doing the following:
   a. At a command prompt, enter the following:
      ```
      STRQSH
      cd /QIBM/UserData/WebCommerce/instances/instanceName/working
      cp WC_Enterprise_App_instanceName.ear
      WC_Enterprise_App_instanceName.ear.bak
      ```
   b. To avoid unnecessarily long waiting times caused by the transfer of data between your local client machine and the iSeries machine running WebSphere Application Server, create the following directory on your local client machine and then copy the WC_Enterprise_App_*instanceName*.ear file to this directory:
      drive:\WebSphere\CommerceServer\working
      where *drive* is a local drive.

      Note: If the *drive*:\WebSphere\CommerceServer\working directory does not exist on your local machine, create it now.
3. Open the WebSphere Application Server Administration Console.
4. From the **Tools** menu, select **Application Assembly Tool**.
5. If a Welcome window opens, select **Cancel** to close that window.
6. Open the enterprise application upon which you are going to work, by doing the following:
   a. From the **File** menu, select **Open**.

b. In the **File name** field, enter:
   - Windows *drive*:\WebSphere\CommerceServer\working\WC_Enterprise_App_*instanceName*.ear
   - AIX /usr/WebSphere/CommerceServer/working/WC_Enterprise_App_*instanceName*.ear
   - Solaris /opt/WebSphere/CommerceServer/working/WC_Enterprise_App_*instanceName*.ear
   - Linux /opt/WebSphere/CommerceServer/working/WC_Enterprise_App_*instanceName*.ear
   - 400 *drive*:\WebSphere\CommerceServer\working\WC_Enterprise_App_*instanceName*.ear

   and click **Open**. Wait for the application to open before continuing to the next steps. This takes several minutes.
7. Right-click **EJB Modules** and select **Import**.
8. In the **File name** field, enter the following:
   - Windows *drive*:\WebSphere\CommerceServer\temp\*yourDeployedJarFile*.jar
   - AIX /usr/WebSphere/CommerceServer/temp/*yourDeployedJarFile*.jar
   - Solaris /opt/WebSphere/CommerceServer/temp/*yourDeployedJarFile*.jar
   - Linux /opt/WebSphere/CommerceServer/temp/*yourDeployedJarFile*.jar
   - 400 *iSeries_drive*:\QIBM\UserData\WebCommerce\instances\instanceName\temp*yourDeployedJarFile*.jar

   where
   - *yourDeployedJarFile* is the name of the JAR file containing the deployed code for your EJB group
   - *iSeries_drive* is the local drive that is mapped to the iSeries IFS.

   Click **Open**, then in the Confirm Values window, click **OK**.
9. Once the *yourDeployedJarFile*.jar file is imported, scroll to the *yourEJBGroup* EJB group (where *yourEJBGroup* is the name of your EJB group) and select this group.
   Information about this group is shown in the pane on the right.
10. In the classpath field for the new enterprise bean, enter any dependent JAR files. For example, you might enter the corresponding

implementation JAR file and the implementation JAR file for the WebSphere Commerce entity beans, as shown below:

`lib/`*`yourImplJarFile`*`.jar lib/wcsejbimpl.jar`

11. Click **Apply**.
12. Add the implementation JAR file for your EJB group to the application, by doing the following:
    a. Right-click the **Files** node for your enterprise application and select **Add Files**. (The **Files** node for the enterprise application is located near the bottom of the hierarchical tree. Note that there are other Files nodes for components within the enterprise application, but you must select the Files node for the whole application.)
    b. In the Add Files window, click **Browse**.
    c. Navigate to the following directory:
       - Windows *`drive`*`:\WebSphere\CommerceServer\temp`
       - AIX `/usr/WebSphere/CommerceServer/temp`
       - Solaris `/opt/WebSphere/CommerceServer/temp`
       - Linux `/opt/WebSphere/CommerceServer/temp`
       - 400 *`iSeries_drive`*`:\QIBM\UserData\WebCommerce\instances\`*`instanceName`*`\temp`
    d. With this directory highlighted, click **Select**.
    e. Return to the Add Files window. Notice that the contents of the temporary directory are displayed. Highlight the `lib` directory. The contents of the `lib` directory are displayed in the pane on the right.
    f. In the pane on the right, select the *`yourImplJarFile`* file and click **Add**. The file is then shown in the Selected Files pane.
    g. Click **OK**.
13. Configure security for your entity bean, by doing the following:
    a. With the EJB Modules node expanded, locate and expand the *YourEJBGroup* node.
    b. Expand **Entity Beans**.
    c. Expand *yourEntityBean* where *`yourEntityBean`* is the name of your entity bean.
    d. Click **Method Extensions**, then in the pane on the right do the following:
       1) Click the **Advanced** tab.
       2) Ensure that **Security identity** is selected.
       3) For each method, ensure that **Use identity of EJB server** is selected.

