Comergent eBusiness System

Release 7.0.2

Developer Guide

Sterling Commerce
An IBM Company

Comergent eBusiness System Developer Guide

Documentation part number: 1-7.0.2-2-02

Preface

Welcome to the Comergent eBusiness System. This *Developer Guide* and the associated documentation provides all the information required for you to implement the Comergent eBusiness System at your enterprise.

Purpose

This guide provides an overview to extending current Comergent Technologies applications and developing new applications for the Comergent eBusiness System. It presents a description of the system architecture, the main Java classes, and a description of the Comergent eBusiness System Software Development Kit (SDK).

Audience

This guide presupposes an advanced level of information systems knowledge, familiarity with basic network and database concepts, Java (including the J2EE specification) and XML. Readers must have a firm understanding of developing Web applications in Java.

Conventions

Throughout this guide, we will use the following conventions shown in Table 1, "Conventions", on page iv:

TABLE 1. Conventions

Type	Convention
File names	**Sample.txt**
Paths and directory names	**/top_level/next_level/next_level/destination_directory/**
Sample code extracts	`public void method(String s)`
Values to be provided	`<value supplied by developer>`

Comments

We welcome your feedback. Our aim is to provide our customers with the best quality documentation possible. Let us know about any inaccuracies or missing information in our documentation. We also welcome suggestions for enhancements to our documentation. Our email address is:

`support@comergent.com`

Contents

CHAPTER 17 Customization Examples.........................165

Introduction to J2EE Web Applications

This chapter presents an overview of the Java 2 Platform, Enterprise Edition (J2EE) and how it is used to deploy Web applications. If you are already familiar with this architecture, then you can skip this chapter.

Architecture

Release 7.0.2 of the Comergent eBusiness System is designed to conform to the Java 2 Platform, Enterprise Edition (J2EE) architecture as defined in *Java 2 Platform Enterprise Edition Specification, v 1.2* published by Sun Microsystems, Inc.

The Comergent eBusiness System is deployed as a Web application that comprises a set of Java classes together with accompanying configuration files, HTML templates, and JSP (JavaServer Pages) pages. It must be installed into a servlet container that conforms to the J2EE standard.

Web Applications

A J2EE Web application is built to conform to a J2EE specification. You add Web components to a J2EE servlet container in a package called a *Web application archive* (WAR) file. A WAR file is a JAR (Java archive) file compressed file.

A WAR file usually contains other resources besides Web components, including:

- Server-side utility classes

- Static web resources (configuration files, HTML pages, image and sound files, and so on)

- Client-side classes (applets and utility classes)

The directory and file structure of a Web application deployed as a WAR file conforms to a precise structure. A WAR file has a specific hierarchical directory structure. The top-level directory of a WAR file is the *document root* of the application. The document root is the directory under which JSP pages, client-side classes and archives, and static Web resources are stored. The document root contains a subdirectory called **WEB-INF/**, which contains the following files and directories:

- **web.xml**: the Web application deployment descriptor. It describes the structure of the Web application.

- Tag library descriptor files.

- **classes/**: a directory that contains server-side classes: servlet, utility classes, and Java Beans components.

- **lib**/: a directory that contains JAR archives of libraries (tag libraries and any utility libraries called by server-side classes).

web.xml File

Every Web application deployed in a servlet container must have a **web.xml** file present in its **WEB-INF/** directory. The structure of every **web.xml** conforms to a DTD published as part of the J2EE specification.

The purpose of the **web.xml** is to specify the general configuration of the Web application as required by the J2EE standard. Specifically:

- initialization parameter values are provided for the Web application

- servlet classes used by the Web application may be declared and given names

- each servlet class is mapped to one or more URL patterns: when the servlet container receives a request whose URL matches a pattern defined in the **web.xml** file, then the corresponding servlet is used to process the request

- initialization parameter values are provided for each servlet if required

- session information (such as time out)

- the location of custom tag libraries used by the JSP pages

JSP Pages

Early Java-based Web applications used only servlets to generate the HTML that was sent back to user's Web browsers. Over time, template mechanisms were introduced that enabled Web developers to generate dynamic content by using templates to generate the HTML. Several such template systems are available, however the J2EE architecture has settled on the use of JSP (JavaServer pages) pages to display content.

When a J2EE application receives a request from a user's browser, it first processes the request to extract parameters from the request and to perform business logic initiated by the request. Once the processing is complete, the Web application must dispatch the request to a JSP page: it does this by using a *request dispatcher*. Typically, the servlet context invokes a request dispatcher by passing the target JSP page to the dispatcher and then the request and response objects are *forwarded* by the request dispatcher.

A JSP page comprises a combination of HTML, JSP tags, and scripting elements such as *scriptlets*.

- HTML: a JSP page can include any amount of normal HTML. This content is passed right through to the browser page without change.

- JSP tags: tags are used to populate the dynamically-generated HTML with values calculated as the page is being generated. There are standard JSP tags such as <jsp:getProperty>, <jsp:include>, and <jsp:forward>. These are available to anyone creating a JSP page. In addition, you can specify that your Web application uses one or more custom tag libraries. Each custom tag library must be declared in the **web.xml** file for the Web application and the declaration must specify both the URI for the tag library and the location of the tag library descriptor (TLD) file.

Attention:	In the Comergent eBusiness System, the use of the tag libraries is now deprecated. For performance reasons, we suggest that you use scriptlets. JSP tags can still be used in some existing applications or specialized integration tasks.

- Scripting elements: You can intersperse the HTML and JSP tags in a JSP page with Java code that is contained between the scriptlet opening tag <%

(or <jsp:scriptlet>) and the closing tag %> (or </jsp:scriptlet>). Scriptlets are most commonly used to manage complex flow control in a JSP page.

Note that most JSP scripting elements can be invoked using a shorter form as described in the following table.

TABLE 2. Short Forms of Standard JSP Tags

Short form	XML form
<%	<jsp:scriptlet>
<%=	<jsp:expression>
<%!	<jsp:declaration>
<%@	<jsp:directive>

Data is passed to a JSP page using a variety of mechanisms, the most important of which are: implicit objects and beans.

• Implicit objects: Every JSP page provides the Web developer with objects that can be used to display data on the generated HTML page. The most important of these are the page, request, session, config, and application objects.

• Beans: Most of the data generated by the business logic of the application is passed to the JSP page by adding Java beans to one of the implicit objects listed above.

Model 2 Architecture

The Comergent eBusiness System is designed to conform to Sun's "Model 2" architecture. In this architecture, three functional components referred to as the Model, View, and Controller (MVC) partition the functionality of the Web application into logically distinct components.
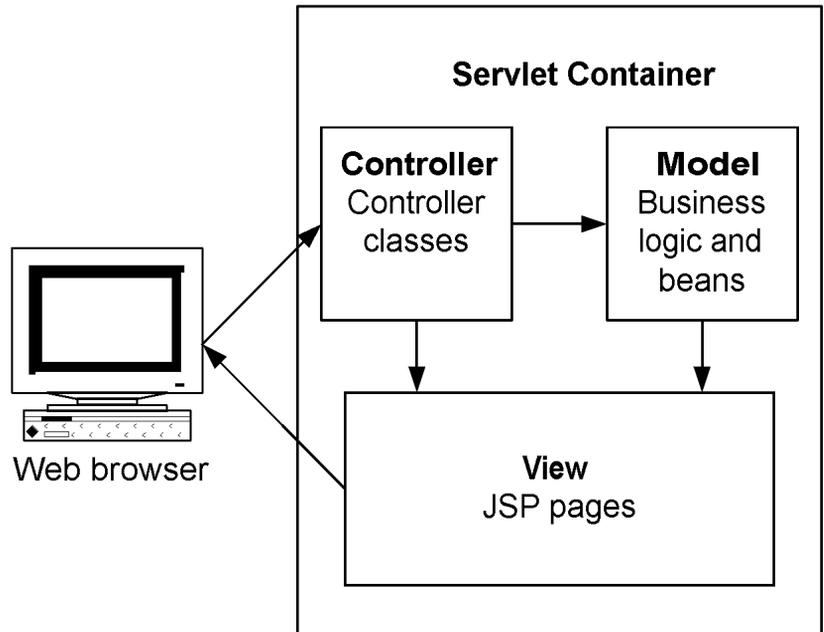
FIGURE 1. Model 2 Architecture

- *Model*: this component manages the data and business objects that are used by the system.

- *View*: this component is responsible for generating the content displayed to the user.

- *Controller*: this component determines the logical flow of the application. It determines what actions are performed on the model and manages the communication between model and view components.

Controllers

In the Model 2 architecture, controllers are Java classes intended to manage the processing of an inbound request and then to forward the request to an appropriate JSP page. The basic structure of a Comergent eBusiness System controller follows this form:

```
public class GenericController extends Controller
{
```

```
public void execute() throws Exception
{
    //Dispatch some business logic
    BizObjs resultBizObjects = calculate();
    //Generate the beans
    Vector beans = generateBeans(resultBizObjs);
    //Attach the beans to the request
    attachBeans(beans);
    // Dispatch to JSP page
    String pageName = choosePageLogic();
    // Dispatch to JSP page
    Dispatcher rd = request.getDispatcher(pageName);
    rd.forward(request, response);
}

protected BizObjs calculate() throws Exception
{
    //do some processing
    return resultBizObjs;
}

protected Vector generateBeans(BizObjs bizObjs)
{
    //create beans from business objects
    return beans;
}

protected void attachBeans(Vector beans)
{
    Iterator it = beans.iterator();
    while (it.hasNext())
    {
        DataBean bean= (DataBean) it.next();
        request.setAttribute (beanName, bean);
    }
}

protected String choosePageLogic()
{
    //logic to determine where to forward the request
    return pageString;
}
}
```

Model

In the Model 2 architecture, the objects that represent data in the system are maintained by the model component. It is common to distinguish the business objects from the beans used in the JSP pages.

Once the business logic class has finished creating and transforming the business objects, the controller class transforms the business objects into their corresponding beans. The beans are then passed to the JSP page for presentation.

View

The user interface of the Web application is served to the browser using JSP pages. Data is passed to each JSP page in the form of beans. These are classes with defined accessor methods that enable the logic on the JSP page to retrieve values using tags of the general form:

```
<%
    DataBean dataBean = request.getAttribute("nameOfBean");
    String stringProperty =
        dataBean.getNamedProperty("nameOfProperty");
%>
```

Note that it is possible to use a combination of scriptlets, simple JSP tags, and more sophisticated custom tags to manage page layout and the display of data.

Further Reading

The published literature on Web applications, J2EE, servlets, and JSP pages is vast. The following are recommended books for further reading:

- Hall, *Core Servlets and JavaServer Pages*, Second Edition, Prentice Hall, 2003

- Hunter, *Java Servlet Programming*, Second Edition, O'Reilly, 2001

- Fields and Kolb, *Web Development with JavaServer Pages*, Second Edition, Manning, 2001

New Features

This chapter presents an overview of the changes made to the Comergent eBusiness System that impact the way you work to implement, customize, and develop applications. A description of new functionality in Release 7.0.2 is documented in the *Comergent eBusiness System Overview Guide*.

Modularity

Architecturally, the main change made to the Comergent eBusiness System Release 6.0 and later has been to organize the product into modules. See CHAPTER 11, "Modularity and Generated Interfaces" for more details.

Modules invoke methods from other modules using the external APIs defined in the com.comergent.api packages. These packages define the externally-supported API for the Comergent eBusiness System and will be supported for two releases. The implementation classes that implement the external API are organized so that they can be maintained and versioned without making changes to the API.

Data Services

The main changes made in the data services layer are:

• The automatic generation of interfaces for the data beans.

- The replacement of business objects by data objects.

- Minor additions and changes to the syntax of the XML schema.

- The introduction of the DataContext class.

- The replacement of the Query business object with the DsQuery class.

In general, none of the changes will cause earlier releases of the Comergent eBusiness System schema to break if deployed into Release 6.0.

File Reorganization

Changes have been made to the directory and file organization of the Comergent eBusiness System. These are:

- The source directories have been organized to reflect the module structure of Release 6.0.

- The application JSP pages have been moved to the *debs_home*/**Comergent/WEB-INF/web/** directory. This protects the JSP pages from unauthorized access. Note that the static content (HTML, GIF files, Javascript, and style sheet files) are stored under *debs_home*/**Comergent/**. In each location, each locale has a separate directory under it.

Hosting and Partner.com

Release 5.6 and earlier releases provided a means to host more than one logical installation of the Comergent eBusiness System as a single Web application. Each hosted partner was supported by their own instance of the data manager and database schema. In Release 6.0 and later releases, this functionality is supported by the creation of Partner.com and Storefront partners in the Comergent eBusiness System. All the information relating to the Partner.com and Storefront partners is held in the same database schema: there is no longer any need for a separate database schema for each hosted partner.

Migrating data from an earlier implementation of hosted partners to Release 6.0 is a complex task. Please consult Comergent Support for further information.

Security

The following changes have been made to the security mechanisms of the Comergent eBusiness System:

- Changes have been made to the access APIs: these ensure that basic entitlement checks can be performed simply at the data bean level.

- Changes have been made to the APIs for creating and modifying ACLs.

- The concept of security domains has been introduced to manage roles. This enables administrators to distinguish roles supported by the enterprise and by partners.

See CHAPTER 7, "Using the Security Mechanisms" for further information.

Software Development Kit

New targets have been added to the SDK to support the creation of the database schema and the loading of data. The targets also manage the new module structure of the source directories. See CHAPTER 14, "Software Development Kit" for further information.

Internationalization

In earlier releases of the Comergent eBusiness System, JSP pages were maintained for each locale supported by the implementation. In Release 6.0 and later, all locales that are part of the same locale family use the same JSP pages. All displayed text in these pages is declared using the text tag. Each text tag has an ID attribute. For each supported locale and for every JSP page and JSP fragment, there is a properties file that maps the ID of a text tag to an appropriate string.

At runtime, the user's current locale is used to render the page using the appropriate properties files and so their Comergent eBusiness System experience matches the locale they have requested. The JSP pages for a given locale family are maintained in a single directory tree under the default locale for that family: for example, en_US for the Roman family of locales.

If you need to support locales that will require substantial changes to the page layout, then you must create a different set of JSP pages that will be used for these locales and these will be organized into a separate locale family. They should be placed in a default locale directory for the family.

See CHAPTER 32, "Internationalization" for further information.

CHAPTER 3 *System Architecture*

This chapter presents a detailed description of the architecture of the Comergent eBusiness System. It assumes a thorough understanding of the J2EE architecture. It also provides an introduction to some of the important Java classes used by the Comergent eBusiness System and its applications.

This chapter is intended to help you if you want to modify or extend existing Comergent Technologies applications or write new applications. Note that not all parts of the Comergent eBusiness System conform to this architectural description. Legacy components that do not match this architecture need to be customized by trained Comergent Technologies professional services staff.

**Comergent Distributed
E-Business System**

DispatchServlet and JSP pages

Web browser

Controllers

Business logic classes
and data beans

Knowledgebase
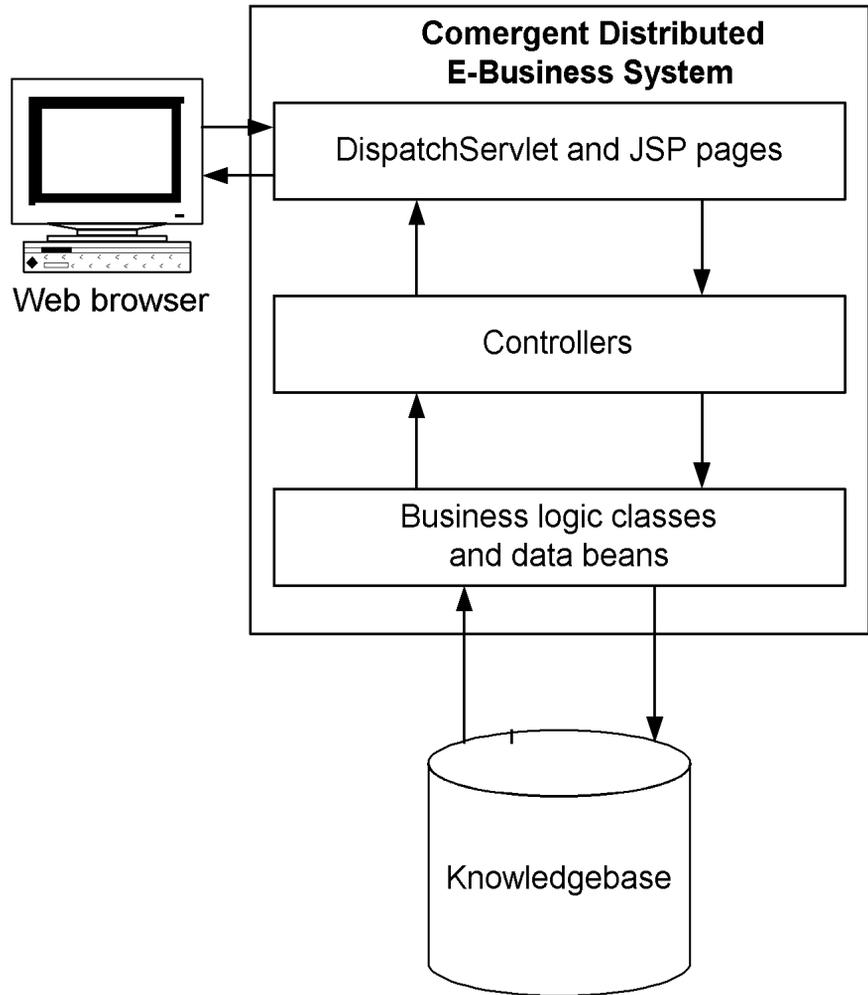
FIGURE 2. Comergent eBusiness System Architecture

Comergent eBusiness System Web Application

When you install the Comergent eBusiness System into your servlet container, it
installs as a WAR file, **Comergent.war**. When the WAR file deploys, it unjars into
a directory called **Comergent/**. See the *Comergent eBusiness System*

Implementation Guide for complete instructions to install the Comergent eBusiness System. See the *Comergent eBusiness System Reference Guide* for a complete description of the directory structure under the **Comergent/** directory.

In the **WEB-INF/** sub-directory, you find the **web.xml** file for the Comergent application.

The most important configuration settings in this file are:

- The definition of the InitServlet and DispatchServlet:

    - InitServlet loads when the servlet container starts and it reads in all of the configuration information for the Comergent eBusiness System using the value of the propertiesFile element: by default this is **Comergent.xml**. See the *Comergent eBusiness System Reference Guide* for a complete description of the configuration files and their settings.

    - DispatchServlet is the main servlet used to process inbound requests. Most of the URLs defined in the servlet mapping section resolve to the DispatchServlet.

- The servlet mapping section maps most URL patterns to the DispatchServlet. Note that "/msg/*" is used to map requests to the MessagingServlet: this ensures that inbound XML messages are processed by this servlet class.

- The session configuration element sets a session timeout value of 30 (minutes). Each implementation of the Comergent eBusiness System must carefully consider an appropriate value for this parameter. Bear in mind the following:

    - End users of the system may leave their browsers unattended while they step away from their desks. If an unscrupulous user can access the browser when a session is still valid, then they can access the system.

    - End users may punch out to other external systems in the course of using the Comergent eBusiness System. The session timeout value must give enough time for users to punch out and return.

    - Each session uses system resources. The greater the session timeout value, then the greater the memory usage of the system.

- The location of the Comergent tag library descriptor (TLD) file is provided. The Comergent tag libraries are documented in CHAPTER 30, "Comergent Tag Library" and CHAPTER 31, "Comergent Internet Commerce Tag Library".

Processing Requests

When the Comergent eBusiness System receives a request from a user's browser, it must determine how to process the request and how to display the result to the user. It does this using the **MessageTypes.xml** configuration files. These files determine the mapping between a request and the logic processing classes and JSP pages used.

1. When a request is received, the message type is identified and the appropriate controller invoked.

2. Additional business logic may be invoked using a business logic or bizAPI class.

3. The controller then forwards the request to the specified JSP page to render the output back to the user's browser.

The messageTypeFilename element of the GeneralObjectFactory element of the **Comergent.xml** file specifies the comma-delimited list of **MessageTypes.xml** file used to specify the message types. Each **MessageTypes.xml** file declares a list of message types organized by message group.

Each request specifies the message type as the cmd parameter. For example, if the URL is of the form:

```
../Comergent/catalog/matrix?cmd=search
```

then the name of the message type is "search".

Each message type is identified by the Name attribute of its MessageType element. The Name attribute identifies which message type is being requested when a user clicks a URL.

Attention:	You must make sure that each message group and message type have a unique name. You must check the collection of **MessageTypes.xml** files to ensure that you have not defined message groups and message types with the same name. See "Overriding MessageType Definitions" on page 17 for an exception to this rule.
	We suggest that you list message types alphabetically by name within message groups as a means of quickly identifying the duplication of message type names.

MessageType elements have one or more of the following child elements:

- BizletMapping: used for message processing, it associates a Bizlet class and a method of this class to process the message.

- BLCMapping: associates a business logic class (BLC) to be used to process the request.

- ControllerMapping: associates a controller to be used to process the request. For message processing, you can specify a BizRouter class to invoke a Bizlet class to process the message.

- JSPMapping: associates a JSP page to be used to display the result of processing the request.

See CHAPTER 5, "Bizlets" for more information about the use of bizlets. A MessageType element may specify any combination of these three elements.

- If no ControllerMapping element is specified, then, by default, the ForwardController class is used. This class simply forwards the request to the JSP page specified by the JSPMapping element. If no JSPMapping element is found or if the specified JSP page is missing, then an error page is displayed.

- If the SimpleController is specified in the ControllerMapping element, then the business logic class specified by the BLCMapping element is invoked to process the request (see "Business Logic Classes" on page 26).

- If a custom controller is specified, then it may process the request itself (see "Controller Classes" on page 20), or it can invoke a business logic class using the *runAppJob()* method of the AppExecutionEnv class (see "AppExecutionEnv Class" on page 26).

- If no JSPMapping element is specified, then the business logic class or controller must specify which JSP page is to be used.

Each request or message is validated against the entitlements system to verify that the user can execute the message type. Not all users can execute all message types: see the *Comergent eBusiness System Reference Guide* for a discussion of how the entitlements mechanism manages message types as part of the security of the Comergent eBusiness System.

Overriding MessageType Definitions

The MessageType element has an optional attribute: IsOverlay. If this attribute is set to "true", then the MessageType definition overrides any previous definition of this message type given in any earlier **MessageTypes.xml** file listed in the messageTypeFilename element.

If two or more definitions are given for the same message type without one specifying the isOverlay attribute, then an initialization error is displayed and the first definition of the message type is used.

Note that the IsOverlay attribute does not change the location of the MessageType: this is still determined by the message group to which the first definition belongs or by the MessageTypeRef element that references the message type.

For example, to override the definition of the adirectLogin message type, you can define an element as follows:

```
<MessageType Name="adirectLogin" IsOverlay="true">
   <ControllerMapping>
       com.comergent.apps.common.controller.MyLoginController
   </ControllerMapping>
   <JSPMapping>../common/adirectPageLoader.jsp</JSPMapping>
</MessageType>
```

The IsOverlay attribute can also be used for MessageGroup declarations so that you can overwrite the definition of a message group, but its use is not recommended.

Default Elements

For each message group, you can specify default BizletMapping, BLCMapping (deprecated), ControllerMapping, and JSPMapping elements. These are used when no mapping is specified for a message type that belongs to the message group.

In general, if no default mapping is specified in a message group, then the system looks for a default mapping in the parent message group of the current message group. If no mapping is found anywhere in the message group tree, then values specified in the MessageGroupDefaults message group are used.

Key Java Classes

At a schematic level, the Comergent eBusiness System applications all have the same structure: they are composed of controllers, business objects and business logic classes (BLCs), and JSP pages.

Wrapper Classes

Several of the standard classes used in J2EE Web applications have been wrapped in Comergent wrapper classes to manage any minor idiosyncrasies among the supported servlet containers:

ComergentContext

This class is used to wrap the servlet container context. You can use it to retrieve the Env object for environment information. Note that any context attribute that is set must be serializable. An exception is thrown if you attempt to set a non-serializable attribute.

It provides the *getResourceAsStream()* method: this method can be used to access a file as a stream for read-only access. You must use the *adjustFileName()* method of the LegacyFileUtils class for write access to a file.

ComergentDispatcher

This class is a lightweight wrapper of the standard RequestDispatcher class: it provides *forward()* and *include()* methods.

ComergentRequest

This class wraps the standard HttpRequest class and provides helper methods to parse the inbound requests and messages.

ComergentResponse

This class wraps the standard HttpResponse class. It provides a *localRedirect()* method to pass a request with a new message type. For example, you may want a controller to process a request, and then to pass the result on to another controller: you do this by calling:

```
response.localRedirect(request, "messageType");
```

This has the effect of submitting the request to the DispatchServlet as if it had been received as an HTTP request.

ComerentSession

This class wraps the standard HttpSession class. When a user first logs in, a User data bean is created and added to the ComergentSession object. You can access user information through the ComergentSession *getUser()* method.

For example:

```
session.getUser().getUserKey()
```

will return the current user's key; and

```
session.getUser().getPartnerKey()
```

returns the key of the partner to whom the user belongs.

The ComergentSession object is used to store information that must be persistent for more than one request of a user's session. Use the

setAttribute(String s, Object o) method to set an object in the session and *getSession(String s)* to retrieve it. Objects stored in the session must implement the Serializable interface: all generated data beans implement this interface and so these may be stored in the session.

The ComergentSession class also provides a *logout()* method: invoking this method immediately invalidates the servlet container session.

Servlets

The main servlets used are:

- InitServlet: this servlet loads when the servlet container starts. Its *init(ServletConfig config)* method initializes the ComergentAppEnv class.

- DispatchServlet: this servlet is used to service almost all requests processed by the Comergent eBusiness System. Its principle method call is:

  ```
  void dispatch(HttpServletRequest request, HttpServletResponse response)
  ```

  This method creates a controller to handle the request with:
  ```
  Controller controller createController(ComergentRequest comergen-
  tRequest)
  ```

  and then invokes:
  ```
  controller.init(comergentContext, comergentSession,
      comergentRequest, comergentResponse);
  controller.execute();
  ```

  Note that the instance of the Controller class created by the *createController()* method is a function of the request. The request message type determines the Controller class because the controller is created by the GeneralObjectFactory class. The GeneralObjectFactory uses the **MessageTypes.xml** file to map from the request message type to a Controller class. See the *Comergent eBusiness System Reference Guide* for more information about the configuration files.

- DebsDispatchServlet: this servlet is used to process XML messages posted from another system to the Comergent eBusiness System. If the content type of the request starts with "application/x-icc-xml" or "text/ xml", then it invokes the MessagingController to process the request.

Controller Classes

The Comergent eBusiness System offers two different ways of using controllers to process requests:

Custom Controllers

You can write your own Controller class by extending the com.comergent.dcm.caf.controller.Controller class. When you do this, you must provide the application logic to determine the JSP page to which the request should be forwarded. For example:

```
boolean processingSuccess = false;
/*
 *
 * Business logic processes request and sets processingSuccess to
 * true if successful.
 */

if (processingSuccess)
{
    callJSP("SuccessMessageType");
}
else
{
    callJSP("FailureMessageType");
}

protected void callJSP(String messageType) throws
    ControllerException, ICCException, IOException
{
    String resource = getJSPName(messageType);
    ComergentDispatcher rd =
        request.getComergentDispatcher(resource);
    rd.forward(request, response);
}

protected String getJSPName(String messageType) throws ICCException
{
    JSPObjectID id = new JSPObjectID(messageType);
    return GeneralObjectFactory.getGeneralObjectFactory().-
        getMapping(id);
}
```

SimpleController

You can extend the SimpleController class to process the request if there is only one exit point from the application logic. The SimpleController uses the message type of the request to determine the JSP page to which the request is forwarded once the application logic is finished. To extend the SimpleController class, overwrite the *calculate()* method.

MessagingController

This class is used to process XML requests (such as price and availability or shopping cart transfer requests from other systems).

DataBean Classes

Access to data in the Comergent eBusiness System is managed through data objects: these are XML documents that describe the business entities such as partners, users, products, and so on. They describe the fields of the data object together with information about how they map to database tables in the Knowledgebase. Each data object XML file is used to generate a corresponding DataBean Java class.

The DataBean classes are the main classes used to represent each business entity in the Comergent eBusiness System. Each business entity such as a user, partner, product, and so on, is represented in memory by an instance of the appropriate DataBean class. See CHAPTER 6, "Introducing Data Beans and Business Objects" for more information. Some legacy application may still use the BusinessObject class, but in general the use of the BusinessObject class is deprecated.

DataBean classes are also used to pass data to JSP pages. Any data object definition in the Comergent eBusiness System XML schema may be used to generate a DataBean class by running the generateBean target (see the CHAPTER 14, "Software Development Kit" for more details).

The DataBean class is a general abstract class and all generated data bean classes extend this class. Each DataBean class provides *restore()* and *persist()* methods that retrieve and save data in the database respectively.

Some applications make use of application beans: see "Application, Entity, and Presentation Beans" on page 68 for a discussion of how these beans are used.

ObjectManager and OMWrapper Classes

In Release 7.0.2, you should not instantiate DataBean classes by using their constructors. Instead use the ObjectManager and OMWrapper classes to create new instances of objects as your applications require them. These classes follow the Factory pattern in that they provide a class designed to generate object instances as they are required. They enable you to switch from one object class to another without changing the application code that creates and uses the objects.

Creating Objects

In general, you should use the OMWrapper class rather than the ObjectManager class, but both can be used. You use these classes to create objects with the following methods:

```
ObjectClass temp_ObjectClass =
    (ObjectClass) OMWrapper.getObject("ObjectName");
```

or

```
ObjectManager temp_ObjectManager = ObjectManager.getInstance();
ObjectClass temp_ObjectClass =
    (ObjectClass) temp_ObjectManager.getObject("ObjectName");
```

Mapping Object Names to Object Classes

The ObjectManager and OMWrapper classes use the **ObjectMap.xml** configuration file (located in *debs_home*/**Comergent/WEB-INF/properties/**) to determine which type of object is created from the object name provided in the *getObject()* method.

Attention:	Do not add comments to the **ObjectMap.xml** file: these can cause errors on initialization.

Each Object element is of the form:

```
<Object ID="ObjectName">
    <ClassName>ObjectClass</ClassName>
</Object>
```

When the *getObject("ObjectName")* method is invoked, an instance of the ObjectClass class is returned. The *ObjectName* must be the name of a Java class or interface and the *ObjectClass* must be a subclass of the *ObjectName* class (possibly itself) or a class that implements the *ObjectName* interface.

If the **ObjectMap.xml** file does not have an Object element whose ID attribute matches the ObjectName parameter, then the ObjectManager or OMWrapper creates an instance of the ObjectName class. That is, it behaves as if there is an element of the form:

```
<Object ID="ObjectName">
    <ClassName>ObjectName</ClassName>
</Object>
```

For example, suppose that the **ObjectMap.xml** file contains the element:

```
<Object ID="com.comergent.bean.productMgr.ProductBean">
   <ClassName>
        com.comergent.bean.productMgr.MatrixProductBean
```

```
        </ClassName>
</Object>
```

Then the following method invocation will create an instance of the MatrixProductBean class:

```
ProductBean temp_ProductBean = (ProductBean)
    OMWrapper.getObject("com.comergent.bean.productMgr.ProductBean");
```

Note that the MatrixProductBean must extend the ProductBean class: otherwise a ClassCastException would be thrown at runtime. However, if there is no element whose ID attribute is com.comergent.bean.productMgr.ProductBean, then the same call would return an instance of the com.comergent.bean.productMgr.ProductBean class.

Restrictions

Note that you cannot create Object definitions so that the class specified in the ClassName element in one Object element is the ID attribute in another Object element. The only exception to this rule is when the class is used both as the ID and ClassName values for a single Object element. In particular, if you extend a data object (see "Extending Data Objects" on page 63), then:

1. Define an Object element that maps the extended class to the extending class:

    ```
    <Object ID="<Extended class>">
        <ClassName><Extending class></ClassName>
    </Object>
    ```

2. Make sure that you replace any reference to the extended data object in any ClassName elements to the extending data object.

Passing Parameters

If you need to pass parameters to the object constructors, then the following OMWrapper method is also available:

```
ObjectClass temp_ObjectClass = (ObjectClass)
    OMWrapper.getObjectArg("ObjectName", Object arg1, ... ,
        Object arg10);
```

In this form, you can pass up to ten parameters as Objects into the method invocation. The following OMWrapper and ObjectManager method calls enable you to pass in an unlimited number of parameters as an array of objects:

```
ObjectClass temp_ObjectClass = (ObjectClass)
    OMWrapper.getObject("ObjectName", Object[] args);
```

or

```
ObjectClass temp_ObjectClass = (ObjectClass)
```

```
temp_ObjectManager.getObject("ObjectName", Object[] args);
```

For example, suppose that the **ObjectMap.xml** file contains the element:

```
<Object ID="com.comergent.bean.productMgr.OrderBean">
    <ClassName>com.comergent.bean.matrix.MatrixOrderBean</ClassName>
</Object>
```

Here, the MatrixOrderBean class is a subclass of the OrderBean class. Suppose that the MatrixOrderBean has a constructor of the form MatrixOrderBean(CartBean cb).

Then the following method invocation will create an instance of the OrderBean class using an instance of the CartBean class as a parameter:

```
Cart temp_CartBean = (CartBean)
    OMWrapper.getObject("com.comergent.bean.partnerMkt.CartBean");
/*
    Code that processes the cart bean object
*/
OrderBean temp_OrderBean = (OrderBean)
    OMWrapper.getObjectArg("com.comergent.bean.productMgr.OrderBean",
        temp_CartBean);
```

Object Pooling

If you expect some classes of object to be created and used frequently, then you can use the ObjectManager and OMWrapper classes to create a pool of objects. The parent object (identified by the ID attribute) must implement the poolable interface. This interface is a part of the com.comergent.dcm.objmgr package. It declares one method *reset()* that you must implement.

When you are finished with a poolable object, you can return it to the object pool by using the *return()* method as follows:

1. In the **ObjectMap.xml** entry for a pooled class, set the MaxPoolSize attribute to the number of objects you want created in the pool:

    ```
    <Object ID="ObjectName" MaxPoolSize="n">
        <ClassName>ObjectClass</ClassName>
    </Object>
    ```

2. Create instances of the object class using OMWrapper and ObjectManager as described above.

3. When you are finished with the object, then return the instance to the pool using:

    ```
    OMWrapper.return(temp_ObjectClass);
    ```

4. or

```
temp_ObjectManager.return(temp_ObjectClass);
```

Note that if you create an object by passing in parameters as described in "Passing Parameters" on page 24, then a new object is created rather than re-using an object from the pool.

Business Logic Classes

Each business logic class (BLC) is a subclass of the BLC abstract class. This abstract class implements the ApplicationObject interface. BLCs can perform the business logic of your implementation of the Comergent eBusiness System.

Each BLC contains a table of business objects such as session, user, product inquiry list for example. In executing the *service()* method of a BLC, it invokes the *persist()* and *restore()* methods of these business objects.

Note:	The use of BLC classes is deprecated. You should use either bizAPI classes or controllers.

Presentation Logic Classes

Presentation logic classes are deprecated. Do not use them.

AppExecutionEnv Class

The AppExecutionEnv class can be used to run business logic classes. However, the use of business logic classes is deprecated, so use this class only to support legacy applications. You use the static methods *runAppObj()* to invoke the creation of a business logic class and to execute its prolog and service methods.

In its most common form, you can use:

```
AppExecutionEnv.runAppObj(String messageType, BizObjTable bizObjects)
```

The AppExecutionEnv class invokes the business logic class determined by the messageType string and which takes the BizObjTable vector of business objects as the input business objects.

AppsLookupHelper Class

There are many situations in the Comergent eBusiness System where the status of a data object is managed using a lookup code. For example, the order status of an order can change several times through the placing of an order. There are also several examples of display fields such as the Title of a user which can take several well-defined values and which need to be managed for different locales. This data is stored in the CMGT_LOOKUPS table of the Knowledgebase database schema.

See the *Comergent eBusiness System Reference Guide* for further information about this table.

For each lookup type, there can be one or more lookup codes and each code has an associated description string. For example:

TABLE 3. **Lookup Example**

Lookup Type	Lookup Code	Description
AddressType	10	Billing
AddressType	20	Shipping

You can use the AppsLookupHelper class to map a lookup code to a description string. By invoking the appropriate method of the AppsLookupHelper class, pass in the lookup code as a parameter and the corresponding String is returned. Depending on which lookup type you are interested in, you choose the appropriate method for that lookup type. The method used determines which lookup type is used to retrieve the lookup code from the CMGT_LOOKUPS table. For example, to retrieve an order status code string, you can write:

```
String orderStatusString =
    AppsLookupHelper.getOrderStatusForCode(orderStatusCode);
```

Conversely, you can retrieve the lookup code using:

```
int orderStatusCode =
    AppsLookupHelper.getCodeForOrderStatus(orderStatusString);
```

Most, though not all, lookup types have helper methods defined. Check the Java doc for the AppsLookupHelper class for details. For further information, see "Support for Lookup Codes" on page 33.

ComergentAppEnv Class

Use the ComergentAppEnv class to provide your code with environment information specific to the application. It provides the following useful methods:

- *adjustFileName()*: this method has been moved to the LegacyFileUtils class. See "LegacyFileUtils Class" on page 29.

- *constructExternalURL()*: use this method to construct a URL that enables a client to be re-directed back to the server. Primarily, you use this method to generate a redirect URL to enable the server to restore session information.

- *getEnv()*: this method returns the environment object.

- *getContext()*: this method returns the application context.

Global Class

The use of this class is deprecated. See "Global Class" on page 467 for a description of its legacy methods. Its logging function has been replaced by the log4j API: see CHAPTER 8, "Logging" for more information. Its support for retrieving the values of properties has been replaced by the Preferences mechanism. If you need to continue to use code that uses the Global class, then replace each usage by the LegacyPreferences class.

GlobalCache Interface

This interface is used to define a cache that provides access to cached objects used by all Comergent eBusiness System applications. It can be used to support a clustered environment in which the Comergent eBusiness System is running on more than one machine.

To use a cache class that implements the GlobalCache interface, you must implement the methods of the interface. The cache class is loaded when the InitServlet *init()* method is invoked. You must provide the name of the class as the General.globalCacheImplClass element of the **Comergent.xml** file. A default implementation is provided with Comergent eBusiness System: com.comergent.dcm.cache.impl.AppContextCache.

You access the implementation of the GlobalCache interface by:

```
GlobalCache globalCache = ComergentAppEnv.getGlobalCache();
```

The interface supports the following methods:

- *public String store(Serializable entry)*: stores an object in the global cache, which remains until the application cleans it up.

- *public boolean store(String id, Serializable entry)*: stores an object in the global cache, which remains until the application cleans it up.

- *public String cache(Serializable entry)*: stores an object in the global cache. The object is available as long as the application is using it, but the cache system cleans it up automatically.

- *public String cache(Serializable entry, long lease)*

- *public boolean cache(String id, Serializable entry)*

- *public boolean cache(String id, Serializable entry, long lease)*

- *public boolean contains(String id)*: checks if the cache contains the specific object.

- *public Object get(String id)*: retrieves the cacheable object.

- *public Object remove(String id)*: removes a cacheable object.

- *public boolean gc()*: This method should be called by a Cron job so the cache can clean up unused entries.

LegacyFileUtils Class

The LegacyFileUtils class provides helper methods for working with files. Its use is deprecated, but it provides support for methods previously provided by the ComergentAppEnv class:

- *adjustFileName()*: It returns the real path name of a file. Use this method to access files for either reading or writing: do not use the *getRealPath()* method because this can return null.. In a clustered envrionment, the *adjustFileName()* method ensures that all members of the cluster access the same file. You must use this method with four parameters:

```
adjustFileName(String fileName, boolean share, boolean xPublic,
    boolean xLoadable);
```

Use of the one-parameter form of this method is deprecated. The boolean parameters are used to determine the location of the file using the configuration parameters specified in the WritableDirectory element of the **web.xml** file.

OutOfBandHelper Class

The OutOfBandHelper class provides a means to generate an output stream using a JSP page as a template. An example of its use is given here:

```
ComergentRequest request = ComergentAppEnv.getRequest();
ComergentResponse response = ComergentAppEnv.getResponse();
ByteArrayOutputStream stream = new ByteArrayOutputStream();
OutOfBandHelper outOfBandHelper = new OutOfBandHelper(request,
    response, stream);
outOfBandHelper.getRequest().setAttribute(
    ComergentRequest.COMERGENT_SESSION_ATTR,
    request.getComergentSession());
outOfBandHelper.callJSP(messageType);
/*
 * Initialize SendSMTP and use the stream to to set the body of the
 * message
*/
String mimeType = "text/html";
```

```
String smtpHost = Global.getString(
    "C3_Commerce_Manager.SMTP.SMTPHost");
SendSMTP smtp = new SendSMTP(smtpHost);
StringBuffer sb = new StringBuffer(subject);
String message = null;
String enc = ComergentI18N.getComergentEncoding();
message = stream.toString(enc);
//Send the mail
smtp.send( from, to, cc, subject, message, mimeType);
```

In this example, you can see how the OutOfBandHelper class is initialized using the existing request and response objects and an output stream. Its *callJSP()* method, generates the output stream by passing the request and response objects to the JSP page determined by the message type parameter, and the output stream can be used by the application to retrieve the content.

The OutOfBandHelper class makes use of session and context information when mapping a message type to a JSP page. Consequently, you can use different JSP pages for different locales in the same way as you do for processing browser requests and the OutOfBandHelper class will resolve which locale's JSP page to use and apply the same failover logic as described in "Failover Behavior" on page 400.

Preferences Class

The Preferences module provides the mechanism for accessing Comergent eBusiness System properties. It is one of the modules provided in the platform modules: see "Preferences Service" on page 44 for more information. The basic usage of the Preferences API is as follows:

```
private static Preferences temp_Preferences =
    Preferences.getPreferences();
String temp_MyPropertyString =
    temp_Preferences.getString("MyProperty");
```

The main methods it supports to retrieve properties are:

- public String getString(String key, String def)

- public boolean getBoolean(String key, boolean def)

- public double getDouble(String key, double def)

- public float getFloat(String key, float def)

- public int getInt(String key, int def)

- public long getLong(String key, long def)

There are corresponding *put*Type*()* methods for each *get*Type*()* method: for example:

• public void putString(String key, String value)

If you invoke the *getPreferences()* method without a parameter, then you retrieve the singleton Preferences object that the Comergent eBusiness System supports. If you pass in the name of a class (for example getPreferences(MyClass.class)), then the object you retrieve is scoped: that is the name of the properties whose values you retrieve using the Preferences object have the package path of the class prepended to the property name you provide.

For example, supose that MyClass is in the com.comergent.myApplication package. Then the following fragments of code are equivalent:

```
private static Preferences temp_Preferences =
    Preferences.getPreferences();
String temp_MyPropertyString =
temp_Preferences.getString("com.comergent.myApplication.MyProperty");
```

and:

```
private static Preferences temp_Preferences =
    Preferences.getPreferences(com.comergent.myApplica-
tion.MyClass.class);

String temp_MyPropertyString =
temp_Preferences.getString("MyProperty");
```

PriceCheckAPI Class

The PriceCheckAPI class provides the main means for applications to retrieve pricing information for products. It provides a number of static methods: these take as arguments a Vector of pricing line items and partner keys for the current user and the partner serving up the prices: either the enterprise or one of the Partner.com partners.

The main method is *Check()*: this method has several forms, but in general they all specialize the following method:

```
public static Vector Check (Vector lineItems, Timestamp date,
    Long partnerKey, Long storeFront, Long verticalKey,
    Long currencyKey)
```

All products must be passed in the Vector of pricing line items: these are objects of the PricingLineItem class. You can specify a quantity in each pricing line item. The date parameter enables you to retrieve prices as they would appear on a specified date: if the parameter is null, then the current date is used. The partnerKey parameter is the partner key of the user whereas the storeFront parameter is the

partner key of the current storefront: that is, think of the partnerKey as representing the buyer and the storefrontKey as the seller. The verticalKey parameter is the key of the current customer type and currencyKey is the key of the current currency.

It also provides methods to retrieve a list of price lists:

- *getAssignedPriceListKey()* returns a List of all price list keys assigned to the partner of the current user regardless of the current selection of currency and customer type.

- *getInContextPricePriceListKey()* returns a List of all price list keys assigned to the partner of the current user based on the session settings for currency and customer type.

Transactions

The Comergent eBusiness System for provides support for transactions: database actions that span one or more atomic operations. In general, you use the Transaction class to manage situations in which several data objects must be persisted together, and if one fails, then they should all fail. See "Transactions" on page 281 for more information.

Message Conversion Classes

Converter Classes

The Comergent eBusiness System must be able to transform XML documents from one form to another. The system uses converters for this purpose: these are classes that implement the Converter interface.

Note:	The converter makes use of stylesheets: these can be compiled into Java classes. A system property setting, compileStyleSheets, controls whether the stylesheets are compiled or not.

Message Categories

In order to convert from one document format to another, you must specify the source and target formats precisely. Each message must belong to a message family and a message version: together these define a *message category*. There can only be one form of a given message type within a message category.

For example, the message family dXML and the message version 5.0 uniquely determine a message category. Within this message category, there is only one form of the message type ShoppingCartTransfer.

Converter Interface

The Converter interface is defined by:

```
public interface Converter
{
    public void setConfig(MessageConversion mc);
    public MessageConversion getConfig();
    public Object getProperty(String name);
    public void setNext(Converter next);
    public Converter getNext();
    public String getIncomingMessageType();
    public String getConvertedMessageType();
    public void setSource(Document doc);
    public void setSource(InputStream is);
    public void setSource(Reader reader);
    public void setSource(DefaultHandler handler);
    public void setTarget(Document doc);
    public void setTarget(OutputStream os);
    public void setTarget(Writer writer);
    public void setTarget(DefaultHandler handler);
    public void setParameter(String paramName, String paramValue);
    public void convert() throws ConverterException;
}
```

To create a converter class, you must implement these methods. In your code, you create a converter using the ConverterFactory:

```
ConverterFactory cf = ConverterFactory.getConverterFactory();
Converter converter = cf.getConverter(String sourceMsgType,
    String sourceMsgCategory, String targetMsgCategory);
```

The static *getConverter()* method of the ConverterFactory class uses several parameters to identify which Converter class should be instantiated. It reads from the **MessageMap.xml** using the source and target message categories together with the message types to determine which Converter class must be used. Once created, the converter converts from a source document to a target document:

```
converter.setSource(srcDoc);
converter.setTarget(targetDoc);
converter.convert();
```

Note that the input and ouput to the conversion process can either be documents or streams.

Support for Lookup Codes

The Comergent eBusiness System uses lookup codes to provide a mechanism for maintaining and displaying locale-specific strings to users. For each lookup type,

you can define one or more lookup codes, and for each lookup code, you can define a string for each supported locale. See the *Comergent eBusiness System Reference Guide* for more information.

What lookup support does the Comergent eBusiness System provide?

The Comergent eBusiness System has the capability of automatically providing lookups between code values and their corresponding strings and from lookup code strings to code values.

If the "code" DsElement is set, then the "string" is automatically populated from the lookup cache. If the "string" value is set, then the "code" is looked up using the string value.

Are string values localized?

Yes. For a code-to-string lookup, the mechanism uses the user's locale to determine which string value to use. For a string-to-code lookup, the mechanism uses the user's locale when searching on a string value to find a corresponding code.

How do I define a code to string mapping?

Code-to-string relationships are defined in the **DsDataElement.xml** schema file. If both of the "code" and "string" DsDataElements are then used in a data object, then the code-to-string mapping is handled automatically.

The following is an example of a DataElement code-string pair.

```
<DataElement Name="OrderStatus" Description="Order Status"
    DataType="LONG" MaxLength="20" LookupType="OrderStatus"
    LookupString="OrderStatusString"/>
<DataElement Name="OrderStatusString" Description="Order Status"
    DataType="STRING" MaxLength="260" LookupType="OrderStatus"
    LookupCode="OrderStatus"/>
```

Are lookups performed for XML messages?

Yes. If a dataobject used for messaging contains a code-string pair, then the string value will automatically be used to look up the code.

How is the lookup cache loaded?

The lookup cache is loaded at system startup.

CHAPTER 4 — *Platform Modularity*

Release 7.0.2 of the Comergent eBusiness System has undergone the following architectural changes designed to make implementations easier to customize and upgrade:

- "Platform Modules" on page 36

This chapter describes the following modules:

- "Data Services" on page 38
- "Event Service" on page 38
- "Help" on page 38
- "Initialization Service" on page 38
- "Logging" on page 40
- "Message Type Entitlement" on page 43
- "Object Manager" on page 44
- "Preferences Service" on page 44
- "Session Reconnect Service" on page 45
- "Tag Libraries" on page 45
- "Thread Management" on page 45

- "Utils" on page 46
- "XML Message Converter" on page 46
- "XML Message Service" on page 46
- "XML Services" on page 47

Overview

The motivation to make architectural changes to the Comergent eBusiness System platform are to build the platform in a more modular way, so that changes and upgrades to the platform can be made more quickly and simply, and so that the modules can be re-used to support different products built using them.

By providing a means of delivering platform functionality in platform modules and by requiring that modules make calls to other modules only through their external interfaces, the following benefits are achieved:

- It is easier to compartmentalize the functionality of applications.
- It is easier to understand and manage the dependencies between parts of the Comergent eBusiness System.
- It is easier to contain the customizations to single modules and understand what effect changes made in a module have on the whole system.
- Modules can be more easily upgraded independently of each other, minimizing the effect that an upgrade may have.
- Upgrades to modules that have not been customized will not affect customizations made in other modules.
- New functionality can be delivered in the form of a module that may be dropped into an existing deployment of the Comergent eBusiness System.

Platform Modules

The Comergent eBusiness System platform is developed as a set of interdependent modules that conform to a common organizational structure. In general, each platform module corresponds to a functional component of the Comergent eBusiness System such as a service or a component of the Comergent eBusiness System platform. The platform modules provide a Java API to other modules. Some modules provide a set of "helper" classes which are used by a number of other modules.

In general, each platform module has the following structure:

- Java classes: organized into the following trees. At build time, the directories for the module is assembled in to a single JAR file.

    - com.comergent.api.*module*: external API interfaces: used by other modules to access functionality provided by the module. In general, when one module makes a call to another module's class, it must do so through the other module's external API. This is the com.comergent.api package for the module.

    - com.comergent.*module*: implementation classes: the implementation of the external API interfaces. When another module makes a call to the module's external API, then the actual classes used are the implementing classes of the module's interface. The implementation packages may include internal classes: used by the implementation classes, but not exposed to outside world and not part of the supported Javadoc.

- Configuration files specific to the module such as properties files. These are intended to live in the class hierarchy so that they can be referenced through *getResource()* calls.

Module Interfaces

Each platform module must provide an external interface so that all calls to Java classes and interfaces within the module are invoked through the interface. This external interface provides a comprehensive set of Javadoc pages so that writers of other modules can use the external interface reliably and easily.

The external interfaces are organized under the following main packages:

- com.comergent.api: this package has all the external APIs supported by the modules. These are organized by module: com.comergent.api.converter, com.comergent.api.logging, and so on.

Invoking Interfaces

You can invoke an interface from a Java class by casting any object or child interface to the interface and then invoke any method that the interface declares. Each module uses one or other of these techniques, but not both. As you work on an existing module or create a new one, be consistent in how you invoke the interfaces. It will make it easier for your colleagues to work on the same module.

In general, you should always try to work with interfaces provided by the com.comergent.api packages: these are the interfaces that the platform modules

will support from one release to the next, even though the underlying implementations of the interfaces may change.

Platform Module Descriptions

This section provides a brief description of the purpose of each platform module and examples of its use.

Data Services

This module provides a re-packaging and clean-up of the existing data services functionality. Its API has been moved out to a separate com.comergent.api.dataservices package. Data services now uses the same preferences mechanism as the rest of the Comergent eBusiness System to manage its properties. Connection pooling has been unified into one pool, and is tunable. Pagination has been updated, and no longer relies on pagination files being written to the file system.

Event Service

This module provides the classes used by the EventBus and Events. See CHAPTER 9, "Events" for more information on how to use events.

Help

This module provides the ComergentHelpBroker class: this is a simple wrapper class to the ServletHelpBroker class of the JavaHelp 2.0 implementation. See "Online Help" on page 247 for more information.

Initialization Service

The Initialization module provides the Initialization service. This is a package that helps you initialize the Comergent eBusiness System using a consistent framework of classes and methods.

The Initialization Manager provides a focal point in which:

- Initialization tasks can be defined
- Policy on failed initialization can be enforced
- Configuration fragments can be aggregated

The Initialization Manager main responsibility is to act on a list of initialization tasks in a well-defined and predictable manner. That implies an ordered list which:

- either, can be defined programatically

- or, can be specfied as an XML-format file

The following code extract provides a typical example of using the InitManager class.

```
InitManager initManager = InitManager.getInitManager();
try
{
    String resourceName = args[0];
    initManager.init(resourceName);
    // or programatically created
    //List modules = initModules();
    //ResourceLocator resourceLocator = createNewResourceLocator();
    //initManager.init(modules, resourceLocator);
}
catch (InitManagerException ime)
{
    log.error(ime, ime);
    System.exit(1);
}
// Initialization completed. OK to go on //
...
```

You can specify the initialization process using an configuration file. Here is a sample file:

```
<?xml version="1.0" encoding="UTF-8"?>
<initializationManager>
<resourceLocator>
    <path>/a/b/c</path>
    <path>.</path>
    <path>CLASSPATH</path>
    </resourceLocator>
<module name="ObjectManager"
    initClass="com.comergent.objectManager.InitHelper>
    <config name="Preferences">
    /com/comergent/objectManager/preferences.xml
    </config>
    <init-param name="param0">param0Value</init-param>
</module>
<module name="module1" initClass="com.comergent.module1.InitHelper>
    <config name="ObjectManager">
        /com/comergent/module1/objectMap.xml
    </config>
    <config name="MessageTypes">
        /com/comergent/module1/messageTypes.xml</config>
    <config name="Preferences">
        /com/comergent/modules1/preferences.xml
    </config>
    <init-param name="param1">param1Value</init-param>
```

```
</module>
<module name="module2" initClass="com.comergent.module2.InitHelper>
    <config name="ObjectManager">
        /com/comergent/module2/objectMap.xml
    </config>
    <config name="MessageTypes">
        /com/comergent/module2/messageTypes.xml
    </config>
    <config name="Preferences">
        /com/comergent/modules2/preferences.xml
    </config>
    <init-param name="param2">param2Value</init-param>
</module>
<!-- it is allowable to have no initClass -->
<module name="custom1" >
    <config name="ObjectManager">
        /com/comergent/module1/overlay/objectMap.xml
    </config>
</module>
</initializationManager>
```

In this example, when the following method is called by the Initialization Manager:

```
com.comergent.objectManager.InitHelper.init(initParams,
    configFragments, resourceLocator)
```

the following information is available:

- initParams has a list of key-value pairs: param0-param0Value

- configFragments has a list of:

    - /com/comergent/module1/objectMap.xml

    - /com/comergent/module12/objectMap.xml

- resourceLocator can find the resource along the path of: /a/b/c, current, and the current classpath.

Logging

This module provides access to the logging service used to record activity in the Comergent eBusiness System. Its property file, **log4j.properties**, is used to configure the behaviour of the logging service. The module is based on the log4j open source project and uses the same syntax for its configuration as follows:

Log4j has the following main components: *loggers*, *appenders*, and *layouts*. These three types of components work together to enable developers to log messages according to message type and level, and to control at runtime how these messages are formatted and where they are reported.

Configuration

You configure the logging platform module using the log4j.properties configuration file by specifying the properties of its loggers, appenders, and layout. For example, the following snippet is used to configure the root logger and the CMGT appender:

```
# Set root category priority
#log4j.rootCategory=info, CMGT
log4j.rootCategory=info, STDOUT
#log4j.rootCategory=info, CMGT, RTS

### START - CMGT
# CMGT appender
log4j.appender.CMGT=com.comergent.logging.ComergentRollingFileAp-
pender
#log4j.appender.CMGT=com.comergent.logging.ComergentDailyRolling-
FileAppender

#log4j.appender.CMGT.layout=org.apache.log4j.PatternLayout
log4j.appender.CMGT.layout=com.comergent.logging.ConversionPattern

# The log format defaults to the "classic" format. This format is
# recommended for actual deployment to allow a log analyzer to
# work correctly.
log4j.appender.CMGT.layout.ConversionPattern=%d{yyyy.MM.dd
HH:mm:ss:SSS} Env/%t:%p:%c{1} %m%n
```

Loggers

Loggers are named entities. Logger names are case-sensitive and they follow the hierarchical naming rule: a logger is said to be an ancestor of another logger if its name followed by a dot is a prefix of the descendant logger name. A logger is said to be a parent of a child logger if there are no ancestors between itself and the descendant logger.

For example, the logger named "com.foo" is a parent of the logger named "com.foo.Bar". Similarly, "java" is a parent of "java.util" and an ancestor of "java.util.Vector". This naming scheme should be familiar to most developers.

The root logger resides at the top of the logger hierarchy. It is exceptional in two ways:

• It always exists;

• It cannot be retrieved by name.

Invoking the class static *Logger.getRootLogger()* method retrieves it. All other loggers are instantiated and retrieved with the class static

Logger.getLogger(String name) method. This method takes the name of the desired logger as a parameter. For example:

```
private static final org.apache.log4j.Logger log =
   org.apache.log4j.Logger.getLogger(PriceCheckAPI.class);
log.debug("got current date: " + date);
```

Loggers may be assigned levels. The set of possible levels, that is DEBUG, INFO, WARN, ERROR and FATAL are defined in the org.apache.log4j.Level class. If a given logger is not assigned a level, then it inherits one from its closest ancestor with an assigned level. More formally:

Level Inheritance: the inherited level for a given logger, is equal to the first non-null level in the logger hierarchy, starting at the logger and proceeding upwards in the hierarchy towards the root logger.

To ensure that all loggers can eventually inherit a level, the root logger always has an assigned level.

Appenders

The ability to selectively enable or disable logging requests based on their logger is only part of the picture. More than one appender can be attached to a logger.

The addAppender method adds an appender to a given logger. Each enabled logging request for a given logger will be forwarded to all the appenders in that logger as well as the appenders higher in the hierarchy. In other words, appenders are inherited additively from the logger hierarchy. For example, if a console appender is added to the root logger, then all enabled logging requests will at least print on the console. If in addition a file appender is added to a logger, then enabled logging requests for the logger and its children will print on a file and on the console. It is possible to override this default behavior so that appender accumulation is no longer additive by setting the additivity flag to false.

The rules governing appender additivity are summarized below:

- The output of a log statement of logger C will go to all the appenders in C and its ancestors. This is the meaning of the term "appender additivity".

- However, if an ancestor of logger has the additivity flag set to false, then logger's output will be directed to all its appenders and its ancestors up to and including the ancestor, but not the appenders in any of the ancestors the ancestor.

- Loggers have their additivity flag set to true by default.

Layouts

Sometimes, you may wish to customize not only the output destination but also the output format. This is accomplished by associating a layout with an appender. The layout is responsible for formatting the logging request according to your wishes, whereas an appender takes care of sending the formatted output to its destination. The PatternLayout, part of the standard log4j distribution, lets you specify the output format according to conversion patterns similar to the C language printf function.

For example, the PatternLayout with the conversion pattern:

```
%r [%t] %-5p %c - %m%
```

will output something like this:

```
176 [main] INFO PriceCheckAPI - got current date: 10/22/2005.
```

The first field is the number of milliseconds elapsed since the start of the program. The second field is the thread making the log request. The third field is the level of the log statement. The fourth field is the name of the logger associated with the log request. The text after the "-" is the message of the statement.

Message Type Entitlement

This module provides the service that checks the entitlement of users to invoke message types.

The interfaces are defined in the com.comergent.api.dispatchentitlement package. This package contains factory classes, interfaces, and exceptions needed for the service. The implementation classes are in the com.comergent.dispatchentitlement package.

The main entry point for this module is the class EntitlementRepository. An instance of this class is obtained from the EntitlementFactory class. Applications can create named instances of the the EntitlementRepository class. Named instances will facilitate unit testing, and may be useful for alternative deployment environments.