        4) Click **Apply** (if you have made any modifications).
    e. In the left navigation pane, right-click **Security Roles** under the *YourEJBGroup* EJB group and select **New**, then do the following:
        1) In the **Name** field, enter `WCSecurityRole` and click **Apply**. Note, if this role exists already, you do not need to perform this step.
    f. In the left naigation pane, right-click **Method Permissions** under the *`YourEJBGroup`* EJB group and select **New**, then do the following:
        1) In the **Method Permission Name** field, enter `WCMethodPermission`
        2) In the **Methods** selection area, click **Add**.
           The Add Methods window opens.
        3) Expand *yourDeployedJarFile*.**jar**, then **Bonus** and then expand each of the **Home** and **Remote** lists of methods.
        4) Hold the Shift key and select all of the home methods and click **OK**.
        5) Repeat the method selection process to add the remote methods as well (if remote methods exist).
        6) In the Roles selection area, click **Add**, select the `WCSecurityRole` and click **OK**.
        7) Click **Apply**.

14. From the **File** menu, select **Save**.
15. Close the Application Assembly Tool.
16. ▶ 400 Copy the newly modified `WC_Enterprise_App_`*`instanceName`*`.ear` file from the local machine to the iSeries machine running WebSphere Application Server. That is, copy the file from the following directory:

    `drive``:\WebSphere\CommerceServer\working`

    into the following directory:

    `iSeries_drive``:\QIBM\UserData\WebCommerce\instances\`*`instanceName`*`\working`

    where *`iSeries_drive`* is the drive letter that you have mapped to your iSeries IFS.

After this step has completed, you have created a new enterprise application that contains all of the previous logic as well as your new business logic. This is all contained in the newly modified `WC_Enterprise_App_`*`instanceName`*`.ear` file.

Assembling modified enterprise beans into an enterprise application

This section describes how to use the Application Assembly Tool to assemble modified WebSphere Commerce enterprise beans into an enterprise application.

In this step, you open your enterprise application in the application assembler tool. Once it is open inside that tool, you can do the following to include a modified WebSphere Commerce enterprise bean to the enterprise application:

1. Make a copy of the class path for the existing version of the EJB group that you have modified.
2. Remove the existing version of the EJB group that you have modified.
3. Import the new version of the EJB group that you have modified. The JAR file for the new EJB group is stored within the EJB Module section of the enterprise application.
4. Set the class path for the modified EJB group.
5. Set up WebSphere Application Server security for methods contained in the modfied entity bean.

AIX Solaris Linux It is advisable to increase the memory heap size value in the `assembly.sh` file to avoid running out of memory when saving new or modified `.ear` files.

This file is located in the following directory:

- AIX `/usr/WebSphere/AppServer/bin`
- Solaris `/opt/WebSphere/AppServer/bin`
- Linux `/opt/WebSphere/AppServer/bin`

To increase the memory heap size, modify the following line:

```
$JAVA_HOME/jre/bin/java
```

so that it appears as follows:

```
$JAVA_HOME/jre/bin/java -mx512M
```

To assemble the modified EJB group into the enterprise application, do the following:

1. Windows AIX Solaris Linux Backup the current enterprise application, by doing the following on the target WebSphere Commerce Server:
   a. At a command prompt, navigate to the following directory:
      - Windows *drive*`:\WebSphere\CommerceServer\working`
      - AIX `/usr/WebSphere/CommerceServer/working`
      - Solaris `/opt/WebSphere/CommerceServer/working`
      - Linux `/opt/WebSphere/CommerceServer/working`

   b. Make a copy of the existing `WC_Enterprise_App_`*`instanceName`*`.ear` file and name it `WC_Enterprise_App_`*`instanceName`*`.ear.bak`.