An application needing to specify dispatch rules or other message type entitlement objects will execute logic similar to the following:

```
import com.comergent.api.dispatchentitlement.EntitlementRepository;
import com.comergent.api.dispatchentitlement.EntitlementFactory;
import javax.xml.dom.Document;
…
Document document = ...;
…
```

```
EntitlementRepository repository =
    EntitlementFactory.getEntitlementRepository();
repository.setRules(document);
```

Object Manager

This module provides the classes used to instantiate objects: see "ObjectManager and OMWrapper Classes" on page 22 for details.

Preferences Service

The Preferences module is used to retrieve and set configuration properties used by the Comergent eBusiness System. You can retrieve properties along these lines:

```
private static final Preferences prefs =
    Preferences.getPreferences(MyClass.class);
// implict scope of "com.comergent.apps.module.MyClass"
int max = prefs.getInt("PromotionManager.maxValue", 100);
int min = prefs.getInt("PromotionManager.minValue", 1);
```

The second parameter in the *getInt()* calls specify the value to return if no property with that name is found. The configuration file in which the property is defined is assumed to be on the classpath: for example in the file **com.comergent.apps.module.Preferences.xml**. If the XML properties file is read in using the Preferences service, then make sure that the XML file uses the Comergent root element. For example:

```
<Comergent>
    <PromotionManager>
        <maxValue>50</maxValue>
        <minValue>20</minValue>
    </PromotionManager>
</Comergent>
```

You can ensure that the Preferences service is used to initialize the properties by customizing the **WEB-INF/properties/init.xml** configuration file by adding an element along these lines:

```
<module name="PromotionMgr">
    <config name="Preferences">
        com/comergent/reference/apps/mktMgr/controller/Init.xml
    </config>
</module>
```

The Preferences class provides methods to get and put property values. For example:

```
prefs.putInt("PromotionManager.maxValue", 25);
prefs.putObject("currentShoppingCart", cartBean);
```

When using the *putObject()* method, the object must meet the requirements of the XMLEncoder API: essentially, that the object's fields must provide getter and setter methods.

Session Reconnect Service

This module is used to manage sessions that are intended to be created and used by applications that connect remotely to the Comergent eBusiness System.

Tag Libraries

The tag libraries provided by the Comergent eBusiness System are produced as a platform module. They are documented in CHAPTER 30, "Comergent Tag Library" and CHAPTER 31, "Comergent Internet Commerce Tag Library", and in the accompanying Javadoc.

Thread Management

This module provides a centralized facility for handling threads: their creation, obtaining their status, and re-use. In general, an application developer will no longer have to invoke:

```
Thread t = new Thread(new MyRunnable());
```
Instead, having a centralized facility will allow you to:

- Pool and re-use thread when appropriate

- Track all running threads to help provide better accounting for CPU and resource usage.

- Provide simple status reporting (scoreboard strategy: central shared location where running thread can report its status).

- Provide simple aborting and interrupt signal via *Thread.interrupt()* invocations to allow long running (but looping) thread to quit early.

The module provides the following functionality:

1. Transparently provide pooling and re-use of thread.

2. For administrative functionality, provide means to query all running threads tracked by the thread manager.

3. For user of thread service, provide means to report current thread status to a common scoreboard.

4. Provide guidance to following simple loop or check interrupted status protocol to allow a long running or looping thread to quit early.

5. Provide a timer facility to allow running thread to be notified when a timer expired. This can be used to implement a simple time-out or timeshare policy.

API and Usage

The API will continue to follow the Runnable() pattern: the application obtains a Thread-like object and use it to execute.

```
Excutor executor = ExecutorFactory.getPooledExecutor();
executor.execute(new MyComergentRunnable());
```

Utils

This module provides utility classes that can be used by other platform modules and applications.

XML Message Converter

This module provides a facility for converting XML documents from one message category (family and version) to another. The package name for the API is com.comergent.api.converter and com.comergent.converter for the implementation classes.

The API package includes:

*   ConverterFactory: this is the Factory class to create converters.

*   Converter: this is the class that converts a document from one message category to another. It can take either documents or streams as the source and targets for conversion.

See "Converter Classes" on page 32 for more information.

XML Message Service

This module is used to create and post outbound messages as XML documents. The API includes MsgContext interface, MsgService interface, MsgServiceFactory class, and theMsgServiceException classes in the com.comergent.api.msgservice package and the implementation classes are in the com.comergent.msgservice package.

The MsgService interface contains a generic *service()* method to post a databean and an XML document as specified in the message context.

The general usage pattern is as follows:

1. create a MsgContext instance using the MsgContextFactory;

2. set appropriate attributes on the context object;

3. create a MsgService instance for the target message family;

4. post a message by invoking the service method with a data bean and message context.

For example:

```
MsgContext ctx = new MsgContext();
ctx.setMessageType("ERPOrderCreateRequest");
ctx.setURL("http://www.server.com");
ctx.setMessageCategory("ERPOrderCreateRequest");
ctx.setContentType("text/xml");
ctx.setRemoteUser(username);
ctx.setRemotePassword(password);
MsgService msgService =
    MsgServiceFactory.getMsgService(ctx.getMessageCategory());
resultBean = msgService.service(requestBean, ctx);
```

XML Services

This module encapsulates functionality for XML parsing, XSL transformation, DOM wrappers, and utility classes.

CHAPTER 5 *Bizlets*

Using Bizlets for Message Processing

The Comergent eBusiness System makes use of a bizlet framework for handling inbound XML messages posted to the system. This replaces the use of BLCs that earlier releases of the Comergent eBusiness System used. This framework can be used to expose functionality of the Comergent eBusiness System to client applications through the combination of HTTP and XML. See "Example Bizlet Usage" on page 53 for an example of using bizlets in this way.

Each inbound message has a message type such as "OrderChangeRequest" declared as the MessageType element of its MessageHeader element. When the message is posted into the system (typically using the http://*<machine:port>*/Comergent/msg/matrix URL), the value of the message type element determines that the appropriate Bizlet processes the message.

The content type of the inbound message must correspond to a content type declared in the **MessageCrackerMap.xml** configuration file. For example:

```
<URLExt Name="integrator">
    <ContentType Name="application/x-icc-xml">
        <MessageCrackerImpl>
            com.comergent.dcm.messaging.ComergentMessageCrackerEx
        </MessageCrackerImpl>
        <ControllerImpl>
```

```
            com.comergent.dcm.messaging.MessagingController
        </ControllerImpl>
        <Request>com.comergent.dcm.messaging.XMLRequest</Request>
        <Response>com.comergent.dcm.messaging.XMLResponse</Response>
    </ContentType>
</URLExt>
```

Standard content types are "text/xml", "application/xml", and "application/
x-icc-xml".

> **Note:** Do not use encoded content types such as "application/
> x-www-form-urlencoded" because the servlet container will attempt to
> unencode the message before the message cracker.

Bizlet Interfaces and Implementation

Each Bizlet that you create should be defined as an interface: for example, the
following is the interface for the OrderBizlet used to process inbound Order
processing messages:

```
package com.comergent.apps.orderMgmt.bizlet;

public interface OrderBizlet extends Bizlet
{
    public OrderBean createOrder(OrderBean orderBean)
        throws ICCException;
    public OrderBean createOrderEx(OrderBean orderBean)
        throws ICCException;
    public OrderBean changeOrder(OrderBean orderBean)
        throws ICCException;
    public OrderBean cancelOrder(OrderBean orderBean)
        throws ICCException;
}
```

Currently beans are the only supported parameters for the interface methods.

> **Note:** The naming convention for the parameters is that you must use the
> same name as the bean with first letter being in lower case.

Bizlet interfaces must be created in packages whose last component is "bizlet": for
example com.comergent.apps.orderMgmt.bizlet. This ensures that the SDK
generates the IDL files when the project target is run.

You should create a corresponding implementation class that implements the bizlet
methods. Update the **ObjectMap.xml** file so that when an instance of the bizlet
class is required an appropriate object is returned. For example, if OrderBizletImpl
implements the OrderBizlet interface, then add the following to **ObjectMap.xml**:

```
<Object ID="com.comergent.apps.orderMgmt.orders.bizlet.OrderBizlet">
```

```
    <ClassName>
        com.comergent.apps.orderMgmt.orders.bizlet.OrderBizletImpl
    </ClassName>
</Object>
```

Each implementation class should extend the AbstractBizlet class (which includes a default definition of the *init()* method) and implement its appropriate interface. For example, here is a fragment of the OrderBizletImpl source code:

```
public class OrderBizletImpl extends AbstractBizlet
    implements OrderBizlet
    {
     /**
      * The BizletFactory will call this method upon instantiation,
      * and Bizlet session maintains the previous session state.
      */
    public void init(BizletSession state)
    {
        // initialization code goes here
    }

    /**
      * Create an order that corresponds to the incoming order
      * @param    OrderBean      orderBean incoming order
      * @return   OrderBean       created order
      * @exception    BizletException
      */
    public OrderBean createOrder(OrderBean orderBean)
        throws BizletException
    {
        // Business logic code goes in here
    }
```

Logging the Inbound XML Messages

You can capture the inbound XML message for logging purposes using code along these lines:

```
XMLRequest xReq = (XMLRequest) ComergentAppEnv.getRequest();
XMLRequestAccessor xAcc =
    (XMLRequestAccessor) xReq.getParameterAccessor();
```

To retrieve messages before they are converted, use:

```
ComergentDocument comergentDocument = xAcc.getInboundMessage();
```

To retrieve messages after they have been converted, use:

```
ComergentDocument comergentDocument = xAcc.getMessage();
```

BizletInvoker Classes

Bizlets are invoked indirectly using BizletInvoker classes. The Invoker classes manage the method invocation to ensure that the correct parameters are passed to the method. By default, these Invoker classes take the same name as the Bizlet interface, but with "Invoker" appended to the class name. For example, OrderBizlet and OrderBizletInvoker.

Invoker classes are generated classes using the following process.

1. A target (typically doGenerateIDL) identifies all interfaces that extend the Bizlet interface. For each such interface, it generates an interface definition language (IDL) file.

2. For each IDL file, a BizletInvoker class is generated: each BizletInvoker implements the corresponding Bizlet interface.

3. You should declare each BizletInvoker class in the **ObjectMap.xml** file.

    ```
    <Object ID="com.comergent.apps.orderMgmt.orders.bizlet.-
        OrderBizletInvoker">
        <ClassName>
        com.comergent.apps.orderMgmt.orders.bizlet.-
            OrderBizletInvokerImpl
        </ClassName>
    </Object>
    ```

    A standard Invoker class is automatically generated for each bizlet, and unless you need to perform some special processing as part of invoking the bizlet, you should be able to use the generated Invoker classes unchanged.

    Note that a naming convention will ensure that if no BizletInvoker class is declared, then the BizRouter will invoke a class whose name is the name of the declared Bizlet appended with "Invoker".

BizletSession Classes

BizletSession classes provide session-oriented state information to the bizlet classes. The naming convention for the session object is to append "Session" to the Bizlet name (for example, OrderBizletSession). Use BizletSession classes to set and retrieve any state that has session scope. When the Bizlet is invoked, its initialization method is called and you can provide initialization information:

```
public void init(BizletSession state)
{
    // initialization code goes here
}
```

Invoking Bizlets

Bizlets are invoked by mapping message types to bizlet methods using the BizletMapping elements of **MessageTypes.xml** files. For example:

```
<MessageType Name="OrderChangeRequest">
    <ControllerMapping>
        com.comergent.dcm.bizlet.BizRouter
    </ControllerMapping>
    <BizletMapping>
        com.comergent.apps.orderMgmt.orders.bizlet.OrderBizlet.-
            changeOrder
    </BizletMapping>
</MessageType>
```

The ControllerMapping element ensures that the correct BizRouter class is used to invoke the correct BizletInvoker class and method. Note that the interface is referenced in the BizletMapping element and that it provides the name of the method that should be invoked to process this message type.

Example Bizlet Usage

In this section, we provide an example of how to use a bizlet to access Comergent eBusiness System functionality. We create a bizlet that can provide support to create, delete, or view shopping carts. Once you have done the work described below, then your client application can simply HTTP post XML messages as described below, and then process the reply once it is received.

You must make sure that the content type used to post the inbound message is declared in the **MessageCrackerMap.xml** configuration file.

Message Types

We must support the following mesage types:

•   ShoppingCartXMLCreateRequest

•   ShoppingCartXMLDeleteRequest

•   ShoppingCartXMLLookupRequest

We want to create the messages along these lines:

•   **ShoppingCartXMLCreateRequest.xml**: we use this to create a new shopping cart.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Comergent>
```

```
            <MessageHeader>
                <MessageType>ShoppingCartXMLCreateRequest</MessageType>
                <MessageVersion>4.0</MessageVersion>
                <MessageID/>
                <SessionID/>
            </MessageHeader>
            <RemoteUser>
            <UserLogin>cchen</UserLogin>
            <UserFullName/>
            <UserAuthenticator>cchen</UserAuthenticator>
            </RemoteUser>
            <OrderInquiryList type="BusinessObject">
                <Name>Toro Demo</Name>
                <LineItemList>
                    <LineItem>
                        <SKU>MX-LNXA</SKU>
                        <Quantity>101</Quantity>
                    </LineItem>
                </LineItemList>
            </OrderInquiryList>
        </Comergent>
```

- **ShoppingCartXMLDeleteRequest.xml**: we use this to delete a shopping cart.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Comergent>
    <MessageHeader>
        <MessageType>ShoppingCartXMLDeleteRequest</MessageType>
        <MessageVersion>4.0</MessageVersion>
        <MessageID/>
        <SessionID/>
    </MessageHeader>
    <RemoteUser>
        <UserLogin>cchen</UserLogin>
        <UserFullName/>
        <UserAuthenticator>cchen</UserAuthenticator>
    </RemoteUser>
    <OrderInquiryList type="BusinessObject">
        <ShoppingCartKey>600568</ShoppingCartKey>
    </OrderInquiryList>
</Comergent>
```

- **ShoppingCartXMLLookupRequest.xml**: we use this to view an existing shopping cart.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Comergent>
    <MessageHeader>
        <MessageType>ShoppingCartXMLLookupRequest</MessageType>
```

```
            <MessageVersion>4.0</MessageVersion>
            <MessageID/>
            <SessionID/>
        </MessageHeader>
        <RemoteUser>
            <UserLogin>cchen</UserLogin>
            <UserFullName/>
            <UserAuthenticator>cchen</UserAuthenticator>
        </RemoteUser>
        <OrderInquiryList type="BusinessObject">
            <ShoppingCartKey>600568</ShoppingCartKey>
        </OrderInquiryList>
    </Comergent>
```

You must create DTDs for the requests and their replies, and copy them to the
WEB-INF/messages/ directory. For example, these are the DTDs for
ShoppingCartXMLCreateRequest.dtd and
ShoppingCartXMLCreateReply.dtd, the DTDs for the request to create a
shopping cart:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
    ShoppingCartXMLCreateRequest
    Document Type Declaration (DTD)
    Version 4.0 Comergent
    14-June-2004
    Authors:
      Comergent
      Contact: (650) 232-6000
                support@comergent.com
-->
<!ENTITY % MessageHeader SYSTEM "MessageHeader.dtd">
%MessageHeader;
<!ENTITY % RemoteUser SYSTEM "RemoteUser.dtd">
%RemoteUser;
<!ENTITY % OrderInquiryList SYSTEM "../bizobjs/OrderInquiryList.dtd">
%OrderInquiryList;
<!ELEMENT Comergent (MessageHeader, RemoteUser, OrderInquiryList)>
```

and

```
<?xml version="1.0" encoding='UTF-8' ?>

<!--
    ShoppingCartXMLCreateReply
    Document Type Declaration (DTD)
    Version 4.0 Comergent
    14-June-2004
    Authors:
      Comergent
```

```
        Contact: (650) 232-6000
                    support@comergent.com
-->

<!ENTITY % MessageHeader SYSTEM "MessageHeader.dtd">
%MessageHeader;
<!ENTITY % ReplyHeader SYSTEM "ReplyHeader.dtd">
%ReplyHeader;
<!ENTITY % OrderInquiryList SYSTEM "../bizobjs/OrderInquiryList.dtd">
%OrderInquiryList;
<!ELEMENT Comergent (MessageHeader, RemoteUser, OrderInquiryList)>
```

Bizlets

We create the ShoppingCartBizlet interface as follows:

```
package com.comergent.apps.channelMgmt.bizlet;

import com.comergent.dcm.bizlet.Bizlet;
import com.comergent.dcm.util.ICCException;
import com.comergent.bean.simple.OrderInquiryListBean;

public interface ShoppingCartBizlet extends Bizlet
{
public OrderInquiryListBean lookupShoppingCart(OrderInquiryListBean
    orderInquiryListBean) throws ICCException;

public OrderInquiryListBean createShoppingCart(OrderInquiryListBean
    orderInquiryListBean) throws ICCException;

public OrderInquiryListBean deleteShoppingCart(OrderInquiryListBean
    orderInquiryListBean) throws ICCException;
}
```

We also create the corresponding implementation class:

```
/*
 * ShoppingCartBizletImpl.java
 * Copyright (c) 2004 Comergent. All rights reserved.
 */
package com.comergent.apps.channelMgmt.bizlet;

import com.comergent.dcm.bizlet.Bizlet;
import com.comergent.dcm.bizlet.AbstractBizlet;
import com.comergent.dcm.dataservices.DataBean;
import com.comergent.dcm.util.ICCException;
import com.comergent.bean.simple.OrderInquiryListBean;
import com.comergent.bean.simple.OrderInquiryListLineItemBean;
import com.comergent.api.apps.commerce.IInquiryList;
import com.comergent.api.apps.commerce.CommerceAPI;
```

```
import com.comergent.api.apps.commerce.IInquiryListTypes;
import com.comergent.api.apps.pricingMgr.APIPriceListAttribute;

import java.util.Iterator;

public class ShoppingCartBizletImpl extends AbstractBizlet
    implements ShoppingCartBizlet
{
public OrderInquiryListBean lookupShoppingCart(OrderInquiryListBean
    orderInquiryListBean) throws ICCException
{
    IInquiryList list = CommerceAPI.getFactory(
        IInquiryListTypes.ORDER_INQUIRY_LIST).getInquiryList(
        orderInquiryListBean.getShoppingCartKey(), true);
    return (OrderInquiryListBean) list.getDataBean();
}

public OrderInquiryListBean createShoppingCart(OrderInquiryListBean
    orderInquiryListBean) throws ICCException
{
    String name = orderInquiryListBean.getName();
    if ((name == null) || (name.length() == 0))
    {
        name = "New Cart";
    }
    Long currencyCode = orderInquiryListBean.getCurrencyLookupCode();
    if (currencyCode == null)
    {
        currencyCode = APIPriceListAttribute.getDefaultCurrencyCode();
    }
    Long customerTypeCode =
        orderInquiryListBean.getCustomerTypeCode();
    if (customerTypeCode == null)
    {
        customerTypeCode =
            APIPriceListAttribute.getDefaultCustomerType();
    }
    IInquiryList list = CommerceAPI.getFactory(
        IInquiryListTypes.ORDER_INQUIRY_LIST).createNewInquiryList(
        name, currencyCode, customerTypeCode);
    Iterator it = orderInquiryListBean.getLineItemIterator();
    while (it.hasNext())
    {
        OrderInquiryListLineItemBean oilLineItemBean =
            (OrderInquiryListLineItemBean) it.next();
        list.addLineItem(oilLineItemBean.getSKU(),
            oilLineItemBean.getQuantity());
    }
    list.save();
```

```
    return (OrderInquiryListBean)list.getDataBean();
}

public OrderInquiryListBean deleteShoppingCart(OrderInquiryListBean
    orderInquiryListBean) throws ICCException
{
    IInquiryList list = CommerceAPI.getFactory(
        IInquiryListTypes.ORDER_INQUIRY_LIST).getInquiryList(
        orderInquiryListBean.getShoppingCartKey(), true);
    list.delete();
    return (OrderInquiryListBean)list.getDataBean();
}
}
```

You must add the following element to the **ObjectMap.xml** to map the interface to the implementation class:

```
<Object
    ID="com.comergent.apps.channelMgmt.bizlet.ShoppingCartBizlet">
    <ClassName>
        com.comergent.apps.channelMgmt.bizlet.ShoppingCartBizletImpl
    </ClassName>
</Object>
```

You must add the message types to the appropriate MessageTypes.xml file:

```
<MessageType Name="ShoppingCartXMLCreateRequest">
    <ControllerMapping>
        com.comergent.dcm.bizlet.BizRouter
    </ControllerMapping>
    <BizletMapping>
        com.comergent.apps.channelMgmt.bizlet.-
        ShoppingCartBizlet.lookupShoppingCart
    </BizletMapping>
</MessageType>

<MessageType Name="ShoppingCartXMLLookupRequest">
    <ControllerMapping>
        com.comergent.dcm.bizlet.BizRouter
    </ControllerMapping>
    <BizletMapping>
        com.comergent.apps.channelMgmt.bizlet.-
        ShoppingCartBizlet.createShoppingCart
    </BizletMapping>
</MessageType>

<MessageType Name="ShoppingCartXMLDeleteRequest">
    <ControllerMapping>
        com.comergent.dcm.bizlet.BizRouter
    </ControllerMapping>
```

```
<BizletMapping>
    com.comergent.apps.channelMgmt.bizlet.-
    ShoppingCartBizlet.deleteShoppingCart
</BizletMapping>
</MessageType>
```

Introducing Data Beans and Business Objects

This chapter presents a brief tutorial that demonstrates how you can use the Comergent eBusiness System to work easily with data beans and business objects. You can also consult the *Comergent eBusiness System Reference Guide* for more information on data beans and business objects.

> **Attention:** In Release 6.4 and later, the use of business objects is not supported. You should use data beans wherever possible. See CHAPTER 40, "Deprecated Concepts" for more information about business objects.

What are Data Beans?

A data bean is a data source-independent representation of a real-world entity in the Comergent eBusiness System. The Comergent eBusiness System uses an external schema (defined as a set of XML files) to define the structure of each type of data bean. For example, data beans are used as data structures for users, product inquiry lists, partners, products, and workspaces.

- Use the OMWrapper and ObjectManager classes to create instances of the DataBean classes. See "ObjectManager and OMWrapper Classes" on page 22 for more information.

- You can create a DataBean using the DataManager. Invoke the DataManager method *getDataBean(String beanName)* to create a DataBean of the named type. This method throws an InvalidBizobjException if no such DataBean class exists.

Note:	The use of this method is deprecated because it does not support extensions of the data object.

Lifecycle of a Data Bean

In general, the basic flow of working with a data object is:

1. Instantiate a data bean object using the OMWrapper class.

2. Add data to the bean by using the set methods to directly insert values into the data fields.

3. Persist the data bean to save the new data object to its data source for the first time.

4. Subsequently, you can retrieve the same data object by setting the value for key fields, and then performing a *restore()* on the data bean to retrieve the current data field values from its data source.

5. Perform any business logic required on the data bean. This may change the in-memory values of fields, but not the values stored in the data bean's data source.

6. Save the changes to the data bean by persisting the data bean to its data source.

7. Later, you may want to delete the data object if it is no longer in use.

8. Eventually, you may want to remove the data from the data source entirely by erasing the data object.

In the case of data objects whose underlying data source is a database, the following table summarizes the Java method calls and the corresponding SQL methods called.

TABLE 4. **Data Bean Lifecycle**

Step	Java Method	SQL Method
Instantiate data object	*OMWrapper.getObject()*	
Populate data fields	*setDataField()*	
Persist for the first time	*persist()*	INSERT
Retrieve data object	*restore()*	SELECT

TABLE 4. Data Bean Lifecycle

Step	Java Method	SQL Method
Business logic that updates field values	*getDataField()* *setDataField()*	
Save changes	*persist()*	UPDATE
Delete data object	*delete()*	UPDATE[a]
Erase data object	*erase()*	DELETE

a. The Delete operation updates the ACTIVE_FLAG column of the underlying database table row: it does not remove the record from the table.

Defining a Data Bean

Data beans are defined using an XML schema. Data beans provide accessor methods to get and set values of particular data fields. In general, you should use data beans when customizing Comergent eBusiness System applications.

Defining the Structure of a Data Object

Each data object must have a defined structure to enable the Comergent eBusiness System to create an instance of the data object. The structure of a data object is defined in its schema XML file: it specifies what fields the data object has and whether it has child objects.

Each data object corresponds to a Java class that extends the DataBean class. We refer to these as data bean classes. The data bean classes are generated automatically as part of the SDK merge process. When you generate the corresponding data bean class, it provides methods that access the fields and child data beans that are declared in the data object XML file.

You can change the definition of the XML schema and hence of data objects and their corresponding data bean classes by editing the XML schema files.

The **DsRecipes.xml** configuration file is used to link each data object and its data source. It also specifies whether the ordinality of the data object is "1" or "n". The data object file is used to specify the precise structure of the data object, and the **DsDataElements.xml** configuration file is used to specify the data type (LIST, LONG, STRING, and so on) of each element.

Extending Data Objects

When you define a data object with an XML schema file, you can declare that it extends another data object by using the Extends attribute. This capability is used in two ways:

- You can use one data object as the parent of several different extending data objects which all share a common set of data fields. For example, many data objects in the Comergent eBusiness System extend the C3PrimaryRW data object: this data object provides the basic OwnedBy and AccessKey data fields used to manage access control.

- You can customize a data object by creating a data object that extends it. By adding data fields to the extending data object, you can add attributes that you need to use as part of your customization. By using the ObjectManager, you can ensure that the extending data object is created when the system is called upon to create a data object of the extended type. Provided that existing code uses the ObjectManager to instantiate instances of the extended data object, then when this code is invoked, instances of the extending data object are created, but these still support the extended data object's interfaces, and so the existing code will continue to work.

The DataManager uses a *recipe* and a *data object* to determine the element structure of the data bean or business object and the location of the data source that provides the element values. When you change the definition of data objects or create new definitions, you must re-run the generateDTD and generateBean SDK targets to create and compile the DataBean classes. See CHAPTER 14, "Software Development Kit" for more details. See "Extending Data Objects" on page 76 for alternate ways to extend data objects.

Data Bean and Business Object Creation

The Comergent eBusiness System's ObjectManager and OMWrapper classes create data beans, and business logic classes and controllers process them. See "ObjectManager and OMWrapper Classes" on page 22 for more information.

Business logic classes are invoked by controllers: each controller is responsible for determining which business logic class (if any) must be created in response to a message and its message type.

The use of business objects and the BusinessObject class is deprecated. Where possible, you should use data bean classes, and use business objects only to maintain legacy code.

DataContext

The *restore()* method takes an instance of the DataContext class as a parameter. The DataContext class is used to specify information about the context in which the *restore()* operation is being performed. It can be used to specify the maximum number of results to be returned and for pagination the number of results on each

page. It can also be used to specify whether an access check should be performed on the results of the *restore()* operation. By default, an access check is performed.

For example, the following code snippet creates a DataContext, sets some context values, and then uses the context and a query to restore a data bean:

```
DataContext temp_DataContext = new DataContext();
temp_DataContext.setMaxResults(DsConstants.NO_LIMIT);
temp_DataContext.setNumPerPage(-1);
skuMappingListBean.restore(temp_DataContext, query);
```

When a DataContext object is initialized, it retrieves from the configuration files values of the DataServices.General.MaxResults and DataServices.General.NumPerCachePage element to set these parameters for the restore operation. By default, no limit is set on either. There are accessor methods available if the behavior of the DataContext needs to be modified. See the DataContext Javadoc for further information.

The DataContext class provides a *setCacheId(String cacheId)* method to support pagination: it identifies the particular cache being used.

What is the DataContext class?

The DataContext class is used to control the behavior of restore and persist operations.

What behavior can be controlled?

A DataContext instance can control the following:

- Whether ACL checking is performed.

- How many query results appear on a page.

- The maximum number of query results that will be processed.

- The use of multiple page sets per Data Bean type and Session.

How do I control ACL checking?

In certain special cases it may be useful to disable ACL checking. This can improve the performance of large requests, but it should only be done if access permissions have already been verified.

The DataContext class provides the following methods to control ACL checking on Data Bean restore and persist requests:

- void disableAccessCheck(): Turns off ACL checking for persist and restore operations.

- void enableAccessCheck(): Turns on ACL checking for persist and restore operations.

- boolean doAccessCheck(): Returns the current setting for ACL checking.

What are the Cache Id methods for?

The Cache Id methods allow an application to specify a unique identifier for pagination of result sets. This new capability allows an application to maintain multiple distinct result sets for a given Data Bean and Session.

If an application does not specify a Cache Id then a combination of Bean name and Session Id are used to identify the cache. In this case any subsequent attempt to restore the same Data Bean within the same session will overwrite any results.

The DataContext class provides the following methods to control Cache Id on Data Bean restore requests:

- void setCacheId(String cacheId): Sets a new cache id. This string is used in combination with the Bean name and session id to generate a unique identifier.

- String getCacheId(): Returns the current cache id (or null if it is not set).

How do Max Results and Num Per Page work?

The setting of Max Results determines the maximum number of records that can be retrieved during a restore. When that number is reached the request is freed.

The setting of Num Per Page determines how many records are saved in each result cache page. If the number found is less than Num Per Page, then no result cache is created.

Note that this combination of attributes now allow the application to retrieve a set of paginated results while still specifying a maximum number of records to retrieve.

The DataContext class provides the following methods to Max Results and Num Per Page on Data Bean restore and persist requests:

- void setMaxResults(int maxResults)

- int getMaxResults()

- void setNumPerPage(int numPerPage)

- int getNumPerPage()

How do I instantiate a DataContext instance?

A new DataContext instance is currently instantiated using the standard "new" mechanism:

```
DataContext dc = new DataContext();
```

What are the Default Settings for a new DataContext?

When "new DataContext()" is invoked, the attributes receive the following default values:

TABLE 5. **DataContext Default Values**

Attribute	Default Value
doAccessCheck	true
maxResults	DataServices.xml maxResults property
numPerPage	DataServices.xml numPerPage property
CacheId	null

List Data Beans

A special class of business objects are called *list data beans* and *list business objects*. You use these classes to manage a list of data objects of the same type. Whenever a data object element is declared with ordinality "n" in a Recipe element, then a list data bean is created. Access entitlements are still managed at the level of the singular business object.

Note:	Earlier versions of data objects defined ordinality in the data object definition file. Now it is the recipe file that determines the ordinality of a data object. In Version 6.0 data objects, the ordinality attribute is still used to declare child, reference, and included data objects.