2. 400 Backup the current enterprise application, by doing the following:

   a. At a command prompt, enter the following:

   ```
   STRQSH
   cd /QIBM/UserData/WebCommerce/instances/instanceName/working
   cp WC_Enterprise_App_instanceName.ear
   WC_Enterprise_App_instanceName.ear.bak
   ```

   b. To avoid unnecessarily long waiting times caused by the transfer of data between your local client machine and the iSeries machine running WebSphere Application Server, create the following directory on your local client machine and then copy the `WC_Enterprise_App_`*`instanceName`*`.ear` file to this directory: *`drive`*`:\WebSphere\CommerceServer\working` where *`drive`* is a local drive.

   Note: If the *`drive`*`:\WebSphere\CommerceServer\working` directory does not exist on your local machine, create it now.

3. Open the WebSphere Application Server Administration Console.
4. From the **Tools** menu, select **Application Assembly Tool**.
5. Open the enterprise application upon which you are going to work by doing the following:

   a. From the **File** menu, select **Open**.

   b. In the **File name** field, enter:

      - Windows *`drive`*`:\WebSphere\CommerceServer\working\` `WC_Enterprise_App_`*`instanceName`*`.ear`
      - AIX `/usr/WebSphere/CommerceServer/working/` `WC_Enterprise_App_`*`instanceName`*`.ear`
      - Solaris `/opt/WebSphere/CommerceServer/working/` `WC_Enterprise_App_`*`instanceName`*`.ear`
      - Linux `/opt/WebSphere/CommerceServer/working/` `WC_Enterprise_App_`*`instanceName`*`.ear`
      - 400 *`drive`*`:\WebSphere\CommerceServer\working\` `WC_Enterprise_App_`*`instanceName`*`.ear`

   and click **Open**. Wait for the application to open before continuing to the next steps. This takes several minutes.

6. Click **EJB Modules**. The pane on the right displays the EJB modules in the enterprise application.

7. Click the EJB module for the EJB group that you have modified. For example, if you have modified a bean in the WCSUser EJB group, you would click **WCSUser**.
8. Click the General tab to view the class path information. Copy this existing class path information into a text file (for example, `WCSUser_path.txt`).
9. Right-click the EJB module and select **Delete**.
10. Right-click **EJB Modules** and select **Import**.
11. In the **File name** field, enter the following:
    - Windows `*drive*:\WebSphere\CommerceServer\temp\Cust_*EJBGroupName*-ejb.jar`
    - AIX `/usr/WebSphere/CommerceServer/temp/Cust_*EJBGroupName*-ejb.jar`
    - Solaris `/opt/WebSphere/CommerceServer/temp/Cust_*EJBGroupName*-ejb.jar`
    - Linux `/opt/WebSphere/CommerceServer/temp/Cust_*EJBGroupName*-ejb.jar`
    - 400 `*iSeries_drive*:\QIBM\UserData\WebCommerce\instances\*instanceName*\temp\Cust_*EJBGroupName*-ejb.jar`

    and click **Open**. In the Confirm Values window, click **OK**.
12. Once the `*EJBGroupJARFile*.jar` file is imported, scroll to the modified EJB group and select this group.
    Information about this group is shown in the pane on the right.
13. Open the text file containing the class path information for the previous version of the EJB group. Select and copy the class path.
14. In the **Classpath** field for the modified EJB group paste in this class path information.
15. Click **Apply**.
16. From the **File** menu, select **Close**.
17. Wait for the file to close, then from the **File** menu, select **Open** and reopen the `WC_Enterprise_App_*instanceName*.ear` file.
18. Configure security for the modified bean, by doing the following:
    a. With the EJB Modules node expanded, locate and expand the node for the modified EJB group.
    b. Expand **Entity Beans**.
    c. Expand the modified EJB group.
    d. Click **Method Extensions**, then in the pane on the right do the following:

1) Click the **Advanced** tab.
2) Ensure that **Security identity** is selected.
3) For each method, ensure that **Use identity of EJB server** is selected.
4) Click **Apply** (if you have made any modifications).

e. In the left navigation pane, right-click **Security Roles** under the modified EJB group and select **New**, then do the following:

1) In the **Name** field, enter `WCSecurityRole` and click **Apply**. Note, if this role exists already, you do not need to perform this step.

f. In the left navigation pane, right-click **Method Permissions** under the modified EJB group and select **New**, then do the following:

1) In the **Method Permission Name** field, enter `WCMethodPermission`
2) In the **Methods** selection area, click **Add**.
The Add Methods window opens.
3) Expand the *modifiedEJBGroup* and select all of the enterprise beans (hold the Shift key while selecting). Click **OK**. All enterprise beans are then displayed under the Enterprise bean column and all methods are shown under the Types column.
4) In the Roles selection area, click **Add**, select the `WCSecurityRole` and click **OK**.
5) Click **Apply**.

19. From the **File** menu, select **Save**.
20. Close the Application Assembly Tool.
21. ▶ 400 Copy the newly modified `WC_Enterprise_App_`*`instanceName`*`.ear` file from the local machine to the iSeries machine running WebSphere Application Server. That is, copy the file from the following directory:

```
drive:\WebSphere\CommerceServer\working
```

into the following directory:

```
iSeries_drive:\QIBM\UserData\WebCommerce\instances\instanceName\working
```

where *`iSeries_drive`* is the drive letter that you have mapped to your iSeries IFS.

After this step has completed, you have created a new enterprise application that contains all of the previous logic as well as your new business logic. This is all contained in the newly modified `WC_Enterprise_App_`*`instanceName`*`.ear` file.

Stopping and removing an enterprise application

This section describes how to use the WebSphere Application Server Administration Console to stop an enterprise application that is currently running and then remove it.

To stop and them remove your enterprise application, do the following:

1. Open the WebSphere Application Server Administration Console.
2. Expand **WebSphere Administrative Domain**.
3. Expand **Nodes**.
4. Expand ***nodeName*** (where ***nodeName*** is the name of your node).
5. Expand **Application Servers**.
6. Windows AIX Solaris Linux Right-click your WebSphere Commerce application server. For example, right-click the **WebSphere Commerce Server - *nodeName*** and select **Stop**.
7. 400 Right-click your WebSphere Commerce application server. For example, right-click the ***instanceName* - WebSphere Commerce Server** and select **Stop**.
8. Expand **Enterprise Applications**.
9. Windows AIX Solaris Linux Right-click your WebSphere Commerce application. For example, right-click the **WebSphere Commerce Enterprise Application - *instanceName*** application and select **Stop**.
10. 400 Right-click your WebSphere Commerce application. For example, right-click the ***instanceName* - WebSphere Commerce Enterprise Application** application and select **Stop**.
11. Right-click your WebSphere Commerce application. For example, right-click the **WebSphere Commerce Enterprise Application - *instanceName*** (or ***instanceName* - WebSphere Commerce Enterprise Application** for iSeries) application and select **Remove**.
12. When prompted to indicate if the application should be exported, select **No**.
13. When prompted to indicate if the application should be removed, select **Yes**.

Importing an enterprise application

This section describes how to use the XMLConfig command line utility to import an enterprise application.

To import the new enterprise application, do the following:

1. Windows AIX Solaris Linux Invoke the XMLConfig tool to import the enterprise application into WebSphere Application Server by entering the following command:
   Windows
   ```
   xmlConfig -import OutputFile.xml -adminNodeName wasHostName
   ```
   AIX
   ```
   /usr/WebSphere/AppServer/bin/XMLConfig.sh -import OutputFile.xml
    -adminNodeName wasHostName
    -nameServiceHost wasHostName -nameServicePort wasAdminPort
   ```
   Solaris Linux
   ```
   /opt/WebSphere/AppServer/bin/XMLConfig.sh -import OutputFile.xml
    -adminNodeName wasHostName
    -nameServiceHost wasHostName -nameServicePort wasAdminPort
   ```
   where
   - *wasHostName* is the name of the node in the WebSphere Application Server that contains your current enterprise application.
   - *OutputFile.xml* is the XML file that describes all of your enterprise beans.
   - *wasAdminPort* is the WebSphere Application Server administration port.
2. 400 Invoke the XMLConfig tool to import the enterprise application into WebSphere Application Server by entering the following command:
   ```
   STRQSH
   PATH=/QIBM/ProdData/WebASAdv4/bin:$PATH
   xmlConfig -import
   /QIBM/UserData/WebCommerce/instances/instanceName/working/OutputFile.xml
   -adminNodeName wasHostName
    -nameServiceHost wasHostName -nameServicePort wasAdminPort
   -instance wasInstanceName
   ```
   where
   - *OutputFile.xml* is the fully-qualfied name of the XML file that describes all of your enterprise beans.
   - *wasHostName* is the name of the node in the WebSphere Application Server that contains your current enterprise application.