In general, you do not need to create DataBeans for list data objects: they are created automatically. See "DataBean Classes" on page 22 for more information. They support automatically generated methods that return a list of the data objects. For example, the following code fragment demonstrates how to restore a list of users. A DataContext object identified by "context" and a DsQuery object identified as "query" are used to restrict the users returned by the *restore()* call:

```
UserListBean userList = (UserListBean)
    OMWrapper.getObject("com.comergent.bean.simple.UserListBean");
// Restore the list.
userList.restore(context, query);
// Return immediately if no results found.
```

```
if (userList.getUserCount() == 0)
{
    return;
}
// At least one user in list, so walk through the list of users
ListIterator userIterator = userList.getUserIterator();
while (userIterator.hasNext())
{
    UserBean user = (UserBean) userIterator.next();
    // Perform any business logic on each user.
}
```

Note the use of the DataContext and DsQuery parameters in the *restore()* method: these are used to manage how the query is executed against the Knowledgebase. See CHAPTER 22, "Data Services Guidelines" for more information about the use of the DsQuery class.

Note that there is a difference between how Release 4.1 DataBeans behave and Release 5.0 and higher list DataBeans behave:

• In Release 4.1, if you create a data object (say, *MyObject*) and a corresponding list data object (*MyObject*List), then the generated BizobjBean *MyObject*ListBean supports the *get*MyObject*ListVector()*. This method returns a Vector of objects that can be cast as the inner class *MyObject*List$*MyObject*.

• In Release 5.0 and higher, if you create a data object (say, *MyObject*) and a corresponding list business object (*MyObject*List), then the generated DataBean *MyObject*ListBean also supports the *get*MyObject*Iterator()* method. This method returns a ListIterator of objects that can be cast as the class *MyObject*.

See "Adding Functionality to an Application" on page 180 for an example of using a list data bean.

Application, Entity, and Presentation Beans

There are several main sorts of data beans used in the Comergent eBusiness System: data beans, application beans, entity beans, and presentation beans. This section describes the main differences between them.

• Data beans are the Java classes created automatically from the XML schema description of the business objects. Running the generateBean

SDK target generates the source code for each data bean. These beans comprise the com.comergent.bean.simple package.

Where possible, you should you use the *instanceof* command to determine the class of a data bean rather than querying for the business object type.

• Application beans are Java classes created to add functionality that simple beans do not support. For example, an application bean may provide extra methods that cannot be automatically generated, or it may combine two or more simple beans to pass data to a JSP page. The application beans are organized by application and each application has a package for its application beans whose name is com.comergent.apps.<*application name*>.bean

Application beans can be subclasses of simple beans, but more often they are Java classes that contain one or more simple beans as member variables.

For example, the com.comergent.appservices.productService.productMgr.BizProductBean application bean class is a Java class that contains a member variable that implements the com.comergent.bean.simple.IDataProduct interface. The BizProductBean application bean class delegates methods such as *getProductID()* to the com.comergent.bean.simple.IDataProduct member variable, but in addition it provides methods to retrieve a product's features, its supersession chain, and prices. Note the use of the IDataProduct interface rather than the ProductDataBean itself: this is an example of using a generated interface rather than the class. See "Generated Interfaces" on page 129 for more information on the generation and use of these interfaces.

By convention, if you create an application bean to wrap a data bean, then you must provide a method called *getDataBean()* that retrieves the data bean.

• Presentation beans are also used to pass data to JSP pages: typically, they differ from application beans in that they do not provide business logic. They may aggregate several data beans into a single class for ease of use, or provide formatting information. As with application beans, presentation beans must provide a method to provide access to the underlying data bean. For example, the IPresProduct interface provides the *getIRdProduct()* method: this returns the IRdProduct interface and you can downcast this to the underlying data bean or extended data bean if need be.

- Entity beans were used in prior releases of the Comergent eBusiness System. They performed the same role as application beans. Their use is deprecated.

Using Stored Procedures

You can make use of stored procedures to restore data objects. The name of the stored procedure is declared in the ExternalName element of the data object. See "Stored Procedures" on page 270 for more information.

When you define data objects, take care to specify the SourceType attribute. It can take the following values:

- "1": the underlying data source uses a table. This is the default value.

- "2": the underlying data source uses a stored procedure.

If no SourceType attribute is defined, then the default value means that a table is the underlying source type for the data object.

Data Bean Methods

In general, you should make use of the generated interfaces that each data bean provides: these organize the accessor and data methods to help you manage access to the data objects during their lifcycle. See "Generated Interfaces" on page 129 for more information. In addition, see "Data Bean Methods" on page 100 for information about the access-checking methods supported by data beans.

IData Methods

The IData interface has these important methods:

- *copyBean()*: this method can be used to copy the values of data fields from one bean to another. It takes one argument: this must be a bean that is either an instance of the same class or a sub-class of the bean invoking this method.

- *delete()*: this method marks the corresponding data object as deleted: the ACTIVE_FLAG column of the database table corresponding to this data object is set to "N" when the object is persisted. Note that you must call *persist()* after calling *delete()*: if you do not, then the deletion does not take effect.

- *erase()*: this method removes the database record corresponding to the business object. Note that removing records from database tables can lead to data integrity problems if other tables refer to keys that have been deleted. In general, you should use this method only if you can account for all usages of the record and its keys and can delete the corresponding records from other tables.

- *generateKeys()*: this method populates the key fields of the data bean. You can call this method without invoking *persist()*. By invoking this method, you can use the generated keys to create other objects that require the keys.

- *setDataContext()*: this method sets the data context so that *restore()* and *persist()* calls use the right values for parameters such as the number of results per page in a paginated data set. See "DataContext" on page 64 for more information on the DataContext class.

- *persist()*: this method saves the data in the data bean to its data source.

- *prune()*: this method is used to mark the bean for deletion in memory. Calling *restore()* after *prune()* has no effect on the bean's underlying data source.

- *restore()*: this method retrieves the data for the data bean from its data source. See "DataContext" on page 64 for information on the use of the DataContext class in the *restore()* method. See CHAPTER 22, "Data Services Guidelines" for information on using the DsQuery class to specify queries as part of the *restore()* operation.

- *update()*: this method updates the database record corresponding to this business object.

Note that any method calls that change state must be followed by a *persist()* call to actually make the change to the database record.

The IData interface also provides the methods, *isRestorable()* and *isPersistable()*, that check whether a data object may be restored or persisted respectively.

IRd and IAcc Interface Methods

The IRd interface provides the read-only accessor methods to the data object fields. The IAcc interface extends the IRd interface by adding the set accessor methods for each data field. Distinguishing between these two interfaces provides you with the ability to pass a read-only object to a client application or JSP page.

For example, suppose that in the Condition data object file, **Condition.xml**, a DataField element is specified as follows:

```
<DataField Name="ControlType"
    Writable="y" Mandatory="y"
    ExternalFieldName="CONTROL_TYPE"/>
```

Then, in the automatically-generated IRdCondition interface, there is a method called:

public Long getControlType()

In the automatically-generated IAccCondition interface, there is a method called:

public void setControlType(Long value) throws ICCException

The signatures of these accessor methods is determined by the corresponding DataElement definition in the **DsDataElements.xml** file:

```
<DataElement Name="ControlType" DataType="LONG"
    Description="Condition Control Type" MaxLength="20" />
```

Note:	If you set the Writable attribute of a data field to "n", then the corresponding *set*DataField*()* method is not generated.

Restoring and Persisting Data

These important operations may be performed on a data object: *delete()*, *persist()*, and *restore()*.

- By calling the *delete()* method on a data object, you mark this object as deleted, and no other application will retrieve this data object. The ACTIVE_FLAG column of the underlying database table has its value set to 'N'. Note that the data object data is not deleted from the data source. If the underlying database table for data object does not have an ACTIVE_FLAG column, then do not use the *delete()* method. You can still use the *erase()* method to remove such data objects from the Knowledgebase.

- When you *persist* a data bean, the Comergent eBusiness System saves the data held in the data object's DsElement tree to its external data source(s). Note that the Comergent eBusiness System manages both the update of existing data objects and the creation of new data objects with the *persist()* method.

- When you *restore* a data bean or business object the Comergent eBusiness System retrieves its data from its external data source(s). If no query

object is specified in the *restore()* method, then all of the data objects whose values in the key fields match those in the data bean are restored.

- Note that if you call *restore()* on a non-list data bean, then you should expect that its data is uniquely retrievable from the values set in its key fields. When the *restore()* call is issued, no check is performed to verify that only one record is retrieved, and so the first record retrieved will be used to populate the data bean. If no record is retrieved, then the *restore()* call throws an ICCException.

- When you call *restore()* on a list data bean, then you must usually specify a DsQuery. If no DsQuery is specified, then the restored list data bean will contain all the data beans of this type. See CHAPTER 22, "Data Services Guidelines" for more information about the DsQuery class.

restore() Method

This section provides description of the main forms of the DataBean *restore()* method.

```
public void restore(DataContext dataContext, DsQuery dsQuery)
```

The principal form of the *restore()* method. Use the dsQuery parameter to specify query to be executed by the restore operation. The dataContext parameter determines the maximum number of objects returned, and for pagination the number of results per page. Use the dataContext parameter to specify whether to check that the current user has the correct entitlements to perform this operation. By default, an access check is performed, so you have to override the access check if you do not want this to be done, using the *disableAccessCheck()* method.

```
public void restore(DataContext dataContext)
```

This is equivalent to calling *restore(dataContext, null)*.

Here is an example of using the DataContext and DsQuery classes together to manage the *restore()* call:

```
try
{
    DataContext dataContext = new DataContext();
    if (doAccessCheck == true)
    {
        dataContext.enableAccessCheck();
    }
    else
    {
        dataContext.disableAccessCheck();
    }
```

```
    dataContext.setNumPerPage(pageSize);
    DsQuery dsQuery = QueryHelper.newWhereClause("PartnerKey",
        DsConstants.EQUALS, partnerKey);
    LightWeightPartnerBean partnerBean =
    (com.comergent.bean.simple.LightWeightPartnerBean)
    com.comergent.dcm.util.OMWrapper.getObject(
        "com.comergent.bean.simple.LightWeightPartnerBean");
    partnerBean.restore(dataContext, dsQuery);
    QueryHelper.freeQuery(dsQuery);
    return partnerBean;
}
catch (ICCException e)
{
    throw (new ProfileMgrException(e));
}
```

persist() Method

This section provides description of the main forms of the DataBean *persist()* method.

```
public void persist(DataContext dataContext)
```

If the dataContext specifies that an access check should be performed, then this form of the *persist()* method performs an access check before performing the operation. If the user does not have the appropriate entitlement, then the operation is not performed.

Miscellaneous Methods

getBizObj() Method

If you want to retrieve a business object representation of the data object and its data, then you can invoke the *getBizObj()* method. This is useful if you want to display the internal structure of the object. For example:

```
BusinessObject bo = bean.getBizobj();
ComergentDocument doc = bo.serializeToXml();
doc.prettyPrint();
```

Note that this is now a deprecated method.

writeExternal() Method

Use this method to write out an XML representation of the data bean and its data.

Child Data Objects

Many data objects declare child data objects using the ChildDataObject element. For example, the ShoppingCart data object declares LineItem as a child data object as follows:

```
<DataObject Name="ShoppingCart" Extends="C3PrimaryRW"
    ExternalName="CMGT_CARTS" ObjectType="JDBC" Version="6.0">
...
<ChildDataObject Access="RWID" Name="LineItem">
    <Relationship CascadeDelete="y" CascadeErase="n"
        ChangeUpdatesParent="y">
        <JoinKeys>
            <JoinKey DstJoinField="ShoppingCartKey"
                SrcJoinField="ShoppingCartKey"/>
        </JoinKeys>
    </Relationship>
</ChildDataObject>
...
</DataObject>
```

Its Relationship element has attributes that describe how child objects should be managed when the parent is updated and whether to update the parent when a child is changed. The JoinKey elements describes how to restore the child data objects: typically, by specifying how values in the parent data object are used to set values in the child data object.

When the parent data bean is generated, it generates a method called *get*ChildDataObject*Iterator()* which returns an ListIterator object containing the child data beans. By iterating through the objects, you can examine each child data bean in turn and access its fields using the standard accessor methods.

For example, the ShoppingCartBean class supports the getLineItemIterator() method. The following lines of code demonstrate how to retrieve a field of a line item:

```
/*
shoppingCartBean is a ShoppingCartBean object that has already been
restored
*/
ListIterator lineItemIterator =
    shoppingCartBean.getLineItemIterator();
LineItemBean lineItemBean =
    (LineItemBean) lineItemIterator.getLineItemBean(0);
Long quantity = lineItemBean.getQuantity();
```

When a parent data object is restored, the child data objects are not restored. They are restored only when the application accesses the children as described above.

Extending Data Objects

It is common for any implementation of the Comergent eBusiness System to need to add data fields to data objects or to create data objects that extend existing data objects.

We recommend storing the additional data in a new database table. A new DataObject should then be defined that accesses the new table. Another new DataObject is then defined that extends the original DataObject by adding a new IncludeDataObject.

For example, suppose that you need to add a new data field to the Order data object to track "hosted" orders: orders that are placed at storefront partners. The extra data field is the partner key of the storefront partner. The recommended approach is as follows:

1. Create a new data object called HostedPartner that has exactly two fields: an OrderKey and a PartnerKey. Set it up to point to a two-column table: CMGT_ORDER_X_PARTNER with columns ORDER_KEY and PARTNER_KEY.

    ```xml
    <?xml version="1.0"?>
    <DataObject Name="HostedPartner"
        ExternalName="CMGT_ORDER_X_PARTNER" ObjectType="JDBC"
        Version="6.0">
        <KeyFields>
            <KeyField Name="OrderKey" ExternalName="ORDER_KEY"/>
            <KeyField Name="PartnerKey" ExternalName="PARTNER_KEY"/>
        </KeyFields>
        <DataFieldList>
            <DataField Name="OrderKey" ExternalFieldName="ORDER_KEY"
                Mandatory="n" Writable="y"/>
            <DataField Name="PartnerKey"
                ExternalFieldName="PARTNER_KEY"
                Mandatory="n" Writable="y"/>
        </DataFieldList>
    </DataObject>
    ```

2. Create a new data object called HostedOrder that extends Order. The **HostedOrder.xml** file looks like this:

    ```xml
    <?xml version="1.0"?>
    <DataObject Name="HostedOrder" Extends="Order" ObjectType="JDBC"
        Version="6.0">
    <IncludedDataObject Access="RWID" Name="HostedPartner"
        Ordinality="1">
        <Relationship CascadeDelete="y" CascadeErase="n"
            ChangeUpdatesParent="y">
    ```

```
                <JoinKeys>
                    <JoinKey DstJoinField="OrderKey"
                        SrcJoinField="OrderKey"/>
                </JoinKeys>
            </Relationship>
    </IncludedDataObject>
    </DataObject>
```
There are three basic approaches that can be used:

1. You can use extension to simply add any additional DataFields and override the table name. This allows you to include all of the data in a new table. This approach is most useful when you need the same data, but need a distinct copy of it. (Perhaps you maintain a snapshot of how an Order looked before it was turned into a HostedOrder)

2. You can extend Order to add an IncludedDataObject for HostedOrder, where HostedOrder only defines additional data for storage in another table. This means that changes to the original Order DataFields will still be persisted to the Order table, but the additional data for HostedOrder will be persisted to a different table. This is the recommended approach described above.

3. You can define HostedOrder specifying that Order is a IncludedDataObject. This accomplishes the same thing as the second alternative. The problem with this approach is that a HostedOrder does not extend Order, and can no longer be treated as an Order by application code.

Note: Using two tables has a slight disadvantage in performance, but query execution has not been a problem area. Using two tables may reduce data redundancy (depending on your requirements).

If you only occasionally reference the customer extension, then you may want to use a ChildDataObject to take advantage of the lazy link mechanism.

Data Bean Example

This section presents the process of defining and using a data object. Suppose that you want to use a data object to represent a simple enquiry from a customer. This will comprise:

• an email address for the customer

• the date the enquiry was made

• the date a response was returned (optional)

• the content of the enquiry

- the content of the response (optional)
- the product ID of the product about which the enquiry was made (optional)

To Create a Data Object Definition

1. Create the business object element Enquiry and add it to the **DsBusinessObjects.xml** file.

```
<BusinessObject Name="Enquiry" Version="6.0"
    Description="Customer enquiry"/>
```

Use the Version attribute to manage different versions of business objects that may be in use simultaneously. Note that the Version attribute is also used to determine whether access checks are performed automatically (Version 5.0 or higher) or not.

2. Create the recipe for this business object and add it to the **DsRecipes.xml** file.

```
<Recipe Name="Enquiry" Version="6.0" Ordinality="n"
    Description="Customer enquiry">
    <DataObjectList>
        <DataObject Name="Enquiry"
            DataSourceName="ENTERPRISE" />
    </DataObjectList>
</Recipe>
```

The Name attribute of the recipe must match exactly (it is case-sensitive) to the Name of the business object. In Release 7.0.2, each recipe may have more than one data object defined in the data object list, but only one may be a *writable* data object. The data objects define the data source names as an attribute of each data object element. It is these entries that determine the sources from which the business object retrieves its data and the source to which the business object may be persisted.

3. Create a file called **Enquiry.xml** to define the data object. The Name of the data object element must match exactly (it is case-sensitive) the Name attribute defined in the DataObject entry of the recipe element.

In this example, the data for these data objects is held in a database table called CMGT_ENQUIRY, and the ExternalFieldName attribute of each DataField element specifies which column is to be used to retrieve the DataField value. For example, the EMAIL_ADDRESS column of the CMGT_ENQUIRY table holds the email address value associated with an enquiry.

```
<?xml version="1.0"?>
<DataObject Name="Enquiry" Extends="C3PrimaryRW" Version="6.0"
```

```
                    ExternalName="CMGT_ENQUIRY"
                    Access="R" ObjectType="JDBC">
                    <KeyFields>
                        <KeyField Name="Key" ExternalName="ENQUIRY_KEY"/>
                    </KeyFields>
                    <DataFieldList>
                        <DataField Name="EnquiryKey"
                            Writable="n" Mandatory="y"
                            ExternalFieldName="ENQUIRY_KEY"/>
                        <DataField Name="EmailAddress"
                            Writable="n" Mandatory="y"
                            ExternalFieldName="EMAIL_ADDRESS"/>
                        <DataField Name="EnquiryDate"
                            Writable="n" Mandatory="y"
                            ExternalFieldName="ENQUIRY_DATE"/>
                        <DataField Name="ResponseDate"
                            Writable="n" Mandatory="n"
                             ExternalFieldName="RESPONSE_DATE"/>
                        <DataField Name="TimeToRespond"
                            Writable="n" Mandatory="n"/>
                        <DataField Name="EnquiryContent"
                            Writable="n" Mandatory="y"
                            ExternalFieldName="ENQUIRY_CONTENT"/>
                        <DataField Name="ResponseContent"
                            Writable="y" Mandatory="n"
                            ExternalFieldName="RESPONSE_CONTENT"/>
                        <DataField Name="SKU"
                            Writable="n" Mandatory="n"
                            ExternalFieldName="SKU"/>
                    </DataFieldList>
                </DataObject>
```

Note the definition of the TimeToRespond data field: it has no
ExternalFieldName attribute because it does not correspond to a database
column. Values for this field are calculated at runtime and are set in the
EnquiryBean so that its value can be displayed.

4. Define Enquiry and EnquiryList DataElements in **DsDataElements.xml**:

```
<DataElement Name="Enquiry" Description="Enquiry"
    DataType="HEADER"/>
<DataElement Name="EnquiryList" Description="Enquiry list"
    DataType="LIST"/>
```

5. Define a DataElement for each DataField in **DsDataElements.xml**.
   DataElements provide data type information used by the DataManager when it
   is retrieving or saving data for this business object type. For example:

```
<DataElement Name="EnquiryKey" LongName="Enquiry Key"
    DataType="LONG"MaxLength="20" />
```

```
<DataElement Name="EnquiryDate" LongName="Enquiry Date"
    DataType="DATE" />
<DataElement Name="ResponseDate" LongName="Response Date"
    DataType="DATE" />
<DataElement Name="EnquiryContent" LongName="Enquiry content"
    DataType="STRING" MaxLength="256" />
<DataElement Name="ResponseContent" LongName="Response content"
    DataType="STRING" MaxLength="256" />
```

Note that we have not included a DataElement for EmailAddress and SKU. The DataElements for these DataFields are already defined and you can re-use DataElements any number of times (as long as the data type is the same in each occurrence).

6. Create entries in the **ObjectMap.xml** file for this data bean. For example:

```
<Object ID="com.comergent.bean.simple.EnquiryBean">
    <ClassName>com.comergent.bean.simple.EnquiryBean</ClassName>
</Object>
<Object ID="com.comergent.bean.simple.IRdEnquiry">
    <ClassName>com.comergent.bean.simple.EnquiryBean</ClassName>
</Object>
<Object ID="com.comergent.bean.simple.IAccEnquiry">
    <ClassName>com.comergent.bean.simple.EnquiryBean</ClassName>
</Object>
<Object ID="com.comergent.bean.simple.IDataEnquiry">
    <ClassName>com.comergent.bean.simple.EnquiryBean</ClassName>
</Object>
```

7. Finally, define a data source element to correspond to the DataSourceName attribute defined in the DataObject element. This data source is defined in the **DsDataSources.xml** file as part of the schema. In most cases, this data source will already be defined: You only need define a new one if you are using a different database or other data source than the rest of the Knowledgebase. For example:

```
<DataSource Name="ENTERPRISE" Version="2.0">
    <Primary Type="SQL" DataService="JdbcService"
        SubType="ORACLE"
        ConnectionString="jdbc:<driver>:<server>:<port>:<sid>"
        UserId="userid" Password="password" />
    <Alternate Type="SQL" DataService="JdbcService"
        SubType="MSSQL"
        ConnectionString="jdbc:<driver>:<server>:<port>:<sid>"
        UserId="userid" Password="password" />
</DataSource>
```

The DataService attribute of the Primary and Alternate elements determine which class is used to process the EnquiryBean *restore()* and *persist()* methods. The remaining attributes determine exactly how the external source is accessed.

8. Run the generateBean SDK target to generate the source code for the new EnquiryBean and EnquiryListBean data beans and the corresponding interfaces. See "Generated Interfaces" on page 129 for more information on these interfaces.

You can now use Enquiry data beans and its interfaces in business logic classes. To create an instance of an Enquiry data bean, you invoke a method of the form:

```
OMWrapper().getObject("com.comergent.bean.simple.EnquiryBean")
```

This returns an EnquiryBean data bean and its structure is as specified in the Enquiry DataObject. Once you have an instance of the QueryBean class, then populate its key fields and restore the bean to retrieve its data:

```
int queryIndex = 0;
try
{
    String queryKey = request.getParameter("querykey");
    queryIndex = Integer.parseInt(queryKey);
}
catch (Exception e)
{
    //Throw exception if parameter not valid
}
QueryBean queryBean = (QueryBean)
    OMWrapper().getObject("com.comergent.bean.simple.EnquiryBean");
queryBean.setKey(queryIndex);
queryBean.restore();
```

To retrieve a list of enquiries:

```
// Use default settings for DataContext parameters
DataContext context = new DataContext();
// Retrieve enquiries relating to a particular product ID, MXWS-7000
DsQuery query =
    QueryHelper.newWhereClause("SKU", DsQueryOperators.EQUALS,
        "MXWS-7000");
EnquiryListBean enquiryList = (EnquiryListBean)
    OMWrapper.getObject("com.comergent.bean.simple.EnquiryListBean");
// Restore the list.
enquiryList.restore(context, query);
// Walk through the list...
ListIterator enquiryIterator = enquiryList.getEnquiryIterator();
while (enquiryIterator.hasNext())
{
```

```
    boolean isModified = false;
    EnquiryBean enquiry = (EnquiryBean) enquiryIterator.next();
    // Process each enquiry
}
```

In general, you should try to ensure that applications that use the EnquiryBean use one of the generated interfaces rather than the data bean itself. This will enable the application to separate out the implementation of the data object from its interface and let you manage what access the application has to the object's data. To retrieve an instance of a class that implements the IAccEnquiry interface, you can use:

```
IAccEnquiry temp_IAccEnquiry = (IAccEnquiry)
    OMWrapper.getObject("com.comergent.bean.simple.IAccEnquiry");
```

DsElement Tree

This section describes methods to retrieve metadata about databeans. It also describes the DsElement tree used to store data in the data object and business object classes. It is covered here only to support legacy applications: all new applications that use the data bean classes should not need to be concerned with it.

Data objects are created as objects of data bean classes. Each data object holds its content as a tree of components called DsElements (see "DsElements" on page 83).

Their content is retrieved from external systems using the XML schema, and the recipes and data sources defined in the XML schema.
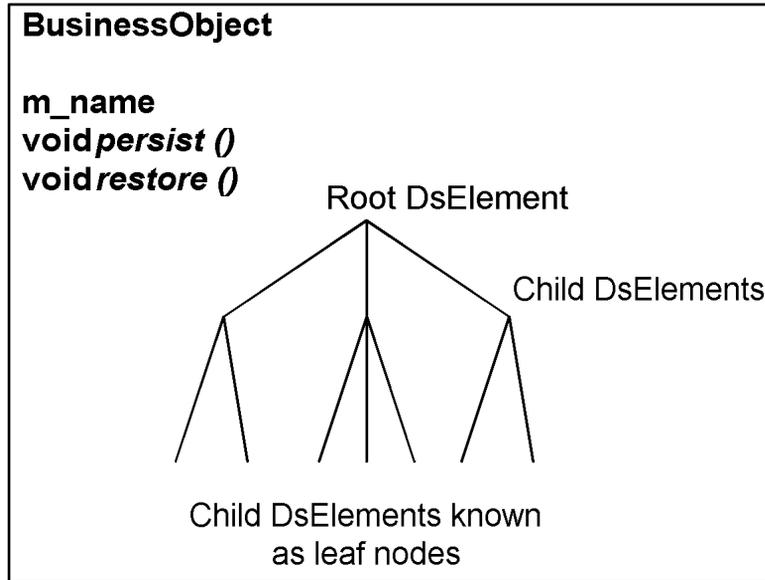
FIGURE 3. Business Object

When the DataManager creates a data bean or business object, it uses the XML schema to determine the structure of its DsElement tree. The DsElement tree is the Java representation of the structure of the business object. The schema also determines the data types that may be inserted at leaf nodes and whether constraints are placed on the values of the node. You access the DsElement tree by invoking the business object method *getRootElement()*.

DsElements

Each DsElement contains data and a DataMap that defines how its data corresponds to its data source. A DsElement may be the child of another DsElement (its *parent*). A DsElement tree is a collection of DsElements, all but one of which have another element in the tree as its parent. By definition, the DsElement with a null parent is the *root* DsElement.

```
DsElement

m_children
m_parent
m_dataMap
m_value

DsElement cloneDsElement ()
DsElement addChild (DataMap dataMap)
void delete ()
String getName ()
int getType ()
DsElement getParent ()
DsElement getByName (String s)
void deleteChild (DsElement child)
```

FIGURE 4. DsElement Methods

The DsElement class provides various additional methods to support navigating through a DsElement tree, notably *children()* that returns an Iterator of the child DsElements of a given DsElement. As well as *getRootElement()*, the business object class also provides the *getElementByName()* method to access directly a named DsElement in its tree.

All DsElements that have the same name, for example child_name, and which are children of a DsElement must have a parent whose name is <child_name>List. The XML schema identifies such elements by defining their ordinality to be "n" as opposed to "1". A DsElement maintains its children in a Vector called m_children.

The DsElement has these important methods:

- *addChild()*: adds a new DsElement defined by the DataMap of this DsElement.

- *cloneDsElement()*: returns a copy of this DsElement.

- *delete()*: sets the DsElemState to DsElemState.DELETED.

- *deleteChild()*: removes a child from the vector m_children by specifying it as a DsElement.

- *getName()*: returns the name of the element as defined by its MetaData.

- *getParent()*: returns the parent of this DsElement.

- *getType()*: returns the type of the element as defined by its DataMap.

DsElement MetaData

It is sometimes useful to retrieve information about a data field and its underlying DsElement. You can use the IData interface method *getMetaData(String elementName)* to this. It returns an object that implements the IMetaData interface. This interace supports the following methods:

- public int getDataType(): returns values as defined in DsDataTypes

- public long getMaxLength(): returns maximum length in bytes

- public long getMaxCharLength(Locale locale): returns maximum length in characters

- public Object getMinValue(): returns the minimum allowed value (or null if there is no minimum)

- public Object getMaxValue(): returns the maximum allowed value (or null if there is no maximum)

- public int getCountAllowedValues()

- public ListIterator getAllowedValueIterator()

- public Object getDefaultValue()

Note that each generated DataBean class implements the IData interface, and so these methods are available to all the generated data beans.

BusinessObject Methods

Use of business objects is deprecated. This section provides information about some business object methods for reference only.

restore() Method

This section provides description of the main forms of the BusinessObject *restore()* method.

```
public void restore(BusinessObject queryObj, int maxResults,
    boolean accessCheck)
```

The principal form of the *restore()* method. Use the queryObj parameter to specify query to be executed by the restore operation. The maxResults parameter determines the maximum number of objects returned. Use the accessCheck parameter to specify whether to check that the current user has the correct entitlements to perform this operation. Once the access check has been performed, then the *restore(BusinessObject queryObj, int maxResults)* is called.

```
public void restore(BusinessObject queryObj, int maxResults)
```

This method calls the *restore()* method *restore(this, queryObj, maxResults, false)*of the underlying data object.

```
public void restore(BusinessObject queryObj)
```

This is equivalent to calling *restore(queryObj, 0)*.

```
public void restore()
```

This form of the method calls the *restore(null, 0)* method.

persist() Method

This section provides descriptions of the main forms of the BusinessObject *persist()* method.

```
public void persist(boolean synch, boolean commit,
    boolean accessCheck)
```

The boolean parameters determine respectively whether the persist operation is synchronized, should be committed to the underlying data source, and whether an access check should be performed prior to persisting.

```
public void persist(boolean synch, boolean commit)
```

This form of the method is equivalent to *persist(synch, commit, false)* for business objects whose Version attribute is 4.0 or less. It is equivalent to *persist(synch, commit, true)* for business objects whose Version attribute is 5.0 or more.

```
public void persist()
```

This form of the method calls *persist(false, true)*.

The BusinessObject class also has these methods:

- *delete()*: empties the business object by deleting its DsElement tree.

- *getRootElement()*: returns the root DsElement of the DsElement tree.

- *getType()*: returns the name of the root element of the DsElement tree. This is the type of the business object.

- *setRootElement()*: sets the root element of this business object.

CHAPTER 7   *Using the Security Mechanisms*

The Comergent eBusiness System offers developers several mechanisms to manage security in their applications. This chapter describes how you can use the entitlements, access control lists, and access policies to manage what users can do: what functions they may perform and what access they to have data objects. It covers:

- "Managing Message Types" on page 90

- "Managing Access to Data Objects Using Access Policies" on page 93

- "Managing Access to Data Objects Using ACLs" on page 100

- "Creating an ACL" on page 102

- "Troubleshooting ACL Issues" on page 103

- "Passing Login Data Through a URL" on page 104

Please consult the *Comergent eBusiness System Reference Guide* for a description of the underlying security architecture.

Managing Message Types

As you customize the Comergent eBusiness System, you must take into account which types of users can execute which message types and which Web pages should be accessible to which users.

Each message type corresponds to a request that the user's browser makes to the server. Message types are organized into message groups. A role is defined as a collection of message groups that are either granted or denied to the role.

```
<RoleDefinition Name="Partner.SalesRep">
    <Description>
        This is the role associated with the Lead Users.
        Lead Users can work leads that are assigned to them.
    </Description>
    <Grant>LeadMgmtDetailGroup</Grant>
    <Grant>ProposalGroup</Grant>
</RoleDefinition>
```

In Release 7.0.2, roles are aggregated into functions: a function is intended to be the collection of roles that correspond to a business function such as finance or sales. Users are assigned functions, and the set of functions available to be assigned to a user depends on their user type. A function is declared in the **Entitlements.xml** configuration file using an element of this form:

```
<UserFunctionMapping Name="IndirectSalesExecutive">
    <Description>Sales</Description>
    <Role>Partner.IndirectBuyer</Role>
    <Role>Partner.SalesRep</Role>
    <Role>Partner.SalesManager</Role>
    <Role>Partner.CustomerServiceRepresentative</Role>
</UserFunctionMapping>
```

The same role can be included in more than one function. Consequently, you can define functions that overlap in some roles, or define a function that is only a subset of another function.

Checking for Entitlement

The system will test whether a user can execute a message type when a request is received. However, to prevent users from seeing error pages, in general, you should perform an entitlement check for each link on a JSP page to test that the user can execute the message type associated to the link.

You can use the *canRequest(String messageType)* method of the User class. You can retrieve the current User object from the session as follows:

```
User sessionUser = comergentSession.getUser();
```

For example, the following lines in a JSP page are used to determine whether the current user can access a promotion detail page.

```
User sessionUser = comergentSession.getUser();
...
<% if (sessionUser.canRequest("PromotionDetailDisplay")
   {
%>
      <A HREF="<%= link("partnerMkt", "PromotionDetailDisplay",
          "PromotionKey=<%= promotion.getKey() %>")%>) %>">
      <%= ph(promoName)%></A>
<%
   } else {
%>
      <%= ph(promoName)%>
<%
   }
%>
```

Managing User Types

There may be situations in which you need to modify an existing user type or you may need to create a new user type.

Adding a Role to a User Type

The definitions of user types are declared in the UserTypeDefinition elements in **Entitlements.xml**. For example, in Release 7.0.2, this is the definition of the IndirectUser user type:

```
<UserTypeDefinition Name="IndirectUser">
    <Description>
        Employees or agents of an indirect commerce partner.
    </Description>
    <Label>User</Label>
    <MandatoryRoleSet>
        <Role>Partner.User</Role>
        <Role>StorefrontCustomer*.TransferUser</Role>
    </MandatoryRoleSet>
    <UserFunctionMapping Name="IndirectCommerce">
        <Description>Commerce</Description>
        <Role>Partner.IndirectBuyer</Role>
        <Role>Partner.CustomerServiceRepresentative</Role>
    </UserFunctionMapping>
    <UserFunctionMapping Name="IndirectSales">
        <Description>Sales</Description>
```

```
            <Role>Partner.IndirectBuyer</Role>
            <Role>Partner.SalesRep</Role>
            <Role>Partner.CustomerServiceRepresentative</Role>
    </UserFunctionMapping>
    <UserFunctionMapping Name="IndirectSalesExecutive">
            <Description>Sales</Description>
            <Role>Partner.IndirectBuyer</Role>
            <Role>Partner.SalesRep</Role>
            <Role>Partner.SalesManager</Role>
            <Role>Partner.CustomerServiceRepresentative</Role>
    </UserFunctionMapping>
    <UserFunctionMapping Name="PartnerProgramManagement">
            <Description>Program Management</Description>
            <Role>Partner.ProductManager</Role>
            <Role>Partner.StorefrontAdministrator</Role>
    </UserFunctionMapping>
    <UserFunctionMapping Name="PartnerBasicAdministration">
            <Description>Basic Profile Maintenance</Description>
            <Role>Partner.BasicAdministrator</Role>
    </UserFunctionMapping>
    <UserFunctionMapping Name="PartnerAdministration">
            <Description>Profile Administration</Description>
            <Role>Partner.BasicAdministrator</Role>
            <Role>Partner.Administrator</Role>
    </UserFunctionMapping>
</UserTypeDefinition>
```

You can add a function or role to a user type simply by editing the
Entitlements.xml file accordingly and by granting appropriate message groups to
the new role. Note that you must restart the Comergent eBusiness System for the
new function to be available for assignment.

The MandatoryRoleSet element specifies the set of roles that cannot be removed
from a user's entitlements. All users of this user type have these roles.

Creating a User Type

You can create the definition of a new user type simply by adding it to the
Entitlements.xml file. Each user type is associated with partner types. The
PartnerTypeDefinition elements of the **Entitlements.xml** configuration file
determine which user types are available to which partners, so that only users of
those types can be created for each partner. For example, consider the following
PartnerTypeDefinition element:

```
<PartnerTypeDefinition Name="IndirectPartner">
    <Description>
        An IndirectCommercePartnerType partner has a
        relationship with the enterprise for
```

```
        the purpose of indirect commerce.
    </Description>
    <UserType>IndirectUser</UserType>
</PartnerTypeDefinition>
```

This says that when a user is created for a partner whose type is IndirectPartner, then only the IndirectUser user type may be selected.

Managing Access to Data Objects Using Access Policies

Release 6.4 of the Comergent eBusiness System introduced a new mechanism to manage access to data object: access policies. Access policies are designed to adhere more closely to the Java Authentication and Authorization Services (JAAS) model. You can chose to use ACLs or access policies to control access to data objects, but for each data object type you must choose one method or the other.

Access policies are particularly important for data objects that can be modified using the DsUpdate functionality introduced in Release 6.4. If your implementation of the Comergent eBusiness System uses DsUpdate, then you must use access policies to manage the data objects updated using DsUpdate.

Access policies are applied to a data object by use of a ResourceClass element. For example:

```
<ResourceClass>com.comergent.bean.simple.PartnerBean</ResourceClass>
```

This element is declared within an AccessPolicy element. You can apply the same access policy to several different data objects by listing each of them as a ResourceClass element. Access policies are inherited by data objects that extend other data objects. For example, if an access policy declares the ShoppingCart as a ResourceClass, then the same access policy is also applied to the ChannelShoppingCart data object because it extends the ShoppingCart data object.

If an access policy is defined for a data object in the **AccessPolicy.xml** configuration file, then it determines access to the data object. If no access policy is defined, then the legacy ACL mechanism is used. See "Managing Access to Data Objects Using ACLs" on page 100 for details.

Overview

An access policy controls access to a data object by specifying the conditions under which a user can perform an action on the data object, referred to as the *resource*. The following actions can be performed on data objects:

- Create

- Delete

- Restore

- Update

The conditions are specified as evaluating principals and expressions, and comparing them to the permitted values of the access policy. In general, principals are attributes of the user attempting an action on the data object, but they may be defined more generally. Expressions may be likened to SQL queries: they act as filters on the lists of data object being tested for access.

For example, suppose that you want to use an access policy to specify that only users that belong to a partner can update their partner profile. In this case, the action is Update and the condition that you want to define is that if you evaluate the partner key of the user object, then it must equal the partner key of the partner data object. In this example, the principal being evaluated is the partner key of the user.

Inheritance

When you define an access policy on a data object, it is inherited by all the data objects that extend it. Note that this means that if one data object extends another and you want to define different access policies for each, then you must declare distinct access policies for each of them.

AccessPolicy.xml Configuration File

You define access policies using an **AccessPolicy.xml** configuration file. Each AccessPolicy element declared in this file can be applied to one data object type: the data object is specified as the DataObject attribute of the AccessPolicy element.

Principal Qualifiers

Principal qualifiers are defined using the PrincipalQualifierDefinition element of the **AccessPolicy.xml** configuration file. Principal qualifiers are essentially Java classes that implement the PrincipalQualifier interface.

```
<PrincipalQualifierDefinition PrincipalType="UserType"
    Class="com.comergent.dcm.entitlement.UserTypeQualifier"/>
```

Access Policies

Each AccessPolicy element specifies which PrincipalQualifier is to be used to evaluate the principal conditions by specifying the name of the PrincipalQualifier as the PrincipalQualifier attribute of the AccessPolicy element.

```
<AccessPolicy Name="UserPolicy" DataObject="UserContact"
```

```
PrincipalQualifier="UserType">
```

Access Checkers

AccessChecker elements are used to define the individual checks that can be made
to determine whether a user can access a data object. Each AccessPolicy element
declares one or more AccessChecker elements. Each AccessChecker element
specifies the permitted values of the principal to be compared, the action type to be
checked, and any expressions to be evaluated to filter the data objects that can be
acted on.

```
<AccessChecker>
    <Principal Select="Partner.DirectCommerceUser"/>
    <Principal Select="Partner.User"/>
    <ActionType Type="Restore"/>
    <BooleanExpression>
        <ComparativeExpression Operator="Equals">
            <Term>user.PartnerKey</Term>
            <Term>resource.PartnerKey</Term>
        </ComparativeExpression>
    </BooleanExpression>
</AccessChecker>
```

In this example of an AccessChecker element, the action type being checked for is
"Restore". The access policy is checked by comparing the user role of the user to
see if one matches either "Partner.DirectCommerceUser" or "Partner.User". The
Expression element is evaluated to see if the PartnerKey field of the data object is
equal to the partner key of the user, and this filter is applied to the data objects in
question.

If there is more than one BooleanExpression element in an AccessChecker element,
then use the Operator attribute to specify whether the boolean expressions should
be combined using AND or OR. If no Operator attribute is specified, then OR is
used by default.

Access Services

You can make use of access services to help retrieve information used to check
access policies. Each AccessServiceDefinition element provides a name and a
class. For example:

```
<AccessServiceDefinition Name="ownersPartnerKey" Type="resource" >
    com.comergent.reference.dcm.entitlement.OwnersPartnerKeyService
    <Description>
        Returns the partner key as a Long value for the owner of the
        resource if the resource extends C3PrimaryRWBean. Otherwise
        returns null.
    </Description>
```

```
</AccessServiceDefinition>
```

This access service retrieves the owner key of the resource on which access is being checked.

Boolean Expressions

BooleanExpression elements are used to express the exact conditions under which access is granted to objects. They may be nested and they take an Operator attribute to specify how child elements should be combined.

As well as child BooleanExpression elements, you can also use ComparativeExpression elements, SetExpression elements, and Not elements to build up complex conditions:

- ComparativeExpression: use this element to compare the values of two fields.

- SetExpression: use this element to test membership of lists.

- Not: use this to wrap another expression so that the opposite boolean value is used.

Example

This fragment of the **AccessPolicy.xml** configuration file provides an example of how access policies are used. It determines access to order inquiry lists as described below.

```
<AccessPolicy Name="OrderInquiryListPolicy"
    PrincipalQualifier="UserRole">
    <Description>
        Controls access to Order Inquiry Lists.
    </Description>
    <ResourceClass>
        com.comergent.bean.simple.OrderInquiryListBean
    </ResourceClass>
    <ResourceClass>
        com.comergent.bean.simple.LightWeightOILBean
    </ResourceClass>
    <AccessChecker>
        <Description>
            Direct partner users with the listed roles can read an
            inquiry list if they own it or routed it to another user.
        </Description>
        <Principal>Anonymous.User</Principal>
        <Principal>Registered.User</Principal>
        <Principal>Partner.DirectBuyer</Principal>
        <Principal>Partner.ProcurementUser</Principal>
```

```
                        <Principal>StorefrontCustomer*.TransferUser</Principal>
                        <Principal>StorefrontCustomer*.User</Principal>
                        <Principal>StorefrontCustomer*.AnonymousUser</Principal>
                        <Principal>StorefrontCustomer*.RegisteredUser</Principal>
                        <ActionType>Restore</ActionType>
                        <BooleanExpression Operator="Or" >
                            <ComparativeExpression Operator="Equals">
                                <Term>user.UserKey</Term>
                                <Term>resource.OwnedBy</Term>
                            </ComparativeExpression>
                            <ComparativeExpression Operator="Equals">
                                <Term>user.UserKey</Term>
                                <Term>resource.RouteFromUserKey</Term>
                            </ComparativeExpression>
                            <BooleanExpression Operator="And">
                                <SetExpression Operator="Intersection" >
                                    <Set>
                                    <Term>"Partner.BasicAdministrator"</Term>
                                    </Set>
                                    <Set>user.roleNameSet</Set>
                                </SetExpression>
                                <ComparativeExpression Operator="Equals">
                                    <Term>user.PartnerKey</Term>
                                    <Term>service.ownersPartnerKey</Term>
                                </ComparativeExpression>
                            </BooleanExpression>
                        </BooleanExpression>
                </AccessChecker>
                <AccessChecker>
                    <Principal>Anonymous.User</Principal>
                    <Principal>Registered.User</Principal>
                    <Principal>Partner.DirectBuyer</Principal>
                    <Principal>Partner.ProcurementUser</Principal>
                    <Principal>StorefrontCustomer*.TransferUser</Principal>
                    <Principal>StorefrontCustomer*.User</Principal>
                    <Principal>StorefrontCustomer*.AnonymousUser</Principal>
                    <Principal>StorefrontCustomer*.RegisteredUser</Principal>
                    <ActionType>Update</ActionType>
                    <ActionType>Create</ActionType>
                    <ActionType>Delete</ActionType>
                    <BooleanExpression Operator="Or">
                        <ComparativeExpression Operator="Equals">
                            <Term>user.UserKey</Term>
                            <Term>resource.OwnedBy</Term>
                        </ComparativeExpression>
                        <BooleanExpression Operator="And">
                            <SetExpression Operator="Intersection" >
                                <Set>
                                <Term>"Partner.BasicAdministrator"</Term>
```

```
                        </Set>
                        <Set>user.roleNameSet</Set>
                    </SetExpression>
                    <ComparativeExpression Operator="Equals">
                        <Term>user.PartnerKey</Term>
                        <Term>service.ownersPartnerKey</Term>
                    </ComparativeExpression>
                </BooleanExpression>
            </BooleanExpression>
        </AccessChecker>
        <AccessChecker>
            <Description>CustomerServiceRepresentatives can create,
            modify or delete any enterprise cart, but not storefront
            carts.
            </Description>
            <Principal>
                Enterprise.CustomerServiceRepresentative
            </Principal>
            <ActionType>Restore</ActionType>
            <ActionType>Update</ActionType>
            <ActionType>Create</ActionType>
            <ActionType>Delete</ActionType>
            <BooleanExpression Operator="And">
                <Not>
                    <SetExpression Operator="Intersection">
                        <Set><Term>service.ownersUserType</Term></Set>
                        <Set>
                        <Term>"StorefrontCustomerUser"</Term>
                        <Term>"StorefrontCustomerAnonymousUser"</Term>
                        <Term>"StorefrontCustomerRegisteredUser"</Term>
                        </Set>
                    </SetExpression>
                </Not>
                <SetExpression Operator="Intersection">
                    <Set>service.csrAssignedPartners</Set>
                    <Set><Term>service.ownersRootPartnerKey</Term></Set>
                </SetExpression>
            </BooleanExpression>
        </AccessChecker>
        <AccessChecker>
            <Principal>*</Principal>
            <ActionType>Restore</ActionType>
            <ActionType>Create</ActionType>
            <ActionType>Update</ActionType>
            <ActionType>Delete</ActionType>
            <BooleanExpression>
                <Never/>
            </BooleanExpression>
        </AccessChecker>
```

```
</AccessPolicy>
```

The way to read this fragment is as follows:

- This access policy determines access to OrderInquiryListBeans and LightWeightOILBeans

- Users who have one of the listed roles (Anonymous.User, Registered.User, and so on) may have Restore access (that is, have read access to the resource) if they satisfy one of the declared BooleanExpressions:

    - Either:

        - The user's key is equal to the owner key of the resource.

    - Or:

        - The user's key is equal to the routed from key of the resource.

    - Or:

        - The user has the Partner.BasicAdministrator role;

        And

        - The user's partner key is the same as the partner key of the owner of the resource.

- Users who have one of the listed roles (Anonymous.User, Registered.User, and so on) may have Update, Create, Delete access (that is, have write, create, and delete access to the resource) if they satisfy one of the declared BooleanExpressions:

    - Either:

        - The user's key is equal to the owner key of the resource.

    - Or:

        - The user has the Partner.BasicAdministrator role;

        And

        - The user's partner key is the same as the partner key of the owner of the resource.

- Users who have the Enterprise.CustomerServiceRepresentative role may have Restore, Update, Create, Delete access (that is, have read, write, create, and delete access to the resource) if they satisfy all of the declared BooleanExpressions:

- • First:
  - • The resource's owner is not a storefront user;
- • And:
  - • The resource's owner root partner key is one of the keys of partners assigned to the user.

If you wanted to change this access policy so that all the users of a partner had read access to an inquiry list owned by one partner user, then you could remove the section of the Restore AccessChecker element that refers to the Partner.BasicAdministrator role.

Managing Access to Data Objects Using ACLs

The following access privileges are defined for each data object that extends the C3PrimaryRW data object:

- • delete
- • insert
- • read
- • write

If you want to control access to a data object using ACLs, then an access control list (ACL) must be attached to it when it is created. When an application retrieves a data object, you can query the data object to determine what access the current user has to it.

Data Bean Methods

- • *boolean isReadable()*: returns true if the specified user has read access to the specified business object; otherwise, returns false. This method is declared in the IRdC3PrimaryRW interface.

- • *boolean isWriteable()*: returns true if the specified user has write access to the specified business object; otherwise, returns false. This method is declared in the IAccC3PrimaryRW interface.

- • *boolean isInsertable()*: returns true if the specified user has insert access to the specified business object; otherwise, returns false. This method is declared in the IAccC3PrimaryRW interface.

- • *boolean isDeletable()*: returns true if the specified user has delete access to the specified business object; otherwise, returns false. This method is declared in the IAccC3PrimaryRW interface.

- • *boolean CheckPermission(int i)*: is a general method to check for permissions on data objects that extend the C3PrimaryRW data object. The int parameter is typically one of AccessControl.READ_PERMIT, AccessControl.WRITE_PERMIT, and so on, but it may be any permission bit mask.

Attaching an ACL to a Data Object

In general, you simply need to perform the following steps:

1. Create the data bean. Typically, if you are creating a new data bean, then this step comprises using the ObjectManager to retrieve a new data bean and then setting its fields using the standard accessor methods.

2. Retrieve or create the ACL.

   - • You can retrieve ACLs by name using the AccessControlFactory class *getAccessControlListByName(String s)* method. Note that uniqueness of ACL name is not enforced by the system, so take care to enforce uniqueness as you create ACLs.

   - • You can create an ACL by first calling AccessControlFactory class *getEditableAccessControlList()* method: this returns an instance of the EditableAccessControlList class. You can then use its accessor methods to set the users, groups, fixed, and roles fields as required. Then call its *save()* method to persist the new ACL to the Knowledgebase.

3. Attach the ACL to the data bean. Use the *attachACL()* method of the AccessControlAPI class: it takes as parameters the data bean, the ACL, the user key of the current user, and a boolean to indicate whether a *persist()* call should be made to save the data object. A check is performed to verify that the current user is entitled to attach the ACL to the data object.

Examples

Here is a fragment of code indicating how an existing, named ACL is attached to a data object.

```
Long userKey = CommerceUtils.getCurrentUserKey();
IOrderInquiryList temp_IOrderInquiryList = (IOrderInquiryList)
   OMWrapper.getObject("com.comergent.api.apps.orderMgmt.oil.
      IOrderInquiryList");
...
```

```
various fields are set
...
AccessControlList temp_AccessControlList =
AccessControlFactory.getAccessControlListByName("Inquiry List ACL");
AccessControlAPI.attachACL(temp_AccessControlList,
    temp_IOrderInquiryList, userKey, true);
```

Here is a fragment of code indicating how a newly-created ACL is attached to a data object.

```
Long userKey = CommerceUtils.getCurrentUserKey();
IOrderInquiryList temp_IOrderInquiryList = (IOrderInquiryList)
    OMWrapper.getObject("com.comergent.api.apps.orderMgmt.oil.
        IOrderInquiryList");
...
various fields are set
...
EditableAccessControlList temp_EditableAcessControlList =
AccessControlFactory.getEditableAccessControlList();
temp_EditableAcessControlList.setName("New Inquiry List ACL");
//Prevent owner from deleting object
temp_EditableAcessControlList.addUser(AccessControl.DELET_PERMIT,
    false, userKey);
//Allow users in owner's group or owner's root group to modify object
temp_EditableAcessControlList.addFixed(AccessControl.WRITE_PERMIT,
    true, AccessControl.GROUP);
temp_EditableAcessControlList.addFixed(AccessControl.WRITE_PERMIT,
    true, AccessControl.ROOT);
//Allow world to read object
temp_EditableAcessControlList.addFixed(AccessControl.READ_PERMIT,
    true, AccessControl.WORLD);
AccessControlAPI.attachACL(temp_EditableAcessControlList,
    temp_IOrderInquiryList, userKey, true);
```

Creating an ACL

In the previous section, an example is given of creating an ACL using the available EditableAccessControlList methods. In addition, you can use the ACLBuilder class to create an ACL or to modify an ACL attached to a data object. It provides helper methods designed to make the process of creating a new ACL as simple as possible.

Note that if you do create custom ACLs for individual data objects, then this can be a performance drag on the system. You should consider carefully whether an appropriate general ACL can be attached to all data objects of a particular type.

Use the AccessControlFactory class *getAccessBuilder()* method to create an AccessBuilder. By default, an object of the AccessControlAdapter class is returned.

Its *grantAccess()* and *denyAccess()* methods act directly on the ACL attached to a data object. Consequently, if you use these methods to modify an ACL attached to a data object, then bear in mind that the underlying ACL may also be attached to other data objects and changes will affect changes to these data objects too.

Troubleshooting ACL Issues

This section describes a simple use case to illustrate how to identify and fix problems that you can encounter when modifying or creating ACLs.

In this example, a user tries to log in, but clicking the **Login** button causes an error page to be displayed with the following message: "User akite does not have READ permission on: Partner". This means that when the access check is being performed on the partner object, the ACL attached to the partner does not grant read access to akite.

1. Find additional information about the user "akite" in the CMGT_USER_CONTACTS table. For example:

    ```
    SELECT * FROM CMGT_USER_CONTACTS WHERE USER_NAME = 'akite'
    ```

    Results:

    -   the USER_KEY is 57901

    -   the ROLES are Partner.User, Partner.DirectCommerce

    -   the user's group key is 4

    -   the user's partner key is 4

2. Find information about the user's partner in the CMGT_PARTNERS table. For example:

    ```
    SELECT * FROM CMGT_PARTNERS WHERE PARTNER_KEY = 4
    ```

    Results:

    -   this partner object has ACL key 3 (ACL key is the ACCESS_KEY field in the CMGT_PARTNERS table)

    -   this partner's owner is identified by USER_KEY 0

3. Find the relevant lines in the CMGT_ACCESSLIST table.

    The error message tells us that either akite is not being granted read access by virtue of their user key or group membership or that the roles that are assigned to akite do not have access to the partner object. From Step 2, we

know that the partner object has an ACL key = 3. So, we need to find all rows in the CMGT_ACCESSLIST table with ACL_KEY = 3 and PERMISSION = 0 (read) to investigate if at least one of these rows allows Partner.User or Partner.DirectCommerce read access. Use the SQL query:

```
SELECT USER_KEY, GROUP_KEY, FIXED, ROLES FROM CMGT_ACCESSLIST
WHERE ACL_KEY = 3 and Permission = 0
```

Inspect the USER_KEY, GROUP_KEY, and FIXED columns: typically, you should look to see that akite's user key is listed in one of the USER_KEY columns; or that akite's group is in one of the GROUP_KEY columns; or that the akite is included by virtue of one of the FIXED values. Since the owner of the partner object has user key 0, you can identify the group to which this user belongs as follows:

```
SELECT GROUP_KEY FROM CMGT_GROUP_USERS WHERE USER_KEY = 0
```

If there are roles defined, then make sure that either Partner.User or Partner.DirectCommerce are listed as roles in any row which grant read access to akite.

4. In this particular example, you may find that the fix is to update CMGT_ACCESSLIST to append the Partner.User role to one of the rows whose ACL_KEY = 3.

Passing Login Data Through a URL

Most users of the Comergent eBusiness System enter the system by pointing their browsers to the appropriate login page. However, sometimes, you may want to enable users to access a specific page such as the detail page of an order directly.

You can do this simply by constructing the URL as you would like it to be, for example:

```
http://server/Comergent/en/US/direct/matrix?cmd=OrderDisplay&-
ShoppingCartKey=600501
```

When a user clicks on this link, their request is routed to the appropriate login page. In the login form, the request data from the original URL is automatically encoded as hidden form parameters, for example:

```
<input type="hidden" name="cmd" value="directLogin" >
<input type="hidden" name="validate" value="true" >
<input type="hidden" name="LoginData-messageType"
    value="OrderDisplay"/>
<input type="hidden" name="LoginData-ShoppingCartKey"
    value="600501"/>
<input type="hidden" name="LoginData-entryPoint" value="direct"/>
```

When the login form is submitted, if the login is successful, then the parameters that begin with "LoginData-" are processed by the LoginController and added back to the request object with "LoginData-" removed from the parameter names. When the request is forwarded to the message type specified by the LoginData-messageType parameter, the original parameters are now available to the controller and JSP pages used to process the request.

Note that in general the message type in the original URL will be changed to the fallback redirect message type for the message type or message group to which the message type belongs. Consequently, take care that your intended message type is the fallback redirect message type for its message group.

You specify the fallback redirect mesage type along these lines:

```
<MessageGroup Name="AdvisorGroup">
    <FallbackRedirect>
    <Redirect EntryPoint="partnerMkt">PartnerHomePageDataDisplay"
    </Redirect>
    <Redirect EntryPoint="catalog">MatrixHomePageDisplay"</Redirect>
    <Redirect EntryPoint="advisor">MatrixHomePageDisplay"</Redirect>
    <Redirect EntryPoint="configurator">MatrixHomePageDisplay"
    </Redirect>
    </FallbackRedirect>
    ...
    Message type definitions
    ...
</MessageGroup>
```

The way to read this XML extract is as follows: if an unauthenticated request is tries to execute any message type in the AdvisorGroup of message types, then redirect them to the PartnerHomePageDisplay message type if the entrypoint of the request is "partnerMkt" or redirect them to the MatrixHomePageDisplay message type if the entrypoint is one of the other three declared.

The LoginData-entrypoint is used to specify which entry point is used to access the system. It is retrieved from the original URL to ensure that the user is directed to the correct login page. A message group may have more than one default message type: they are differentiated by their Key attribute that specifies different entry points. For example:

```
<GroupDefault Key="partnerMkt" Value="IndirectWorkspaceDisplay" />
<GroupDefault Key="marketPlace" Value="DirectWorkspaceDisplay" />
```

CHAPTER 8 *Logging*

This chapter describes the logging mechanism provided by the Comergent eBusiness System. It enables application writers to log activity in the Comergent eBusiness System. It uses the log4j API and **log4j.properties** configuration files to configure the logging behavior.

The logging capability also provides support for auditing changes to data objects. See "Auditing Changes to Data Objects" on page 108 for more information.

Overview

The log4j API provides a flexible and extensible logging framework to manage the logging behavior of the Comergent eBusiness System. Its basic configuration and use is described in the *Comergent eBusiness System Implementation Guide*. This section describes the use of the framework as you customize and extend the Comergent eBusiness System.

Note that this framework replaces the previous framework used by the Comergent eBusiness System: this used the Global class and its *log*Level*()* methods. These are now deprecated. See "Global Class" on page 467 for more information on these methods.

To use the log4j API, you should create a Logger class in each class file along these lines:

```
private static final org.apache.log4j.Logger log =
    org.apache.log4j.Logger.getLogger(NameOfClass.class);
```

When you want to make a log entry call:

```
log.info("This is a log entry");
```

The method you call depends on the logging level at which you want to record the message. You can use the following methods:

- *debug()*
- *error()*
- *fatal()*
- *info()*
- *warning()*

You can also use *log(priority, message)*, but in general the listed methods should be sufficient.