     Note: 400 The value for *wasHostName* is case-sensitive and must match the value that is in the TCP/IP configuration. (Use a command-line processor to access CFGTCP, option 12 to verify host name.)
   - *wasAdminPort* is the WebSphere Application Server administration port.

- `wasInstanceName` is the WebSphere Application Server instance name.

---

400 While attempting to run the `XMLConfig -import` command, you may receive the following error message:"Cannot expand the ear file under `/QIBM/UserData/WebAsAdv4/`*`wasInstanceName`*`/installedApps/WC_Enterprise_App_`*`instanceName`*`.ear`". If you recieve this message, remove or rename the preceding directory and run the command again.

---

3. 400 After you have imported the enterprise application, you must run a script to modify directory permissions. To run this script, do the following:
   - At a command prompt, enter the following:

     ```
     STRSQH
     cd /QIBM/ProdData/WebCommerce/bin
     changeAuthority wasAdminInstanceName instanceName
     ```

     where
     - *`wasAdminInstanceName`* is the name of your WebSphere Application Server administration instance.
     - *`instanceName`* is the name of your WebSphere Commerce instance.

Starting an enterprise application

This section describes how to use the WebSphere Application Server Administration Console to refresh the view and then start an enterprise application.

1. Open the WebSphere Application Server Administration Console.
2. Expand **WebSphere Administrative Domain**, then **Nodes**, then *nodeName*
3. Highlight the *nodeName* node.
4. Click the **Refresh selected subtree** icon.
5. Start your WebSphere Commerce application by doing the following:
   - Expand **Application Servers**.
   - Windows AIX Solaris Linux Right-click your WebSphere Commerce application. For example, right-click the **WebSphere Commerce Server -** *instanceName* and select **Start**.
   - 400 Right-click your WebSphere Commerce application. For example, right-click the *instanceName* **- WebSphere Commerce Server** application and select **Start**.

Appendix C. Tips for VisualAge for Java

This section describes some tips that relate to troubleshooting, performance improvements and simplification within the development environment.

Changing properties for the servlet engine in the WebSphere Test Environment

The properties for the servlet engine in the WebSphere Test Environment are controlled by the `default.servlet_engine` properties file. Within this file, you can modify the document root for the Web server and you can change the port used by the WebSphere Test Environment.

You may want to change the port, in a case where the WebSphere Test Environment has ceased to function and rebooting is not an option. To change the port, do the following:

1. Open the `VAJ_install_path\IDE\ProjectResources\IBM WebSphere Test Environment\properties\default.servlet_engine` file in a text editor.
2. In the `<transport>` stanza, change the `8080` value in the following line to an available port.
   ```
   <arg name="port" value="8080"/>
   ```
3. Change the `8080` value in the following lines to the same port specified above.
   ```
   <hostname-binding hostname="localhost:8080" servlethost="default_host"/>
      <hostname-binding hostname="127.0.0.1:8080" servlethost="default_host"/>
   ```
4. Save file.
5. Open the WebSphere Test Environment Control Center and start the Servlet Engine.

By default, the document root for the WebSphere Test Environment is `VAJ_install_path\IDE\ProjectResources\IBM WebSphere Test Environment\hosts\default_host \default_app\web`. To change this to another directory, do the following:

1. Open the WebSphere Test Environment Control Center and stop the Servlet Engine.
2. Open the `VAJ_install_path\IDE\ProjectResources\IBM WebSphere Test Environment\properties\default.servlet_engine` file in a text editor.
3. In the `<websphere-webgroup name="default_app">` stanza, change the `<document-root>$approot$/web</document-root>` to the following:
   ```
   <document-root>your_document_root</document-root>
   ```

where `your_document_root` is the desired document root.