Auditing Changes to Data Objects

In many implementations, you may want to provide an audit trail that tracks changes made to data in the Comergent eBusiness System. In Release 7.0.1 and higher, you can do this by logging any changes made to data objects. If you set the logging level to INFO or higher in any DataBean class, then whenever *persist()* is invoked on an instance of this class, then a log message is written out to the Logger for the class. For example: the following is a sample line that is written out when a change is made to a partner:

```
2006.01.18 13:41:05:546 Env/http-8080-Processor23:INFO:PartnerBean
Updating: com.comergent.bean.simple.PartnerBean KeyFields - Partner-
Key: 301 Changes -PartnerKey -> old: 301 new: 301PartnerName -> old:
Scalar2 new: Scalar2 LegalName -> old: null new: null ParentCompany -
> old: null new: nullStatus -> old: A new: A DunBradID -> old: null
new: nullBusinessID -> old: Scalar2-001 new: Scalar2-
001PartnerTypeCode -> old: 10 new: 10PartnerLevelCode -> old: 20 new:
20XMLMessageVersion -> old: dXML 4.0 new: dXML 4.0BusinessTransaction
-> old: SELL new: SELL NetWorth -> old: null new: null NumEmployees -
> old: null new: null PotRevCurrFy -> old: null new: null PotRevNextFy
-> old: null new: null ReferenceUseFlag -> old: null new: null Coterm-
DayMonth -> old: null new: nullURL -> old: http:///www.scalar.com new:
http:///www.scalar2.com LogoURL -> old: null new: null DistiAccess ->
old: null new: null YearEstd -> old: null new: null AnalysisFy -> old:
null new: null FyEndMonthCode -> old: null new: null AccountManagerKey
-> old: null new: null MessageURL -> old: null new: null EmailAddress
```

```
-> old: null new: nullCommerceCategory -> old: 2 new: 2 PartnerRefNum
-> old: null new: null ParentKey -> old: null new: null RootPartnerKey
-> old: null new: null ParentCode -> old: null new: null CustomField1
-> old: null new: null CustomField2 -> old: null new: null
CustomField3 -> old: null new: null CustomField4 -> old: null new:
null CustomField5 -> old: null new: null PartnerCom -> old: null new:
null Storefront -> old: null new: null URLName -> old: null new: null
ContentType -> old: null new: nullPartnerStatusCode -> old: 10 new:
10OrganizationType -> old: DirectPartner new: DirectPartner Inherited-
PartnerStatusCode -> old: null new: nullCreditLimit -> old: 0.0000
new: 0.00AvailableCredit -> old: 0.0000 new: 0.0000CreditCurrencyCode
-> old: 23 new: 23 MaxAssignableReps -> old: null new: null Remote-
Prices -> old: null new: null RemotePriceExpiryInterval -> old: null
new: nullCoopPercentage -> old: 0.000000 new: 0.000CoopAccountMax ->
old: 0.000000 new: 0.00 PartnerID -> old: null new: nullOwnedBy ->
old: 0 new: 0AccessKey -> old: 5601 new: 5601UpdateDate -> old: 2006-
01-18 13:39:33.0 new: 2006-01-18 13:41:05.484UpdatedBy -> old: 0 new:
0CreateDate -> old: 2006-01-04 13:19:38.0 new: 2006-01-04
13:19:38.0CreatedBy -> old: 0 new: 0
```

You can dynamically change the logging level for any class in the Comergent eBusiness System through the administration UI: see the *Comergent eBusiness System Administration Guide* for details. However, if you do this, then the change to the logging level is not persistent, and will be lost if the servlet container is restarted. In addition, the logging is written out to the standard Appender which may not be secure.

You should specify any audit logging by customizing the **log4j.properties** configuration file: this ensures that the auditing will continue to be done even if the servlet container is restarted, and you can specify a custom Appender to process the audit information. For example, you can specify that the Appender posts the logging message to a remote Web server which can be secured independently of the Comergent eBusiness System.

As an example, the following entries in the **log4j.properties** configuration file ensure that all changes to the UserContact data object are audited:

```
log4j.logger.com.comergent.bean.simple.UserContactBean=info
log4j.appender.com.comergent.bean.simple.UserContactBean=com.comer-
gent.logging.ComergentRollingFileAppender
log4j.appender.com.comergent.bean.simple.UserContactBean.layout =
org.apache.log4j.PatternLayout
```

If you want to specify that a remote log server can connect asa client in order to save audit information from the Comergent eBusiness System, then you could specify:

```
log4j.appender.com.comergent.bean.simple.UserContact-
```

```
Bean=org.apache.log4j.net.SocketHubAppender
log4j.appender.com.comergent.bean.simple.UserContactBean.port=4321
```

Events

This chapter describes the use of the event mechanism provided by the Comergent eBusiness System. It enables application writers to fire events from an application and to have one or more applications respond to the event.

Overview

Events are a means for applications to signal to other applications when a notable incident has occurred in the Comergent eBusiness System: examples of events include users logging in, orders being placed, price lists being assigned to partners, leads being assigned to partners, and so on.

Applications fire events by instantiating an *event producer*, using it to create an event, and then invoking the EventBus to fire the event. Applications use *event consumers* to respond to events: the EventBus instantiates event consumers for each type of event. One event can be processed by one or more consumers.
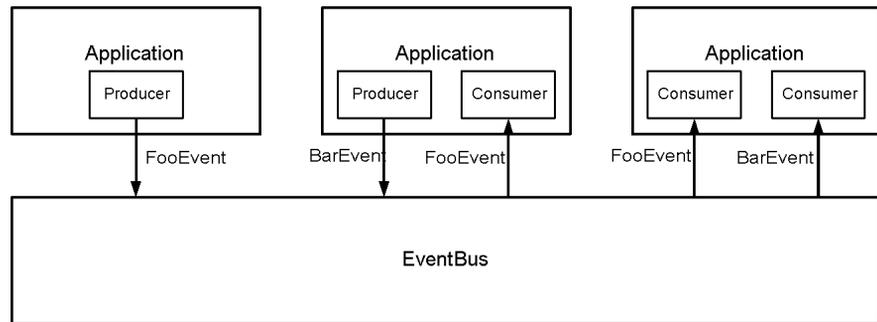
FIGURE 5. Applications and Events

Events are identified by their class: all event classes must implement the ComergentEvent interface, but can otherwise be created to suit the needs of the communication between its event producer and its event consumers.

Firing an Event

Application writers can fire events from their application by using the EventBus class as follows:

```
ApplicationEventProducer applicationEventProducer =
    new ApplicationEventProducer();
ApplicationEvent applicationEvent =
    applicationEventProducer.createEvent();
EventBus.getInstance().fireEvent(applicationEvent);
```

The event producer class (in this case, ApplicationEvent Producer) is created by the application developer: it must implement the EventProducer interface, though currently this interface defines no methods. It must have one or more methods to create events: by convention these methods are called *createEvent()*, and they may take zero or more arguments: typically, you pass into the event the information that the event consumers will need to process the event. The application event must implement the ComergentEvent interface. In particular, it must implement the *getSource()* method that returns the event producer responsible for creating the event.

Processing an Event

Application writers that want to respond to an event must write an event consumer class that implements the EventConsumer interface. This interface declares the *handleEvent(ComergentEvent event)* method. The application writer must register the event consumer by adding it to the **events.xml** configuration file as follows:

```
<event class ="com.comergent.api.apps.application.ApplicationEvent">
   <description>
       This event is used to signal an event in this application.
   </description>
   <consumers-list>
       <consumer>
com.comergent.api.apps.consumingApplication.ApplicationEventConsumer
       </consumer>
       <consumer>
com.comergent.api.apps.consumingApplication2.DifferentEventConsumer
       </consumer>
   </consumers-list>
</event>
```

When an event is fired, each of the registered event consumers for events of this type is instantiated and each processes the event in the order they are declared in **events.xml** in turn until one of the consumers vetoes the event. At that point or when there are no more event consumers to be used, the EventBus returns control to the application that fired the event and the application continues. If no event consumer vetoes the event, then control is returned when the last consumer has completed processing the event.

events.xml DTD

Here is the DTD for the **events.xml** configuration file:

```
<?xml version='1.0' encoding='UTF-8' ?>

<!ELEMENT events ( event* ) >
<!ELEMENT event ( description?, consumers-list ) >
<!ATTLIST event
   class CDATA #REQUIRED
>
<!ELEMENT description (#PCDATA)>
<!ELEMENT consumers-list ( consumer* ) >
<!ELEMENT consumer (#PCDATA)>
```

Events

Every event must implement ComergentEvent interface and must also implement either the IREvent (for read-only events) or IRWEvent interfaces (for read-write events). Currently, both these interfaces declare no methods.

An event that implements the Vetoable interface is used in situations where you want to allow for the possibility that one event consumer can block other event consumers from processing the event. Once an event consumer has vetoed an event, then no other event consumers can process the event.

Vetoable Interface

This interface declares the following methods:

- public void veto(): used by an event consumer to veto an event.

- public boolean isVetoed(): used to test whether an event has been vetoed.

Example

In this section, we present a simple example of how to use events. The DispatchServlet class will produce an event that records the message type of the inbound request. Each such event will be consumed by a class that logs the message type to the log stream.

Event Classes

MessageTypeEvent

```
package com.comergent.dcm.core;

import com.comergent.api.eventbus.*;

public class MessageTypeEvent implements ComergentEvent, IREvent
{
    private MessageTypeEventProducer m_MessageTypeEventProducer;
    private String m_MessageTypeString;

    public MessageTypeEvent(MessageTypeEventProducer mtep, String s)
    {
        m_MessageTypeEventProducer = mtep;
        m_MessageTypeString = s;
    }

    public String getMessageType()
    {
        return m_MessageTypeString;
    }

    /**
     * Returns the sender of this message.
     */
    public EventProducer getSource()
    {
        return m_MessageTypeEventProducer;
    }
}
```

MessageTypeEventProducer

```
package com.comergent.dcm.core;

import com.comergent.api.eventbus.*;
import com.comergent.dcm.util.*;

public class MessageTypeEventProducer implements EventProducer
{
    public MessageTypeEvent createEvent(ComergentRequest cr)
    {
        String temp_MessageTypeString = null;
        try
        {
            temp_MessageTypeString = cr.getMessageType();
        }
        catch (ICCException icce)
        {
            //Log the fact that no message type could be retrieved
            temp_MessageTypeString = "NO MESSAGE TYPE RETRIEVED";
        }
        return new MessageTypeEvent(this, temp_MessageTypeString);
    }
}
```

MessageTypeEventConsumer

```
package com.comergent.dcm.core;

import com.comergent.api.eventbus.*;

public class MessageTypeEventConsumer implements EventConsumer
{
    public boolean handleEvent(ComergentEvent ce)
    {
        String temp_MessageTypeString =
            ((MessageTypeEvent) ce).getMessageType();
        Global.logInfo("Message type is " + temp_MessageTypeString);
        return true;
    }
}
```

DispatchServlet Changes

Add this code to the *executeController()* method of the DispatchServlet class:

```
MessageTypeEventProducer messageTypeEventProducer =
    new MessageTypeEventProducer();
MessageTypeEvent messageTypeEvent =
    messageTypeEventProducer.createEvent(comergentRequest);
EventBus.getInstance().fireEvent((ComergentEvent) messageTypeEvent);
```

The *fireEvent()* method throws an EventBusException, so be prepared to catch and handle the exception.

events.xml File

Add this event to the **events.xml** configuration file:

```
<event class ="com.comergent.dcm.core.MessageTypeEvent">
    <description>
        This event is used to record the message type of inbound
            requests.
    </description>
    <consumers-list>
        <consumer>
            com.comergent.dcm.core.MessageTypeEventConsumer
        </consumer>
    </consumers-list>
</event>
```

Testing the Example

Build the Comergent eBusiness System with these changes. When you start the system and log in, then you should see logging messages for each message type executed by the DispatchServlet.

Sending Email from the Comergent eBusiness System

This chapter describes how to generate an email message that can be sent from the Comergent eBusiness System. This functionality can be used to notify users of events such as the creation of a new invoice or to provide them with a confirmation such as the placing of an order.

Framework

In Release 6.4, two new mechanisms are used to send email messages:

- The OutOfBandHelper class is used to generate the body of email messages using JSP pages. See "OutOfBandHelper Class" on page 29 for an example of its use.

- The Event mechanism is used to determine which email is sent out when an event such as a user placing an order occurs. See CHAPTER 9, "Events" for more information on events.

This framework provides support for sending different email messages to users based on their locale. For example, you can use different JSP pages as the email templates for en_US and fr_FR users.

Typically, this is how these mechanisms are used:

1. An application invokes business logic to process a request. As part of the business logic, you want to send an email from the Comergent eBusiness System.

2. Define an event that will be used to initiate sending the email and declare it in the **events.xml** file. For example:

```
<event class="com.comergent.api.apps.orderMgmt.orders.-
    OrderPlaceEmailEvent">
    <description>This event is fired when an Order is placed
    </description>
    <consumers-list>
        <consumer>
            com.comergent.reference.apps.orderMgmt.orders.-
                OrderEmailEventHandler
        </consumer>
    </consumers-list>
</event>
```

3. Add code to fire the event from the application. See "Firing an Event" on page 112 for some sample code to do this.

4. Write the event consumer that will receive and process this event from the EventBus. This event consumer will actually send the email using the OutOfBandHelper class. For example, the OrderEmailEventHandler class marshalls data from the event, and then calls:

```
com.comergent.reference.apps.commerce.OutOfBandMailHelper.-
sendMail(from, to, cc, subject, messageType, requestAttributes);
```

The OutOfBandMailHelper class uses the OutOfBandHelper class to generate the body of the email message as follows:

```
ComergentRequest request = ComergentAppEnv.getRequest();
ComergentResponse response = ComergentAppEnv.getResponse();
ByteArrayOutputStream stream = new ByteArrayOutputStream();
// Now create an out of band helper
OutOfBandHelper outOfBandHelper =
    new OutOfBandHelper(request, response, stream);
Set keys = attributeMap.keySet();
Iterator iter = keys.iterator();
while (iter.hasNext())
{
    String key = (String)iter.next();
    Object value = attributeMap.get( key);
    request.setAttribute( key, value);
}
outOfBandHelper.getRequest().setAttribute(
    ComergentRequest.COMERGENT_SESSION_ATTR,
    request.getComergentSession());
```

```
//The message type is mapped to a JSP page using the standard
//message type mapping files.
outOfBandHelper.callJSP(messageType);
```

Current Usage of the Framework

Release 7.0.1 uses the framework described above. It uses the following files and data to configure the system:

- *Emails.xml: a number of configuration files are maintained in the **WEB-INF/emails/** directory. These map the email events to the form of the email that should be sent. For example, the **OrdersEmail.xml** configuration file includes the following element:

```
<EmailEvent Name="OrderChangeEvent">
    <EmailList>
        <Email>
            <From>@SMTP_SENDER@</From>
            <To>@OWNER@</To>
            <ContentType>text/html</ContentType>
            <Subject Bundle="com.comergent.reference.apps.order-
Mgmt.orders.OrdersResourceBundle">
            ORDER_CHANGE_SUBMITTED_MSG_SUBJECT
            </Subject>
            <Body>OrderEventEmail</Body>
        </Email>
    </EmailList>
</EmailEvent>
```

  This means that when an OrderChangeEvent is fired, an email should be sent to the owner of the order and that the subject line should be read from the OrdersResourceBundle properties file. Tokens can be used for system properties (such as SMTP_SENDER) and for request-specific properties (such as OWNER).

- The body of the email will be generated using the JSP page that the OrderEventEmail message type maps to:

```
<MessageType Name="OrderEventEmail">
    <JSPMapping>
        ../orderMgmt/Orders/OrderDownload.jsp
    </JSPMapping>
</MessageType>
```

- The JSP page used to render order email messages is **OrderDownload.jsp**.

Legacy Email Method

Release 6.4 introduced a new mechanism for generating email messages. This section describes the legacy method used in earlier releases.

Overview

When customizing a Comergent eBusiness System application or creating a new application, it is often desirable to generate email message and send it to one or more recipients. Java provides a basic collection of classes to do this, and the Comergent eBusiness System provides a framework of templates and token replacement to make the task simple for you.

Each email message is generated from a template which contains tokenized strings. These can be replaced by strings generated by the application. Typically, these are either retrieved as parameters from the application environment or are generated by the application itself from request and session data and the business logic used to generate the email message.

The email messages may contain URLs that recipients of the email can click to access more information through the Web UI to the Comergent eBusiness System. For example, when a user places an order, an email message is sent to them to confirm the order. The email message provides a link that points back to the order detail page, and so the user can review the order details simply by clicking the link in the message. See "Generating URLs" on page 121.

Templates

Most system-generated email is created by processing templates. Each template can be plain text or an HTML document, and can include tokenized strings along these lines:

```
...
<body>
<font face="arial" size="-1">
<p></p>
<p>Hello *FIRST_NAME* *LAST_NAME*</p>
<p>Greetings from *MANUFACTURER_NAME*</p>
<p>We thought you'd like to know that we received your order today.
</p>
<br>
Your order number is <b>*ORDER_NUMBER*</b> <br>
Your order has *ORDER_NUM_ITEMS* item(s): <br>
The following items were included in your order: <p>
...
```

The tokens are replaced as the template is processed. See "Example" on page 121 for code examples of the processing steps.

Generating URLs

URLs included in the email messages are generated by a call to the *constructAppURL()* method of the ComergentAppEnv class. For example:

```
protected String getOrderDisplayPage() throws ICCException
{
    Long cartKey = iro.getShoppingCartKey();
    String orderPage = null;
    try
    {
        orderPage = ComergentAppEnv.-
        constructAppURL(I18NOkayConstants.LOWER_CASE_DIRECT,
        OrdersConstants.ORDER_DISPLAY_PAGE,
        I18NOkayConstants.SHOPPING_CART_KEY + "=" + cartKey);
    }
    catch(Exception e)
    {
        throw new ICCException(e);
    }
    return orderPage;
}
```

This method generates a string that looks like this:

```
http://<server:port>/Comergent/en/US/direct/matrix?
    cmd=OrderDisplayPage&ShoppingCartKey=5
```

The parameters in the *constructAppURL()* method set the "direct", "OrderDisplayPage", and "ShoppingCartKey" sub-strings of the URL. The "Comergent" string is the Web application context which you may have changed as part of your implementation of the Comergent eBusiness System, and "matrix" is the value of the DefaultHostedPartner element in your **Comergent.xml** file.

Example

In this section, we present a simple example of how to use the email template mechanism. It follows the generation of the email used to confirm the placing of an order.

The first method call is invoked by the Java class that is processing the creation of an order. If the order is successfully created, then the class invokes *sendCreateOrderEmail()* to generate and send the email message. For example:

```
public void sendCreateOrderEmail(IOrder ior)
```

The method is called by passing in an IOrder object. This object provides the relevant user and order information required to retrieve the To email address and to build up the order details.

sendCreateOrderEmail() invokes:

```
notifyByEmail(ior, OrdersConstants.ORDER_CREATE_EMAIL_FILENAME,
com.comergent.dcm.util.ComergentI18N.getBundle(-
"com.comergent.reference.apps.orderMgmt.orders.-
OrdersResourceBundle").getString("ORDER_CREATE_MSG_SUBJECT"));
```

The *notifyByEmail()* method takes the IOrder object, together with the name of the email template to be used for the order creation template and a subject line retrieved from the appropriate resource bundle. Its body (simplified) looks like this:

```
boolean SEND_ORDER_EMAILS =
    Global.getBoolean(OrdersConstants.RULE_SEND_ORDER_EMAILS);
if (!SEND_ORDER_EMAILS)
    return;
String owner_email = ior.getAccOrder().getEmailAddress();
String mail_template = Global.getString(tfile);
String locale_mail_template =
    OrdersUtils.getLocaleSpecificTemplate(mail_template);
try
{
    OrderLocalEmail ole = (OrderLocalEmail)
        OMWrapper.getObject("com.comergent.apps.-
        orderMgmt.orders.bizAPI.OrderLocalEmail");
    String message = ole.formMessage(ior, locale_mail_template);
    String file_ext =
        OrdersUtils.getFileExtension(locale_mail_template);
    if(owner_email != null)
        OrdersUtils.SMTPSend(owner_email, null, subj,
            message, file_ext);
}
catch(Exception e)
{
    if(THROW_EMAIL_EXCEPTION)
        throw new ICCException("Unable to send email");
}
```

The key method is *formMessage()*. This method does the following:

```
protected String formMessage(IOrder inIor,
    String locale_mail_template) throws ICCException
{
    ior = inIor;
    iro = ior.getAccOrder();
```

```
email_orderCurrency = iro.getCurrencyLookupCode();
IPartnerMgrAPI iPartnerMgrAPI = (IPartnerMgrAPI)
    com.comergent.dcm.util.OMWrapper.getObject(
"com.comergent.api.apps.profileMgr.partnerMgr.IPartnerMgrAPI");
IRdPartnerHelper ph =
    iPartnerMgrAPI.getEnterprisePartnerHelper();
enterprisePartner = ph.getDataBean();
String msg = formEmailMessage(locale_mail_template);
String message = OrdersUtils.formatMessage(msg);
return message;
}
```

The key method is *formEmailMessage()*: it takes only the template as an argument, retrieving other data from the field variables set in the *formMessage()* method:

```
protected String formEmailMessage(String rawPOFileName)
    throws ICCException
{
    StringBuffer strBuf = new StringBuffer();
    try
    {
        String createFileName =
            ComergentAppEnv.adjustFileName(rawPOFileName);
        BufferedReader in =
            new BufferedReader(new FileReader(createFileName));
        String s = null;
        while((s = in.readLine())!= null)
        {
            String s1 = processLine(s);
            strBuf.append(s1);
            strBuf.append(I18NOkayConstants.STRING_NEWLINE);
        }
        in.close();
    }
    catch(Exception e)
    {
        OrdersUtils.logVerbose("Unable to form email message - " + e);
    }
    return strBuf.toString();
}
```

The processing work is done in the *processLine()* and *processLine_Repeat()* methods as follows:

```
protected String processLine(String s) throws ICCException
{
    String ret = s;
    int i = 0;
    while (true)
    {
```

```
        String foo = OrdersUtils.getToken(s);
        if (foo == null || foo.length() < 1)
        {
            return s;
        }
        s = processLine_Repeat(s, foo);
    }
}

protected String processLine_Repeat(String s, String foo)
    throws ICCException
{
    String ret = s;
    if(foo.equalsIgnoreCase(I18NOkayConstants.-
        STRING_MANUFACTURER_EMAIL))
    {
        String val = enterprisePartner.getEmailAddress();
        ret = OrdersUtils.replaceInstancesInString(s, foo, val);
    }
    else if(foo.equalsIgnoreCase(OrdersI18NOkayConstants.-
        EMAIL_TOKEN_ORDERNUMBER_STRING))
    {
        String email_OrderNumber = iro.getOrderNumber();
        ret = OrdersUtils.replaceInstancesInString(s, foo,
            email_OrderNumber);
    }
    ...
    else if (foo.equalsIgnoreCase(OrdersI18NOkayConstants.-
        EMAIL_TOKEN_ORDERRETRIEVALWEBPAGE_STRING))
    {
        String val = getOrderDisplayPage();
        ret = OrdersUtils.replaceInstancesInString(s, foo, val);
    }
    else
    {
        //Cannot find value for this token
        ret = OrdersUtils.replaceInstancesInString(s, foo,
            I18NOkayConstants.EMAIL_TOKEN_UNKNOWN_TOKEN);
    }
    return ret;
}
```

Note the call to the *getOrderDisplayPage()* method: this builds up the URL to the order. See "Generating URLs" on page 121.

CHAPTER 11 *Modularity and Generated Interfaces*

Release 7.0.2 of the Comergent eBusiness System has undergone the following architectural changes designed to make implementations easier to customize and upgrade:

- "Modules" on page 126

- "Generated Interfaces" on page 129

These changes are related in that the interfaces are organized by modules and that changes to interfaces may be contained to changes within individual modules.

Overview

The motivation to make these architectural changes are to ensure that customizations are more contained and can be better preserved during upgrade from one release of the Comergent eBusiness System to another.

By providing a means of delivering functionality in modules and by requiring that modules make calls to other modules only through their external interfaces, the following benefits are achieved:

- It is easier to compartmentalize the functionality of applications.

- It is easier to understand and manage the dependencies between parts of the Comergent eBusiness System.

- It is easier to contain the customizations to single modules and understand what effect changes made in a module have on the whole system.

- Modules can be more easily upgraded independently of each other, minimizing the effect that an upgrade may have.

- Upgrades to modules that have not been customized will not effect customizations made in other modules.

- New functionality can be delivered in the form of a module that may be dropped into an existing deployment of the Comergent eBusiness System.

Modules

The Comergent eBusiness System is developed as a set of interdependent modules that conform to a common organizational structure. In general, each module corresponds to a functional component of the Comergent eBusiness System such as an application or a component of the Comergent eBusiness System platform. Some modules may support both a Java API and a user interface whereas other may just support a Java API provided to other modules. Some modules provide a set of "helper" classes, JSP pages, and other files such as Javascript files and images which are used by a number of other modules.

In general, each module has the following structure:

- Java classes: organized into three trees. At build time, the directories for all of the modules are assembled in to a single tree under the com.comergent package.

  - external API interfaces: used by other modules to access functionality provided by the module. In general, when one module makes a call to another module's class, it must do so through the other module's external API. This is the com.comergent.api package for the module.

  - implementation classes: the implementation of the external API interfaces. When another module makes a call to the module's external API, then the actual classes used are the implementing classes of the module's interface. The implementation packages may include internal classes: used by the implementation classes, but not exposed to outside world and not part of the supported Javadoc. This is the com.comergent.apps or com.comergent.appservices package for the module.

  - reference components: Controller classes and JSP pages always comprise part of the reference implementation and their source is provided with the Comergent eBusiness System. Resource bundles are also provided as part

of the reference. This is the com.comergent.reference package for the module.

- JSP pages: possibly organized into directories depending on the organization of the module. They should always access other modules' classes through the external APIs exposed by the other modules. This ensures that JSP pages can be re-used from release to release provided that the external APIs are supported.

- Resource bundles, Javascript, and static files (such as images and HTML fragments).

- Configuration files specific to the module such as **MessageTypes.xml** files and business rules.

Module Interfaces

Each module must provide an external interface so that all calls to Java classes and interfaces within the module are invoked through the interface. This external interface provides a comprehensive set of Javadoc pages so that writers of other modules can use the external interface reliably and easily.

The external interface for each module will typically be a combination of handcrafted interfaces and automatically-generated interfaces. Most modules provide handcrafted interfaces for presentation beans that enable presentation beans to manipulate data beyond the simple accessor methods of the generated data bean interfaces. The presentation beans usually wrap a data bean and implement the same interfaces, but in addition they implement helper methods and some business logic.

The external interfaces are organized under the following main packages:

- com.comergent.api: this package has all the module external APIs. These are organized into:

  - apps: these are the application hand-crafted APIs. Typically, these are presentation bean interfaces, utility interfaces, and factory classes.

  - appservices: these are the packages provided by the service modules used by other applications.

  - dcm: these are the external APIs offered by the Comergent eBusiness System platform.

- com.comergent.bean.simple: this package has all the automatically-generated bean interfaces and the data bean classes themselves.

The generated interfaces are provided for each of the data objects declared in the XML schema files. These are organized to provide appropriate levels of access to the data fields of the underlying data beans. This helps to ensure that there is a clearer separation between presentation and business logic in the Comergent eBusiness System. See "Generated Interfaces" on page 129 for more information about the generated interfaces.

Invoking Interfaces

You can invoke an interface from a Java class by casting any object or child interface to the interface and then invoke any method that the interface declares. In the Comergent eBusiness System, use one of the following techniques to do this:

- "Using the Object Manager" on page 128

- "Using Factory Classes" on page 129

Each module uses one or other of these techniques, but not both. As you work on an existing module or create a new one, be consistent in how you invoke the interfaces. It will make it easier for your colleagues to work on the same module.

In general, you should always try to work with interfaces provided by the com.comergent.api packages: these are the interfaces that the modules will support from one release to the next, even though the underlying implementations of the interfaces may change.

Using the Object Manager

You can use the ObjectManager class to return an appropriate interface as follows. Suppose that you want to retrieve the IAccProduct interface to set the data fields of a product. Then make a call along these lines:

```
IAccProduct temp_IAccProduct =
    (com.comergent.bean.simple.IAccProduct)
        com.comergent.dcm.util.OMWrapper.getObject(
            "com.comergent.bean.simple.IAccProduct");
```

Provided that there is an entry in the **ObjectMap.xml** file that specifies the object to be returned and provided that the object implements the IAccProduct interface, then this call will succeed and methods on the interface can be invoked. For example, if the **ObjectMap.xml** file contains:

```
<Object ID="com.comergent.bean.simple.IAccProduct">
        <ClassName>com.comergent.bean.simple.ProductBean</ClassName>
```

Then, the ObjectManager returns a com.comergent.bean.simple.ProductBean object and this can be cast to the IAccProduct interface because the com.comergent.bean.simple.ProductBean class implements the com.comergent.bean.simple.IAccProduct interface.

Using Factory Classes

Calls to an interface can be provided by Factory classes that return an instance of the interface. For example, the package com.comergent.api.apps.commerce provides a public interface IInquiryListFactory. If another module needs an instance of this Factory interface, then it calls the CommerceAPI class's *getFactory(int i)* method. The int parameter determines what sort of Factory class is returned. In turn, the calling module can now invoke methods on the IInquiryListFactory to return inquiry list interfaces of the appropriate type. For example,
getInquiryList(Long listKey, boolean bFillPrices) returns an object that implements the IInquiryList interface.

Generated Interfaces

When you need to access data on a particular data object, you must use the generated interfaces that each data object provides. These generated interfaces are created and compiled when the SDK generateBean target is run as part of the deployment of your Comergent eBusiness System.

For each data object declared as a DataObject within the **DsRecipes.xml** file, and for any parent, reference, or child data objects, the following classes and interfaces are generated and compiled in the com.comergent.bean.simple package:

- *<Name>*.java: this is the data bean class. It implements the interfaces listed here. In addition, if the data object extends another data object, then the bean extends the *<Parent>*.java bean.

- IAcc*<Name>*.java: this interface extends the IRd*<Name>*.java by providing the write (set) accessor methods on all of the data fields of the data object. In addition, if the data object extends another data object, then the IAcc interface extends the IAcc*<Parent>*.java interface.

- IData*<Name>*.java: this interface extends the IAcc*<Name>*.java by providing *restore()* and *persist()* methods on the data object. In addition, if the data object extends another data object, then the IData interface extends the IData*<Parent>*.java interface.

- IRd<*Name*>.java: this interface provides the read-only (get) accessor methods to the data fields of the data object. In addition, if the data object extends another data object, then the IRd interface extends the IRd<*Parent*>.java interface.

- In addition, list beans also implement the IData<*Name*>List.java interface. Each list interface extends the IDataList.java interface as well as the list interface of any parent object.

In general, you should use the IRd interface for any objects to be passed to JSP pages so that the objects are effectively read-only. Only use objects that implement the IData interface when you know that you need to either restore or persist the data object.

Example of a Generated Interface

Consider the ACL data object: the **ACL.xml** file reads:

```
<?xml version="1.0"?>
<DataObject Name="ACL" Extends="C3PrimaryRW"
    ExternalName="CMGT_ACLS"
    Access="RWID" Ordinality="1"
    ObjectType="JDBC" Version="5.0">
    <KeyFields>
    <KeyField Name="AccessKey" ExternalName="ACL_KEY"
        KeyGenerator="ACLKey"/>
    </KeyFields>
    <DataFieldList>
        <DataField Name="AccessKey"
            Writable="n" Mandatory="n"
            ExternalFieldName="ACL_KEY"/>
        <DataField Name="ACLName"
            Writable="y" Mandatory="n"
            ExternalFieldName="NAME"/>
    </DataFieldList>
    <ChildDataObject Name="Access" />
</DataObject>
```

Consequently, the IRdACL.java class declares:

```
public interface IRdACL extends IRdC3PrimaryRW
```

and exposes the methods:

- public Long getAccessKey();

- public String getACLName();

The IAccACL.java class declares:

```
public interface IAccACL extends IAccC3PrimaryRW, IRdACL
```

and exposes the methods:

•   public void setACLName(String value) throws ICCException;

•   public void addAccess(AccessBean bean) throws ICCException;

The IDataACL.java class declares:

```
public interface IDataACL extends IAccACL,IDataC3PrimaryRW, IData
```

In general, this interface may declare no additional methods beyond those declared in the IData interface because all the standard methods to read and write data from external data sources are declared in this interface.

CHAPTER 12

Implementing Logic Classes

This chapter and the next two chapters present a description of how to implement business logic classes (BLCs) at an implementation of the Comergent eBusiness System. Before reading this chapter, you must have a working understanding of the basic architecture of the Comergent eBusiness System and of Java.

> **Note:** The use of BLCs is deprecated. In general, new applications should use bizlets, controllers, and BizAPIs to implement their business logic.

Key Concepts

To understand fully how the Comergent eBusiness System works as an application, you must understand its architecture.

An installation of Comergent eBusiness System processes requests as they are received from users' browsers, and messages from other Comergent eBusiness Systems and from external systems. You must configure the Comergent eBusiness System to process each type of request and message.

The core of the Comergent eBusiness System is the *C3* Commerce Manager. This powerful and flexible server is designed to seamlessly integrate a network of channel partners and the external systems that make up the e-commerce environment of each partner.

Each Comergent eBusiness System server in the network of sales partners works both as a server in relation to inbound requests from browsers and as a client as it retrieves information from other Comergent eBusiness System servers and external systems.

To customize the Comergent eBusiness System in your environment, you need to consider how the system retrieves data from your external systems. In general, you can use the schema and Service classes to retrieve data from a local database source or from another Comergent eBusiness System server by exchanging messages. However, you have to produce custom BLCs to retrieve information from an external system other than these.

Application Logic Classes

Application logic classes are implemented as bizAPI, business logic , or controller classes.

- bizAPI classes are used to manage the business logic of business objects. Conceptually, each bizAPI class corresponds to a business object and its methods correspond to the actions that can be performed on the business object. For example, the OrderInquiryList bizAPI class provides the following methods: *duplicate()*, *copyLineItem()*, and *changeOwner()* which correspond to actions that can be performed on a product inquiry list. It implements the com.comergent.api.apps.orderMgmt.oil.IOrderInquiryList interface.

  The bizAPI classes are defined in the com.comergent.apps.*<application>*.bizAPI packages. Typically, they implement an interface declared in the corresponding com.comergent.api.apps.*<application>* package.

  For example, the Order bizAPI class is in the com.comergent.apps.orderMgmt.orders.bizAPI package. It extends the OrderInquiryList class and implements the com.comergent.api.apps.orderMgmt.orders.IOrder interface.

- Each BLC is a subclass of the BLC abstract class. This class implements the ApplicationObject interface. BLCs perform the business logic of your implementation of the Comergent eBusiness System. Each BLC contains a table of business objects such as session, user, and shopping cart for example. In executing the *service()* method of a BLC, it invokes the *persist()* and *restore()* methods of these business objects.

> **Note:** In general, the use of BLC classes is deprecated. You should use either controllers or bizAPI classes to manage your business logic.

- Some Comergent eBusiness System use controller classes to perform business logic. These classes are to be found in the com.comergent.reference.apps.<*application*>.controller packages for each application.

The Comergent eBusiness System comes with a number of standard bizAPI classes, BLCs, controllers, and JSP pages. However, you may need to create new logic classes or modify the existing classes. CHAPTER 13, "Implementing Application Logic Classes" provides guidance on how to do this.

Business Objects

See CHAPTER 6, "Introducing Data Beans and Business Objects" for more information.

XML Schema

The *Comergent eBusiness System Reference Guide* covers the schema and Service classes in detail. You should manage data access using these features if possible.

Naming Service

To retrieve parameters at runtime, the Comergent eBusiness System provides a naming service to access either a flat file or a database to recover parameters.

Application logic classes can invoke a naming service by calling the static class NamingManager methods *getInstance()* and *getInstance(int i)*. Both these methods return an object that implements the NamingService interface.

- If no integer argument is provided, then an object of default type is created, either a NamingServiceProperties object or a NamingServiceDatabase object.

- If the integer argument is the constant NamingManager.DATABASE, then a NamingServiceDatabase object is created.

- If the integer argument is the constant NamingManager.PROPERTIES, then a NamingServiceProperties object is created.

- If the integer argument is not one of these two, then an object of default type is created.

In all cases, the Comergent eBusiness System accesses the **Comergent.xml** file to determine exactly how the NamingService object should be created:

- If a NamingServiceDatabase object is to be created, then the NamingManager.database entries are used to specify the connection to the database.

- If a NamingServiceProperties object is to be created, then the NamingManager.properties entry is used to determine which properties file holds the parameter values.

Once the NamingService object is created, you use the methods listed below to retrieve the parameters as a NamingResult class:

- public NamingResult get(int key)

- public NamingResult get(Long key)

- public NamingResult get(String key)

The key parameter provides a means of retrieving only those parameters whose name begins with the key string.

The NamingResult class provides the *get(String parameter)* method to return the value of the parameter.

NamingService Example

For example the following code fragment recovers the value of the message URL parameter for a distributor referred to by its partner key.

```
NamingService namingService = NamingManager.getInstance();
NamingResult namingResult = namingService.get(partnerKey);
String url = namingResult.get(NamingResult.MESSAGE_URL);
```

Note that by default, the type of NamingService created is a NamingServiceDatabase object because in **Comergent.xml** the NamingManager defaultType element is set to "database".

Implementing Application Logic Classes

This chapter presents a detailed description of how to implement application logic classes at an installation of Comergent eBusiness System. Before reading this chapter, you must have a working understanding of the basic architecture of the Comergent eBusiness System and of Java. It covers:

bizAPI Classes

You can manage the business logic of an application by creating bizAPI classes. They must be invoked from the controller class specified by the message type of the request.

There is no standard interface or abstract class for bizAPI classes. Typically, each bizAPI class corresponds to a business object such as a product inquiry list or a quote. It provides methods that correspond to the actions that you can perform on the business object. It may also provide helper methods to support the business logic such as tax calculations or looking up currency codes.

Business Logic Classes

The use of BLC classes is deprecated in Release 6.4 and later. See "Business Logic Classes" on page 464 if you need to support a legacy use of business logic classes.

Controller Classes

Every inbound request received by the DispatchServlet is processed by a controller: the message type determines which controller class is used. There are some lightweight controllers such as ForwardController and SimpleController. These make use of the **MessageTypes.xml** files to determine the JSP page to which the request is forwarded.

Most requests are handled by a custom controller: the message type determines which controller class is used. Each custom controller must override the *execute()* method declared by the Controller class: it is called by the DispatchServlet when the servlet container receives the request. This method executes the business logic required to process the request either by itself or through invoking bizAPI classes.

A typical *execute()* method looks like this:

```
public void execute() throws ControllerException, ICCException,
    IOException
{
    //Get the cart key
    Long cartKey = getCartKey();
    //If from update update the order
    ShoppingCart updatedCart = (ShoppingCart) updateCart(cartKey);
    Hashtable supersededSKUsFromAdd = getSupersededSKUs();
    //Get order factory
    try
    {
        boolean bShowPromos = true;
        String szShowPromos = ComergentAppEnv.getEnv().-
            getProperty(I18NOkayConstants.SHOW_PROMO_BUSINESS_RULE);
        if ((szShowPromos != null) &&
            (szShowPromos.equalsIgnoreCase(
                I18NOkayConstants.STRING_FALSE)))
        {
            bShowPromos = false;
        }
        ChannelCartPresentationBean cartDisplay =
            (ChannelCartPresentationBean)
                getPresentationBean(cartKey, bShowPromos);
        /*
```

```
                            If this is an update message for addSKUs, then  fill it
                            with vector for bad SKUs
                    */
                    cartDisplay.setUpdate(isUpdate);
                    cartDisplay.setOperation(updateOperation);
                    cartDisplay.setAddSKUErrors(vBadSKUs);
                    cartDisplay.setSupersededSKUs(supersededSKUsFromAdd);
                    //Set the beans for the request
                    request.setAttribute("cartPresentation", cartDisplay);
                    callJSP();
            }
            catch( Exception e)
            {
                throw new ICCException( e);
            }
    }
```

CHAPTER 14   *Software Development Kit*

You can use the Comergent eBusiness System Software Development Kit (SDK) to install and customize your implementation of the Comergent eBusiness System. The HTML documentation provided with each version of the SDK is intended to provide you with a basic overview of how the SDK works and its use to manage projects. This chapter is devoted to describing the basic structure of a customization project. In particular, follow the guidelines here to organize your project so that it follows the customizations guidelines.

Project Organization

Each project built using the SDK is created on top of a release of the Comergent eBusiness System. When you create the project using the newproject target, the SDK creates a set of project files that are suitable for that release. All of the customizations that you make in the project are made by adding files to the project. Files can be added to the project in these ways:

- Use the customize target to copy a file from the release into the project. When you use the customize target, the file is automatically copied into the appoprate sub-directory of the project.

- Create the file manually in the appoprate sub-directory of the project.

See "Project File and Directory Locations" on page 142 for information on where files must be located.

Project File and Directory Locations

In this section, we assume that you have created a project called *project*, and so you will have a project directory called ***sdk_home*/projects/*project*/**. You must make sure that each of the project files is in the appropriate location under the project directory as follows:

• Java source files: these must be placed under the ***project*/src/** directory, and follow the package organization for the Comergent eBusiness System.

• JSP pages: in this release, the JSP pages are organized by module and locale under the ***project*/WEB-INF/web/** directory.

• Schema files: these comprise the data object files, and the supporting data services files. All your customizations should be maintained under the ***project*/WEB-INF/schema/custom/** directory. Make sure that the schemaRepositoryExtn element is set to "WEB-INF/schema/custom".

Java Source Files

Under the ***project*/src/** directory, we suggest that you follow these guidelines to organize your customizations to the Comergent eBusiness System:

• Use the com/comergent/api/ packages to add your extensions to the Comergent eBusiness System API. In general, you should create new classes that extend the existing API: do not overwrite the release API because that can affect any upgrade.

• Use the com/comergent/apps/ and com/comergent/appservices/ packages to add implementation classes: these may be entirely new classes or new classes that extend existing implementation classes.

• Use the com/comergent/reference/ packages for controller classes. You can customize existing controller classes or create new controller classes.

JSP Pages

Under the ***project*/WEB-INF/web/** directory, we suggest that you follow these guidelines to organize your customizations to the Comergent eBusiness System:

• Where appropriate, use the existing organization of JSP pages to add new JSP pages or to customize existing ones.

• If you are adding a new functionality module, then create a new directory under the appropriate locale(s) for the module, and follow the same naming conventionas you do for Java classes created for the module.

Schema Files

Under the ***project*/WEB-INF/schema/custom/** directory, we suggest that you follow these guidelines to organize your customizations to the Comergent eBusiness System:

- Add new data objects by:

  - Put the XML definition of the data object in ***project*/WEB-INF/schema/ custom/**. For example, create the file ***project*/WEB-INF/schema/custom/ Comment.xml**

  - Modify ***project*/WEB-INF/schema/custom/DsBusinessObjects.xml** by adding the new business object. For example:

```
<?xml version="1.0"?>
<Schema Name="project" Description="project Custom Schema"
    Version="6.0">
    <BusinessObject Name="Comment" Version="6.0"
        Description="Comment BusinessObject"/>
</Schema>
```

  - Modify ***project*/WEB-INF/schema/custom/DsDataElements.xml** by adding the new data elements for the header and list data objects, together with any new fields declared by the data object. For example:

```
<?xml version="1.0"?>
<Schema Name="project" Description="project Custom Schema"
    Version="6.0">
    <DataElement Name="Comment" Description="Comment data object"
        DataType="HEADER"/>
    <DataElement Name="CommentList" Description="Comment list data
        object" DataType="HEADER"/>
    <DataElement Name="CommentKey" Description="Comment Key"
        DataType="LONG" MaxLength="20"/>
</Schema>
```

  - Modify ***project*/WEB-INF/schema/custom/DsRecipes.xml** by adding a recipe element. For example:

```
<Schema Name="project" Description="project Custom Schema"
    Version="6.0">
    <Recipe Name="Comment" BusinessObject="Comment"
        Description="Default Approvals List Recipe" Version="6.0">
        <DataObjectList>
            <DataObject Name="Comment" Access="RWID"
            DataSourceName="ENTERPRISE" Ordinality="n"
            Version="6.0"/>
        </DataObjectList>
    </Recipe>
```

```
</Schema>
```

- Modify the appropriate key generator file, for example ***project*/WEB-INF/ schema/custom/OracleKeyGenerators.xml**, by adding any new keys required:

```
<?xml version="1.0"?>
<Schema Description="project Custom Schema" Name="project"
    Version="6.0">
    <KeyGenerator Name="CommentKey" KeyProcedureName="COMMENTKEY"
        GeneratorType="PROCEDURE" />
</Schema>
```

CHAPTER 15   *Tailoring the Comergent eBusiness System*

Overview

Embracing all channels, all selling partners, and all products, the Comergent eBusiness System enables the most powerful collaborative commerce experience available. With the Comergent eBusiness System, companies can successfully pursue e-business strategies by seamlessly integrating their existing channel partners into Web commerce, offering vast opportunities for their partners to contribute value-add during the selling process. By guiding customers through the entire sales process, streamlining the selection of complex products and services, allowing purchases to be consummated across channel partners, and providing insight into the effectiveness of commerce activities, Comergent Technologies maximizes e-business sales potential while strengthening relationships with a company's channel partners.

The Comergent eBusiness System is a complete suite of enterprise-class applications developed using open, standards-based technologies such as Java, XML, and Java 2 Enterprise Edition (J2EE). The Comergent eBusiness System is not a collection of code stubs or building blocks to help jump-start the custom development of complex e-commerce applications. The Comergent eBusiness System applications are pre-packaged solutions that are ready to deploy, easy to configure, and easy to administer. The Comergent eBusiness System enables

companies to rapidly implement technology solutions to business problems and reduce time to market for a competitive advantage.

The Comergent eBusiness System is deployed as a Java J2EE Web application, using the services provided by commercial Java application servers. This provides an open foundation for the integration of Comergent eBusiness System with other applications as well as the extension of Comergent eBusiness System functionality. In addition, the Comergent eBusiness System application architecture itself has been designed to meet the key goals of extensibility and integration.

A key requirement for any Web-based enterprise application is the ability to configure, customize, and extend the application to meet the needs of the customer. The Comergent eBusiness System has been designed throughout to support customization, extension, and integration with other applications. This document describes how this is accomplished.

Customization Components

The following components and source code are delivered as part of the SDK installation of the Comergent eBusiness System to allow for the capability to make specific extensions and customizations:

- **Presentation Layer Components**: JavaServer Pages (JSPs) Template Source Code Files, Cascading Style Sheets (CSSs), Java Resource Bundle Files, Javascript files, Java source code for Controllers (as in the Model/ View/Controller paradigm, see "Controllers" on page 150) defining end user page flows.

- **DataServices Layer Components**: Business Object Schema Definition Files, Knowledgebase Database Definition Scripts, Document Type Definition (DTD) Files for all XML messages supported by the system

- **Application and Kernel Configuration Files**: The XML configuration files are used to modify the behavior of the Comergent eBusiness System platform.

- **API and Developer Documentation**: Developers Guide, Administration Guide, Implementation Guide, and Reference Guide PDF files. The JavaDocs, the industry standard HTML-based documentation format, generated directly from the source code detailing the input/output parameters and usage guidelines for all public and protected methods in all Comergent Technologies packages is provided along with a higher level index tying together Presentation, Logic, and Data Layer components.

Platform Components

The shaded areas in the diagram below represent the internal architectural modules of the Comergent eBusiness System system. These modules are documented and serve as the platform upon which applications are written.

*   **Comergent eBusiness System Platform**: This module represents the underlying platform-level mechanisms of the Comergent eBusiness System architecture, including the Data Services module, the system-to-system messaging manager including XSL/XSLT-based conversion services, and the security, authentication and data entitlement system.

*   **Comergent eBusiness System Application Engine**: This module represents the supporting framework for Comergent eBusiness System applications. It defines the Java Interfaces that must be implemented by the various business logic modules associated with applications, provides the mechanism for interfacing between JSP pages and the underlying business logic, and provides common application services that are used by the application logic.
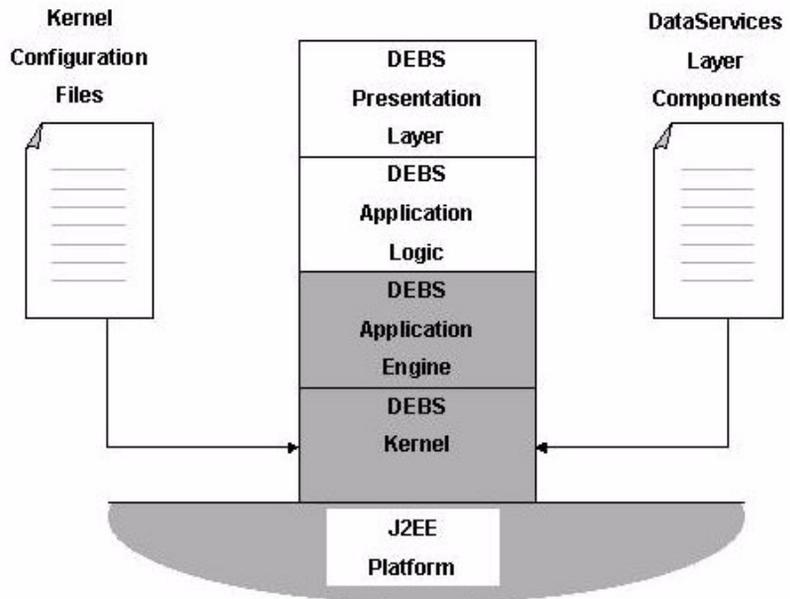
FIGURE 6. Comergent eBusiness System Architecture

Extensions and Maintenance

This section describes in greater detail the various mechanisms within the Comergent eBusiness System environment that support the customization and extension of the application suite.   The source code and customization components detailed on the previous page are sufficient to extend the application when applied in the following manner:

Extending the Presentation Layer

The Comergent eBusiness System application suite is deployed as part of the overall Web site of a company and as such must be able to be customized to integrate with the look-and-feel of that site.

Look-and-feel customizations are achieved by making changes to the JSP pages for each application page, by editing the resource string files, and by adjusting the font size and color specifications in the cascading style sheet file.

Page flow customizations are achieved through modification of server-side Java components known as "Controllers" which contain the reference page flow logic.

JSP Pages

All Comergent eBusiness System applications support a "thin client" user interface based on the HTML Web standard. Users connect to Comergent eBusiness System applications using a standard Web browser; no special software installation or download is required at the client side. One of the key benefits of HTML is its flexibility. Web-based user interfaces can be created that satisfy a large variety of look and feel requirements. In addition, adhering to the thin-client HTML model of user interface ensures that Comergent eBusiness System applications can be smoothly integrated into the appearance and functionality standards of any Web site.

The Comergent eBusiness System user interface is implemented using the Java Server Pages (JSP) standard, defined as part of the overall Java 2 Enterprise Edition (J2EE) standard. Each screen within the Comergent eBusiness System user interface is defined by a JSP page. JSP pages define a mechanism for interleaving HTML formatting directives with Java source code. This provides a way to support both the dynamic content (for example, an order's line-item list) and the static content (the site navigation buttons) of the page in a single source file.

All Comergent eBusiness System applications are implemented using "Model 2" JSP pages, which means that there is a strict separation between the business logic of an application screen and its presentation to the user. The business data and logic

are made available to the JSP page through the underlying Comergent eBusiness System architecture, but the JSP page itself contains only formatting directives and control structure. This provides a very flexible user interface infrastructure, supporting conditional display of text, etc. while still keeping the JSP pages clean and simple. The JSP pages can be customized to any look and feel by administrators who are HTML capable but have little or no Java programming experience.

All Comergent eBusiness System JSP pages make extensive use of the HTML Cascading Style Sheets (CSS) mechanism. This allows most look-and-feel parameters (font, color, spacing, size, and so on) to be defined in a separate CSS file and referenced by name in the JSP pages themselves. This allows the appearance of the page to be significantly modified via changes to the CSS file, without modifying the JSP pages themselves.

Comergent Technologies supplies source code template files for each Java Server Page as part of any implementation. This allows any customer to control the look-and-feel of the Comergent eBusiness System applications at their site, both upon initial implementation and over the life-cycle of the site.

Page Layout and Branding

Overall page layout and appearance, including background colors and branding images, can be modified by editing the JSP page(s) associated with each Web page. The JSP page consists primarily of HTML text that specifies the page layout and structure. Components on the page can be re-sized or rearranged by editing the HTML.

All page image references can be changed to new graphics files containing the appropriate pictures and company logos.

Resource Strings

All text strings defined by the applications, including error message text are defined in resource files that are separated from the application code. These strings can be modified by editing the Java Resource Bundle implementations. Java Resource Bundles are a standard mechanism within the Sun Java Development Kit to support internationalization efforts.

Style Sheets

All aspects of text font style and color are controlled via the HTML Cascading Style Sheets mechanism. The style sheet files define HTML element classes that specify the font, text color and background colors of text displayed within the pages.

A new graphic design can be applied to all the pages simply by editing the style sheet file to assign new font style size and color attributes to the various elements displayed within the page.

Page Flow

It is sometimes desirable to alter the page flow of a Comergent eBusiness System application or to incorporate the services or capabilities of an outside application into the Comergent eBusiness System page flow. This can be accomplished in a number of ways.

HTTP Links

The URL linkages between pages can be edited within the JSP template itself. A new button can be created that invokes an alternate application functionality and page flow, or the URL of an existing command button can be edited to reference a different page. In addition, in-line Java code can be inserted in a JSP page that can include arbitrary Java packages and invoke the incorporated Java code to call out to external systems or provide information to be displayed within the page.

Controllers

Comergent eBusiness System JSP pages follow the Model 2 standard, implementing the "Model/View/Controller" (MVC) paradigm that helps to cleanly separate business logic from presentation. Controller code is typically responsible for:

• validating and converting HTTP request parameters, usually form posts, to the appropriate data objects on which the Comergent eBusiness System application logic acts;

• matching application results to specific JSP pages for presentation to the user;

• performing any conditional routing of page flow based on normal or exception results returned or raised by requested application logic.

Controllers are essentially lightweight orchestrators of logic request flows initiated via Web browsers, bridging Web clients to the presentation independent Comergent eBusiness System application logic.

New controllers can be written as part of a Comergent eBusiness System installation that can alter the flow between existing Comergent eBusiness System application pages without modifying any of the underlying business logic implemented by the operations within a page. Further, source code for the Controllers user for end user page flows can be modified appropriately. Controller

implementers must limit themselves to documented Comergent eBusiness System APIs to achieve compatibility with future releases of the Comergent eBusiness System.

Extending the Data Services Layer

All access to business information or data is mediated through the Comergent eBusiness System Data Object mechanism. The data object definition mechanism (described in CHAPTER 6, "Introducing Data Beans and Business Objects") can be used to extend these objects to contain additional information.

Data Object Schemas

All business information that is used and managed using Comergent eBusiness System applications (for example, orders, user profiles, product data) is accessed through a set of data objects defined by the applications. These data objects define the various readable and writable fields and attributes of business data.

Each data object is defined by an XML schema file that defines the field names and data types of the data object. A data object can be customized through sub-classing or by editing its associated schema file to add new fields and delete or rename existing ones. The sub-classing mechanism isolates customizations in a new XML schema file. This is the most upgrade-friendly approach and, therefore, the recommended one.

The fields of a data object can be accessed by name from within the UI code in a JSP page. This means that changes to a data object can easily be reflected in the UI of a Comergent eBusiness System application without any Java coding.

Extending a Data Object

An existing Comergent eBusiness System data object can be extended by editing its XML schema definition file. For example, a new data field can be added to an existing data object by inserting the appropriate XML element(s) in the schema definition file.

The XML schema file is then processed to produce the associated Java class for the data object and to generate the Java bean that can be referenced by JSP pages that interact with that data object.

Presentation code within a JSP page can reference the new data object data field just like any built-in field of the object. The value can be read and displayed or set from a form field, all without requiring any programming outside the XML schema file, controllers, and the JSP page.

Creating a New Data Object

New data objects can be created with the same XML-based mechanism. A new data object's content can be referenced through the automatically-generated Java bean interface by any interested JSP page and Controller pair.

We recommend that when you create a new data object, you create a new database table to store the corresponding data. Name the table with a prefix that reflects the project title rather than the standard CMGT prefix. This ensures that there is no risk of a naming collision when upgrades take place.

Data Source Mapping

The Comergent eBusiness System Data Services module is responsible for managing all interaction between the Comergent eBusiness System applications and various data sources, including relational databases, XML messaging, and data management system APIs.

The Data Services module provides a variety of built-in data interface modules. One provides access to relational databases via the Java JDBC standard. Another provides access to remote data via system-to-system XML messages.

Each data interface module is built using the module API provided by the Data Services mechanism. New data interface modules can be written using this API to access data that is managed by various legacy systems or other mechanisms.

XML Message DataService

The set of XML messages supported by the Comergent eBusiness System application suite is defined via a message map definition file. The message map file can be edited to define new messages that can be mapped to existing or new business logic.

System-to-System Messages

The Comergent eBusiness System architecture supports integration with other applications through the exchange of XML messages via the HTTP protocol. Any business logic operation provided by a Comergent eBusiness System application can be invoked using this XML interface. The specific operation to be performed along with all of its required input parameters are specified as part of the XML structure defined by an XML Document Type Definition (DTD).

Comergent eBusiness System also defines a set of XML messages to support pre-defined business processes required by the applications. One example is the remote pricing and availability check performed between a manufacturer and a distributor site. Another is a catalog content syndication export. These business messages are defined via a Comergent eBusiness System-specific XML message that defines all

possible aspects of the interaction. These "master messages" can then be converted into different specific XML syntax structures via the use of the XSLT standard that provides for the transformation between XML document structures.

The set of XML operations is defined as part of the Comergent eBusiness System system configuration in the "message map" file. New messages can be defined and existing messages modified or extended by changing the entries in the message map.

Message Syntax Conversion

Comergent eBusiness System supports the use of the XML style sheet translation mechanism to convert messages of one XML syntax into an alternate syntax. Thus, all existing Comergent eBusiness System messages supporting various business processes can be mapped into the specific XML syntax of a variety of XML standards that support those same operations.

Simultaneous Support for Multiple Message Formats

Comergent eBusiness System can also keep track of which syntax structure or message version is supported by the various business partners of a Comergent eBusiness System installation and simultaneously exchange different message sets with different partners.

Extending the Application Logic Layer

Occasionally in the course of implementing a Comergent eBusiness System e-commerce site it is necessary to create a new business logic module for some operation specific to that customer's business. Comergent eBusiness System supports the definition of new business logic classes and the extension of existing classes using the Java interfaces defined by the application architecture and even through the sub-classing of existing business logic classes.

Application Logic

The Comergent eBusiness System application suite functionality is implemented via a set of Business Logic Classes defined as Java class objects, known as the Comergent eBusiness System Application Logic. See CHAPTER 13, "Implementing Application Logic Classes" for more information. The core of the Comergent eBusiness System is a collection of business logic class implementations that cover a wide variety of operations.

Occasionally in the course of deploying the Comergent eBusiness System system it is necessary to implement new business logic or modify existing business logic operations. This is easily accomplished by writing new Java code within the overall Comergent eBusiness System framework.

Code implementing new business logic must be written to conform to the overall set of Java interfaces defined by the Comergent eBusiness System architecture. These interfaces are documented in the reference documentation for the product. New business logic operations can be invoked from JSP pages or using XML messages, just like the built-in business logic of the applications.

The definition of all the Comergent eBusiness System Java packages, classes, and methods can be found in the product Javadoc. Javadocs are an industry standard format, developed by JavaSoft, for displaying the internal input/output parameters and usage guidelines.

There are cases where it may be desirable to modify existing application business logic. This can be accomplished by defining a subclass of an existing business logic class. The subclass can perform additional or alternate operations, invoking the pre-defined business logic operation at the appropriate time. Definitions for understanding how to correctly subclass the existing business logic classes can be found in the Javadoc. The source code for the Application Logic can also be referenced for this purpose as well, if necessary.

Writing New Application Logic

The Comergent eBusiness System Application Logic class interface defines the execution environment for a new piece of business logic, including the parameter definitions and values (from either a Web page form or a system-to-system XML message), and the session state object. New business logic code can, of course, be completely free-form, doing whatever it wants to within the context of how it is invoked, loading and calling outside Java or non-Java software modules, etc. However, it is more common to define new business logic in the existing Comergent eBusiness System architectural framework, utilizing new or existing business object definitions and interacting with back-end data stores through the Comergent eBusiness System Data Services module. In this way new business logic can reap the benefits of the Comergent eBusiness System architecture, including independence from specific databases or data source types.

Extending Existing Application Logic

It is also possible to extend existing business logic. This is typically done by defining a new Java subclass of an existing business logic class. The new subclass is again free to implement its mission however it sees fit, but it will commonly perform some activity that is complementary to or supervisory of the existing business logic, and invoke the base class to execute that business logic at the appropriate time. In this way it is possible to significantly extend or control existing Comergent eBusiness System business logic in a way that is external to the

definition of that code and is therefore maintainable and can be supported across upgrades of the underlying Comergent eBusiness System system.

System Configuration Files

Because the source code for the Comergent eBusiness System Platform is Comergent Technologies proprietary information and, as such, is not provided to the customer. The Comergent eBusiness System Architecture has instead been designed to use XML configuration files to specify values for all tunable parameters in the Comergent eBusiness System platform. The XML configuration files can be edited by hand or, more intuitively, modified through the Comergent eBusiness System Enterprise Administration interface.