4. Change the `8080` value in the following lines to the same port specified above.

   ```
   <hostname-binding hostname="localhost:8080" servlethost="default_host"/>
      <hostname-binding hostname="127.0.0.1:8080" servlethost="default_host"/>
   ```
5. Save file.
6. Open the WebSphere Test Environment Control Center and start the Servlet Engine.

Resolving persistent name server problems

If you experience problems with your persistent name server, you may need to drop and recreate the persistent name server database. Refer to the *Commerce Studio Installation Guide* for details on creating this database.

Alternatively, if a message indicating that the port is currently in use, ensure that you do not have WebSphere Application Server running.

Deleting compiled JSP files

VisualAge for Java maintains a project folder in which compiled JSP files are stored, for performance reasons. There may be times when you need to delete compiled JSP files. For example, if you remove a data bean from a JSP file, you may experience errors the next time you call the JSP file. In this case, you can delete the compiled JSP file.

To delete compiled JSP files, do the following:

1. In VisualAge for Java, select the Project tab.
2. Scroll down to the **JSP Page Compile Generate Code** project.
3. Select either the whole project (if you need to delete many compiled JSP files) or expand the project and delete only those that must be recompiled.

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10594
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS 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.

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:

400	iSeries
AIX	MQSeries
AS/400	Net.Commerce
CICS	Net.Data
DB2	S/390
@server	VisualAge
IBM	WebSphere
	zSeries

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

Oracle is a registered trademark of Oracle Corporation in the United States, other countries, or both.

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

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

Index

A

access control 85
- command-leval 96
- Groupable interface 101
- policies 87
- Protectable interface 101
- protecting resources 102
- resource-leval 96

adapters 9
application architecture 4

C

CMDREG 28
code repository 187
command design pattern 20
command flow 24
command registry 26
commands
- command context 127
- customize existing 139
- factory 23
- framework 21
- implementation 123
- interfaces 22
- registration 26
- types 11
- writing new business policy commands 153
- writing new controller commands 128
- writing new task commands 138

controller command invoker data bean 40
controller commands
- customize existing 139
- long-running 131
- writing new 128

customized code
- deployment 187
- packaging 126

D

data beans
- activating 40
- BeanInfo 40
- customize existing 145
- description 13
- interfaces 38

data beans *(continued)*
- command data bean 39
- input data bean 39
- smart data bean 38
- types 37

database commits 135
database considerations
- datatype 80
- naming 78

database locks 73
deployment descriptors 47
deployment of
- modified commands and data beans 192
- modified entity beans 192
- new commands and data beans 189
- new entity beans 190

design patterns 19
- command 20
- display 36
- model-view-controller 19

development environment 185
differences from Version 4.1 15
display design pattern 36

E

EJB deployed code 188
entity beans
- cache 75
- deployment descriptors 47
- description 12
- extending 48
- overview 45
- transactions 73
- using 77

error handling 111
- command 111
- exception types 111
- flow 112
- in custom code 114
- JSP 121
- trace 119

F

flushRemote method 75

J

JSP templates 13
- setting attributes 41

M

messages
- creating messages 116
- properties files 112

model-view-controller design pattern 19
modifyIsolationLevel command 349

O

object life cycles 72
object model extension methodologies 48

P

packaging customized code 126
persistence 45
protocol listeners 8

R

relationship groups 92
run-time architecture 6

S

servlet engine 8
session beans
- recommended use 52
- writing new 71

software components 3

T

task commands
- customize existing 143
- writing new 138

terms and conditions 159
tracing execution flow 119
trading agreements 147
transaction isolation levels 47
transaction scope 135

U

URLREG 26

V

view commands
- format input properties to 132
- required properties 43

VIEWREG 32

W

Web controller 10

Part Number: CT1JYNA

Printed in U.S.A.

GC09-4951-02

(1P) P/N: CT1JYNA