Upgrading the Comergent eBusiness System

This chapter describes how to plan your approach to upgrading your Comergent eBusiness System from Release 6.7. The specific issues of upgrading from Release 6.7 to Release 7.0 are covered in "Upgrading From Release 6.7" on page 160. It addresses:

- "Access Control" on page 160

- "Database Schema" on page 163

- "System Properties" on page 163

- "Tag Libraries" on page 164

Overview of Upgradability

The Comergent eBusiness System has been designed from the ground up to meet the dual challenges of providing out-of-the-box application functionality against common business scenarios while providing the necessary flexibility to handle the extensions and customizations that occur in the normal course of deployment.

The software installation includes all the necessary source code, configuration files, data initialization scripts, and other tools necessary to perform the kinds of modifications described in this *Comergent eBusiness System Developer Guide*. This chapter describes upgrade considerations and process for each supported

customization technique as described in CHAPTER 15, "Tailoring the Comergent eBusiness System".

The Comergent eBusiness System System supports a progressive sequence of customization techniques designed to make the most common customizations the easiest to implement initially and to roll forward during an upgrade.

Customer Upgrade Scenarios

This section enumerates and explores the common reasons for upgrading the Comergent eBusiness System. The motivations for upgrade can directly impact the style and scope of the upgrade activity.

Upgrade Motivations

The following motivations are considered and occasionally referenced in the material which follows:

- Upgrading to obtain stability and performance benefits from the latest release.

- Upgrading to implement a new module which requires the latest Platform version.

- Upgrading to obtain across the board functionality and/or usability enhancements.

- Upgrading to obtain additional platform or standards support.

- Upgrading to implement a specific feature enhancement which involves inter-module communication and interaction.

- Upgrading to obtain inter-enterprise communication enhancements.

- Upgrading to obtain enhanced administration tools.

Upgrade Considerations by Customization Technique

The following categories of customization are supported by the Comergent eBusiness System architecture.

Upgrading Presentation

Presentation is typically the most extensively customized area. At minimum, the application of custom-branding is generally required during deployment to make the reference UI conform to the customer's Web UI style guidelines and standards.

Re-applying existing customizations to a new version of the Reference UI requires the fairly manual but mechanical process of comparing and merging individual JSP pages and Controller files. In that upgrade scenario, upgrade cost will be directly proportional to the extent of page reorganization and modification.

Alternatively, upgrade may focus on retaining the previously customized UI on the latest server-side APIs with minimal functional modifications to existing pages. This may be the case where upgrade is driven primarily by the desire to implement additional Comergent eBusiness System modules. In this case, upgrade focuses on backward compatibility of existing, customized JSP pages and Java-based controllers.

Upgrade Considerations for Customized JSP Files

The Model 2 JSP architecture is the foundation of presentation in the Comergent eBusiness System Releases 4.x and higher. This employs a Model/View/Controller, or MVC, design pattern with JSP as the Web page templating language and Java-based Controller classes as the orchestrators of page flow and request routing and processing. JSP pages have dependencies on specific Data and Logic Beans, for example, the Order Detail page depends on the OrderPresentationBean from which it acquires its data for display. The Order detail page itself controls specific layout and display of the order. In this case, the OrderPresentationBean provides both formatting specific logic and access to the underlying order business data.

The following are typical presentation customizations which must be re-applied or otherwise accounted for during upgrade.

- Application of custom branding in the form of images and text styles

- Addition of surrounding page content to the reference UI, for example, adding a site-wide navigation frame and branding header to Comergent eBusiness System Order Management

- Re-organization of page content

- Page flow modification

Page flow changes are accomplished by modifying or creating new Java Controllers, so typically some degree of Controller customization is performed along with JSP customization.

Administration pages are not intended to be branded or otherwise customized and therefore should not be affected during the upgrade process.

Upgrading From Release 6.7

This section describes the main issues that must be addressed in upgrading from Release 6.7 to Release 7.0, and how they should be tackled.

Access Control

Release 7.0 deprecates support for access control lists (ACLs) and replaces them with access policies. If you use ACLs in your implementation of the Comergent eBusiness System, then you must migrate them to be defined as access policies. See CHAPTER 7, "Using the Security Mechanisms" for descriptions of ACLs and access policies.

API Changes

The modularization of the Comergent eBusiness System platform has included the creation of new APIs and some reorganization of the existing platform packages and classes. As a result, some of your custom code may need to be changed to match the new APIs. This section describes the changes by listing the Release 6.7 packages and classes that have new Release 7.0 equivalents.

TABLE 6. API Changes by Package

Release 6.7 Packgage[a]	Release 6.7 Class	Release 7.0 Package	Release 7.0 Class
dcm.converter	*	converter	*
dcm.core	ComergentAppEnv.adjustFileName()	dcm.core	LegacyFileUtils.adjustFileName()
dcm.core	ComergentAppEnv.getEnv().getProperty()	preferences	Preferences
dcm.core	ComergentHelpBroker	help	ComergentHelpBroker
dcm.core	Global.logLevel methods	org.apache.log4j	Logger
dcm.core	Global.getProperty methods	preferences	Preferences
dcm.dataservices	*	api.dataservices	*
dcm.dataservices	IAcc*	api.dataservices	IData*
dcm.dataservices	IRd*	api.dataservices	IData*
dcm.dataservices	DsQuery	api.dataservices	IDsQuery
dcm.dataservices	DsUpdate	api.dataservices	IDsUpdate
dcm.dataservices	TransactionSupport	api.dataservices	ITransactionSupport

TABLE 6. API Changes by Package (Continued)

Release 6.7 Packgage[a]	Release 6.7 Class	Release 7.0 Package	Release 7.0 Class
dcm.eventbus	*	api.eventbus	*
dcm.objmgr	*	objmgr	*
dcm.util	OMWrapper	api.objmgr	OMWrapper
dcm.xml.dom	*	api.xml	*
dcm.xml.util	*	api.xml	*

a. All packagenames begin with com.comergent, and so this is omitted in package names here.

Logging

If you have logging calls that look like:

```
Global.logInfo("Your logging message");
```

You must change these to calls along these lines:

```
Logger log = org.apache.log4j.Logger.getLogger(OrdersAPI.class);
log.info("Your logging message");
```

Object Manager

If you have calls to:

```
com.comergent.dcm.util.OMWrapper.getObjectArg("com.comer-
gent.apps.salesContracts.bizAPI.SalesContractACLs", this);
```

You must change these to:

```
com.comergent.api.objmgr.OMWrapper.getObjectArg("com.comer-
gent.apps.salesContracts.bizAPI.SalesContractACLs", this);
```

There are also some changes to the calls that are available. When calling with
multiple arguments, you have to call by passing in the class. For example:

```
OMWrapper.getObjectArg("com.comergent.apps.salesCon-
tracts.bizAPI.SalesContractACLs", this);
```

has to be changed to:

```
OMWrapper.getObject(com.comergent.apps.salesContracts.bizAPI.Sales-
ContractACLs.class, this);
```

Calls to the *getObjectArg()* method in which you pass a number of parameters to
the constructor have been removed. You should now pass in in array of objects for
the parameters: OMWrapper.getObjectArg(String s, Object[] o).

Properties

If you have calls to retrieve property values such as:

```
String mergeLines =
    Global.getString("Quotes.mergeLineItemsInProductList");
```

You must change these to calls along these lines:

```
com.comergent.preferences.Preferences prefs =
    Preferences.getPreferences(MyClass.class);
String mergeLines =
    prefs.getString("Quotes.mergeLineItemsInProductList", "Never");
```

You must make the corresponding changes for calls to Global.getBoolean().

SDK Upgrade Tool

If you use the SDK to upgrade your Release 6.7 project to Release 7.0.1, then it will update most of the API usages in your project that have changed between the two releases. However, it does not update the following:

- BusinessObject: You should remove usages of this class.

- ComergentDocument: this class used to have a *addTextElement()* method which has been removed.

- ConverterFactory: The getNativeMessageFactory() methods have been moved to the com.comergent.dcm.messaging.MessagingHelper class.

- Global: the debugStream variable has been removed.

- IData: deprecated methods such as the following:

    - *copyBean()*

    - *generateKeys()*

    - *getUser()*

    - *getValueByName()*

    - *setUser()*

    - *updateWORestore()*

- IDataList: the *getRootElement()* has been removed.

- Logger: the two String form of the *info()* and *debug()* methods must be updated manually.

- ObjectManager.getObject(String s): if the argument is not provided as a quoted string.

- Transaction.init(): You should remove instances of this method from your project code.

- Classes that implement the com.comergent.api.appservices.productService.IProdServBeanRoot interface. You should modiy these so that they use the new method *restoreAndReturnBoolean()* to replace their uses of *restore()*.

- TableController: you should migrate controllers that extend this class to use the new CIC UI tags. If you must continue to use this class, then extract its source file from your Release 6.7 release, and copy it into your project.

- XMLParser: this class now returns instances of the org.w3c.dom.Document class and so code that expects instances of the ComergentDocument class will break. In cases where you need to use a deprecated method of the ComergentDocument class, you should change your code from:

```
doc = new XMLParser().parse(parserReader);
```

to:

```
doc = new ComergentDocument(XMLParser().parse(parserReader);
```

- XMLUtils: this class used to have a *formatTS()* method which has been removed. The method is now available using the com.comergent.api.appservices.productService.util.ProdMgrUtils class.

Database Schema

The database schema has changed in Release 7.0.1 with the addition of new tables, new columns to existing tables, and other new database objects such as sequences, indexes,and so on. You must migrate your current database schema to a Release 7.0.1 schema by following the instructions provided in the *Comergent eBusiness System Implementation Guide*.

System Properties

In earlier releases of the Comergent eBusiness System, the system properties that determined how the Comergent eBusiness System behaved were maintained in the **Comergent.xml** configuration file and other files that it referenced. Release 7.0 uses a Preferences-based system to manage properties and so you must migrate your configuration files to the new preferences files. See "Preferences Service" on page 44 for more information about Preferences.

Tag Libraries

Earlier releases than Release 7.0 provided JSP tags in two tag libraries: the Comergent tags and the CIC tags. These are documented in CHAPTER 30, "Comergent Tag Library" and CHAPTER 31, "Comergent Internet Commerce Tag Library" respectively. Release 7.0 has introduced changes to the CIC library which mean that earlier usages of the CIC tags may break.

Consequently, Release 7.0 provides a backward-compatible set of tags that behave identically to the older CIC tags. These are referenced as cic67 tags and are one-for-one equivalent to the Release 6.7 cic tags of the same name.

All the tag classes are supported in the **cmgt-taglib.jar** JAR file. They are declared in **cmgtinclude.jspf** by the following:

```
<%@ taglib uri="/cic67" prefix="cic67" %>
<%@ taglib uri="/cic" prefix="cic" %>
<%@ taglib uri="/cmgt" prefix="cmgt" %>
```

If you have used CIC tags in your custom JSP pages, then you have the following choices:

1. Either: change your use of the cic tags to cic67 in your pages. For example, change:

    ```
    <cic:column width="11%" ...>
    ```

    to:
    ```
    <cic67:column width="11%" ...>
    ```

2. Or: update the syntax of your cic tags to reflect the new syntax of Release 7.0 tags.

CHAPTER 17 *Customization Examples*

This chapter presents detailed descriptions of sample enhancements to the Comergent eBusiness System. They demonstrate the main steps required to modify the system.

The first examples demonstrate how the look-and-feel of pages can be changed. The last of these is a simple exercise to demonstrate how to re-use existing message types and controllers.

The subsequent examples lead you through more complex steps that show how controllers, data objects, business logic classes, data beans, and JSP pages are used:

• We show how a new data object and data bean are defined and how they are used to save data provided by a user.

• We show how to create a list data object and data bean and use them to display a table of data to users.

• We show how the access control mechanism is used to manage access to data. This example also shows how to create and use a system parameter as a business rule.

Setting up the SDK

Before starting work on the examples, you should set the SDK up on your system. You can use the SDK to ensure that as you modify your Comergent eBusiness

System, the changes that you make are all managed in one location. At minimum, you should follow the following steps:

1. Install the SDK on your system.

2. Run the install target to install your release of the Comergent eBusiness System into the SDK.

3. Run the newproject target to create an SDK project for your work. We will refer to the name of the project as *project* in the following examples.

4. Database targets:

   a. If you plan to run against an Oracle database server, then run the installOracle target. Then run the env.setDBType target to specify that the project uses an Oracle database.

   b. If you plan to run against a SQL Server database server set up as an ODBC data source, then run the installODBC target. Then run the env.setDBType target to specify that the project uses an ODBC database.

   c. If you plan to run against a DB2 database server, then run the installDB2 target. Then run the env.setDBType target to specify that the project uses a DB2 database.

5. Enter the database connection information in the ***project*_dev.properties** file. This ensures that the connection information is built into the data sources files and the database-specific data objects are used.

6. Run the merge target: this verifies that the basic build process can run in your environment.

7. Run the createDB and loadDB targets: these create the Knowledgebase schema and load the minimal data into it.

8. Run the dist target: this creates a new version of the **Comergent.war** file. You should be able to verify that you can deploy this WAR file into your servlet container and that it runs successfully.

If you run into difficulties during one of these steps, then you should troubleshoot the problem before continuing with these examples. See CHAPTER 14, "Software Development Kit" for more information.

Presentation

This section provides a number of examples of changing the look-and-feel of pages. Bear in mind that all of the JSP pages are organized by locale. All of the

instructions below are relative to a locale directory such as ***debs_home*/
Comergent/WEB-INF/web/en/US/**. If you want the same change to be effective in
more than one locale, then you must change the relevant files for each locale:
resources files of JSP and HTML pages.

Headers and Sidebars

As part of the process of branding the Comergent eBusiness System, you can
customize the customer-facing pages. Most of the reference customer-facing pages
use two frames:

- a header frame generated from **home/matrix_header.jsp**

- a data frame generated from data JSP pages such as **orderMgmt/Orders/
  OrderDisplayData.jsp**

You can customize the header frame to brand these pages simply by modifying the
matrix_header.jsp file or by substituting a different URL in the src attribute of the
frame element of the frameset page such as **fs_home.jsp** or **fs_directHome.jsp**.

Using the SDK

In this example, we will change several of the end-user facing pages. We will
replace the Matrix branding of the reference pages with branding for a different
company called Anderel. For example, to modify the **matrix_header.jsp** file,
perform the following steps:

1. Run the customize target as follows:

   ```
   sdk customize matrix_header.jsp
   ```

   This copies the **matrix_header.jsp** page under the **web/** sub-directory of
   your project folder.

2. Make your modifications to the JSP page file.

   a. For example, change the color of the banner by changing the line:

   ```
   <table bgcolor="#2f4f88" cellpadding="0" cellspacing="0"
       border="0"  height="45" width="100%">
   ```

   to:

   ```
   <table bgcolor="#ee9999" cellpadding="0" cellspacing="0"
       border="0"  height="45" width="100%">
   ```

   b. Change the logo by changing the line:

   ```
   <th><img src="../htdocs/manufacturer/images/matrix_logo.gif"
       border="0"></th>
   ```

   to:

```
<th><img src="../htdocs/manufacturer/images/anderel_logo.gif"
    border="0"></th>
```

Place a GIF image called **anderel_logo.gif** in the *sdk_home*/**projects/**
project/**en/US/htdocs/manufacturer/** directory.

3. Run the merge target to copy the modified file over to the **build/** directory:

```
sdk merge
```

Customer-facing pages in the reference implementation use two frames arranged
vertically using the rows attribute of the frameset element:

```
<FRAMESET rows="50,*" border="0" framespacing="0" frameborder="NO">
```

By using the cols attribute of the frameset element, you can move the navigation
bar to the left or right of the data frame. For example, customize the
fs_HomeLoggedIn.jsp page by changing the frameset as follows:

```
<FRAMESET cols="62,*" border="0" framespacing="0" frameborder="NO">
    <FRAME name="navigation"
        src="<%=link("*","MatrixHeaderDisplay") %>"
        marginwidth="0"  marginheight="0"  scrolling="no">
    <FRAME name="data"
        src="<%=link("*", "OrderDataDisplay") %>"
        marginwidth="0" marginheight="0" scrolling="Auto">
</FRAMESET>
```

Cascading Style Sheets

Use cascading style sheets wherever possible to manage the look-and-feel of Web
pages. They provide a mechanism to ensure that your customer-facing pages have a
uniform look-and-feel and changes to the page style need be made in only one place
rather than on every page.

If you want to make a change to the style of your customer-facing Web pages, then
you can make the change to the **css/ie_main.css** and **css/nn_main.css** cascading
style sheets. For example, changing the table.standardBackgroundColor,
tr.standardBackgroundColor, and td.standardBackgroundColor elements to
#f2de14 will change the table displays to a yellow from the current pale blue.

Cascading style sheets are maintained separately in each locale. Thus you can
maintain different look-and-feels for different locales, but correspondingly if you
want the same style change in every locale, then you must make the change in each
locale's cascading style sheets.

Using the SDK

For example, to modify the **ie_main.css** file, perform the following steps:

1. Run the customize target as follows:

```
sdk customize ie_main.css
```

This copies the **ie_main.css** page under the **en/US/css/** sub-directory of your project directory.

2. Make your modifications to the CSS file.

3. Run the merge target to copy the modified file over to the **builds/*project*/** directory:

```
sdk merge
```

Modifying Table Columns

In many parts of the customer-facing pages, the Comergent eBusiness System displays lists of business objects to users. For example, when a direct commerce customer clicks **My Lists**, the **orderMgmt/oil/workspace/ WorkspaceActiveOILData.jsp** page displays a list of their current active inquiry lists in the form of a table. It uses a JSP page fragment, **WorkspaceActiveListsDataTable.jspf**, to display the table.

Each row of such tables comprises table cells of the form:

```
<td class="dataTable" align="center"><%=ph(elementString)%></td>
```

where the elementString is calculated from an attribute of each data object.

You can change the order in which the columns of the table are displayed simply by editing the JSP page to change the relative position of the table cells in the row. You can remove a column simply by removing the corresponding cell from the row. You can add columns by adding the new table cell to the table row. Make sure that the data bean includes the information you want to display. If it does not, then you will need to modify the data object definition. (See "Extending and Modifying Existing Data Objects" on page 173.)

Note:	Note that whenever you make changes to the cells of a table, then you must modify the header rows too.

For example, suppose that you want to change the table of active product inquiry lists to include the status of each inquiry list. Make the following changes to the **orderMgmt/oil/workspace/WorkspaceActiveListsDataTable.jspf** page:

1. Run the customize target as follows:

```
sdk customize WorkspaceActiveListsDataTable.jspf
```

This copies the **WorkspaceActiveListsDataTable.jspf** page under the **WEB-INF/web/** sub-directory of your project folder.

2. Modify the scriptlet that generates the display strings for each inquiry list by adding the line:

```
String status = acart.getCartStatus();
```

3. Add two cells to the header of the table: one for the text, say "List Status" and one for the spacer element to separate it from the next column:

```
<th class="dataTable" align="center">
<cmgt:text id="*">List Status</cmgt:text>
</th>
<th width="-1%"><img src="../images/spacer.gif" width="1"
    height="1" border="0"></th>
```

Note the use of the text tag: this ensures that the table header text can be localized through the use of resource bundles. See CHAPTER 32, "Internationalization" for more information on localizing JSP pages.

4. In the corresponding location in the body row of the table, add two cells:

```
<td class="dataTable" align="center"><%=ph(status)%></td>
<td width="-1%"><img src="../images/spacer.gif" width="1"
    height="1" border="0"></td>
```

5. Run the merge target to copy the modified file over to the **build/** directory:

```
sdk merge
```

Notes

Note that JSP fragments (denoted by ***.jspf** files) are often used as included fragments of other JSP pages: for example:

```
<%@ include file="./WorkspaceActiveListsDataTable.jspf"%>
```

These fragments may not get automatically re-compiled when the Web application is re-deployed. Take care to check that the pages are re-compiled by the servlet container or force their re-compilation by deleting the compiled classes from the servlet container's working directory.

Note the use of the method *ph(status)* to wrap the displayed text: this is a protection from cross-site scripting attacks. It ensures that any HTML tags in the displayed string are displayed as text rather than interpreted as tags. See "Scriptlets" on page 233 for more information.

Adding a Shortcut Link

Description

In this example, we add a link to the Enterprise Home page that takes an enterprise user directly to their user profile. Do the following:

1. Create a link on the existing Enterprise Home page.

2. Specify a controller to process the request.

We make use of the fact that an appropriate controller already exists. This is not always possible and in the following examples we have to modify existing controllers and write a new Controller class. Similarly, we do not have to create a new JSP page: the UserDetail JSP page already exists: we ensure that the right user bean is passed to the page for display.

Enterprise Home Page

The current Enterprise Home page uses two subsidiary JSP pages to generate the two frame page. The navigation frame is generated using the **enterpriseMgr/ home/HomeNav.jsp** page and we shall leave this frame unchanged. The data frame is generated using the **enterpriseMgr/home/HomeData.jsp** page and we will add a link to this page as follows:

1. Run the customize target as follows:

   ```
   sdk customize HomeData.jsp
   ```

   This copies the **HomeData.jspp** page under the **WEB-INF/web/** sub-directory of your project folder.

2. Edit the **HomeData.jsp** by inserting the following:

   ```
   <%
   String userKey = user.toString();
   String groupKey = user.getGroupKey().toString();
   String parameterString = "UserKey=" + userKey + "&GroupKey=" +
       groupKey + "&Command=update";
   %>
   <TD>
   <A TARGET="_top"
       HREF='<%= link("enterpriseMgr","SysUserDetailDisplay",
           parameterString)%>'>
       <cmgt:text id="*">My User Profile</cmgt:text>
   </A>
   </TD>
   ```

Note that the User object present in the session is used to retrieve the user key and group key information so that they may be passed as parameters. The *link()* method is used to generate the URL in the correct form. See "Scriptlets" on page 233 for more information about the use of this method.

3. Run the merge target to copy the modified file over to the **build/** directory:

    ```
    sdk merge
    ```

When a user clicks this link, this is how the Comergent eBusiness System processes the request:

1. The request is received by the DispatchServlet. It creates a ComergentRequest object and instantiates a controller by calling *createController(comergentRequest)*.

2. The createController method retrieves the message type from the request and calls the GeneralObjectFactory to create a controller. In turn, the GeneralObjectFactory uses the **MessageTypes.xml** file to see what controller class should be created for the command (cmd) "SysUserDetailDisplay". The message type entry for this message type is:

    ```
    <MessageType Name="SysUserDetailDisplay">
        <JSPMapping>
            ../profileMgr/userContact/SysUserDetailFrame.jsp
        </JSPMapping>
        <ControllerMapping>
            com.comergent.dcm.caf.controller.ForwardController
        </ControllerMapping>
    </MessageType>
    ```

    Consequently, the GeneralObjectFactory creates an instance of the ForwardController class.

3. Next, the DispatchServlet initializes the controller and then calls its *execute()* method.

    The ForwardController is a very lightweight controller. Its *execute()* method simply calls *callJSP()*. This method retrieves the message type from the request and calls *callJSP(String messageType)*. In turn, this method retrieves the appropriate JSP page from the message type:

    ```
    ../profileMgr/userContact/SysUserDetailFrame.jsp
    ```

    Then it creates a Dispatcher using this JSP page and forwards the request.

4. The JSP page, **SysUserDetailFrame.jsp**, is made up of several frames. The source for each is generated from a dynamically generated URL. For example the URL to generate the data frame is generated from:

```
<%= link("partnerMkt", "SysUserDetailData") %>
```

Each of the frame requests is processed along the same lines: the cmd parameter is evaluated and the appropriate business logic class (BLC), controller, and JSP page are used to generate its content.

In this example, the SysUserDetailData message type, has a mapping to the com.comergent.apps.profileMgr.userMgr.controller.-UserContactGetController controller. This controller is invoked in order to populate the request with the GroupBean, PartnerBean, and UserContactBean used in the JSP page to which SysUserDetailData maps: **../profileMgr/userContact/UnifiedSysUserInfoData.jsp**.

Extending and Modifying Existing Data Objects

In many implementations of the Comergent eBusiness System, you have to make changes to the functionality of the system. In particular, you may find that you must either extend the functionality of an existing data object or modify its structure. You may need to add or modify an attribute, or store different information with the data object.

For example, an implementation may require that products have an additional attribute such as the associated product manager or that users provide not only their username and password when they log in, but also another piece of data such as the name of their company.

Note that there are essentially two different ways in which you can modify a data object.

- For business objects whose corresponding data object's Version attribute is 5.0 or higher, you can also define a new data object that *extends* the current data object. See the *Comergent eBusiness System Reference Guide* for more details on extending data objects.

- For business objects whose data object's Version attribute is 4.1 or lower, you can only modify them by modifying the data object element that corresponds to the business object.

In general, you should extend existing data objects rather than modify them: this will make easier the process of upgrading the Comergent eBusiness System from one release to another.

Using the custom Schema Directory

With the release of Release 6.3 and SDK Version 2.0, it is possible to manage your customizations to the schema using a special sub-directory of the schema directory as follows:

1. In your project templates directory make sure that the **DataServices.xml** file declares the schemaRepositoryExtn element:

```
<schemaRepositoryExtn controlType="text" runtimeDisplayed="true"
ChangeOnlyAtBootTime="true" visible="true" boxsize="45"
displayQuestion="Schema Repository Directory Location"
defaultChoice="WEB-INF/schema" help="Enter the path of the XML
schema Repository.">WEB-INF/schema/custom</schemaRepositoryExtn>
```

2. Check that your project has a directory called: *sdk_home*/**projects**/*project*/ **WEB-INF/schema/custom/**. It should contain empty copies of the standard **DsBusinessObjects.xml**, **DsDataElements.xml**, and **DsRecipes.xml** files.

3. As you create new data objects or extend existing ones, make sure that the new data object XML files are placed in the **custom/** sub-directory. Update the **DsBusinessObjects.xml**, **DsDataElements.xml**, and **DsRecipes.xml** files in this directory to add new business objects, data objects, and data elements as you create them.

4. When you build your project by running the merge target, the new data objects are merged in from this sub-directory and beans are generated for each of the new data objects.

Extending a Data Object

In this example, we show how a data object of the reference implementation is extended to add functionality. We will add a field to the Product data object that can be used to associate a user with the product. For example, this field might be used to associate a product manager with each product.

1. Locate the main schema files under the project directory: they are in the *sdk_home*/**projects**/*project*/**WEB-INF/schema/** sub-directory.

2. Create a new business object declaration in the **DsBusinessObjects.xml** file:

```
<BusinessObject Name="MatrixProduct" Version="5.0"
   Description="Matrix product business object"/>
```

3. Create a new recipe in the **DsRecipes.xml** file:

```
<Recipe Name="MatrixProduct" Version="6.0"
   BusinessObject="MatrixProduct"
   Description="Product object for Matrix implementation">
```

```
        <DataObjectList>
            <DataObject Name="MatrixProduct" Access="RWID"
                DataSourceName="ENTERPRISE"/>
        </DataObjectList>
    </Recipe>
```

4. Create new data elements for MatrixProduct and MatrixProductList in the
    DsDataElements.xml file:

```
<DataElement Name="MatrixProduct"
    Description="Product for Matrix implmentation"
    DataType="HEADER"/>
<DataElement Name="MatrixProductList"
    Description="Product list for Matrix implmentation"
    DataType="LIST"/>
```

5. Define the new MatrixProduct data object. Create a new file,
    MatrixProduct.xml in the schema directory. The file specifies the data object
    as follows:

```
<?xml version="1.0"?>
<DataObject Name="MatrixProduct" Extends="Product"
    ExternalName="CMGT_PRODUCT"
    Access="RWID" Ordinality="n"
    ObjectType="JDBC" Localized="y"
    Version="6.0">
    <DataField Name="EnterpriseUser"
        Description="Enterprise user acting as product manager
            for product"
        Writable="y" Mandatory="n"
        ExternalFieldName="ENTERPRISE_USER" />
</DataObject>
```

    Notice that that this data object will use the underlying table
    CMGT_PRODUCT. It extends the Product data object by adding one data
    field.

6. Create a new data element for EnterpriseUser in the **DsDataElements.xml**
    file:

```
<DataElement Name="EnterpriseUser" Description="Enterprise user"
    DataType="LONG" MaxLength="20"/>
```

7. Add a column to the CMGT_PRODUCT table:

```
ALTER TABLE CMGT_PRODUCT ADD (ENTERPRISE_USER NUMBER(20))
```

    Note: this is is the Oracle syntax: it is different for other database servers. If
    you have existing users in the CMGT_PRODUCT table, then you may
    manually update their rows to add a non-null value to this column. If you

are altering the table before populating the Knowledgebase, then you must modify the **ProductList** XML file by adding the EnterpriseUser element to each Product element before you load the data.

8. Regenerate the DTDs and regenerate the data bean classes and interfaces. You can use the SDK targets provided as part of the Software Development Kit. From the command line, run the SDK script specifying the merge target to generate automatically the new Bean classes as follows.

```
sdk merge
```

This copies over the new schema files to the builds directory and generates the corresponding databean classes and interfaces. See CHAPTER 14, "Software Development Kit" for more details.

9. Modify the **ObjectMap.xml** file so that whenever a product is instantiated using the ObjectManager, then the MatrixProductBean class is used:

```
<Object ID="com.comergent.bean.simple.ProductBean">
    <ClassName>com.comergent.bean.simple.MatrixProductBean
</ClassName>
```

10. Run the merge target to copy the modified files over to the *sdk_home*/**builds/ project/** directory:

```
sdk merge
```

If you now deploy the modified Comergent eBusiness System, then the system should run as before. No business logic has been changed, but every time a product is accessed the actual class instantiated is now the MatrixProductBean.

Note:	In this example, note that the product export and import functionality needs to be customized in order to include the new field. You must modify the ImportHandler and ExportHandler classes to do this.

To use the extended data object, you must determine how a product manager enterprise user can be associated with a product using the Comergent eBusiness System administrative interface and how this information might be used by applications. For example, when an enterprise administrator creates a new product, you can provide a drop-down list of enterprise users so that one can be selected as the responsible product manager for the product.

•   When you are creating or modifying an instance of the extended data object, then you must be able to set a value for this new field. Typically, the object is wrapped in an application bean (see "Application, Entity, and Presentation Beans" on page 68). To set the value, you must modify the controller managing the class so that the field can be accessed along these

lines. For example, in the ProdMgrProdGenController class, the application bean is an instance of an object implementing the IBizProduct interface. This interface provides access to the data bean through its *getIaccProduct()* method. The IBizProduct object is refered to as pb, and so you can add:

```
String enterpriseUser = request.getParameter("EnterpriseUser");
MatrixProductBean mpb =
    (MatrixProductBean) pb.getIAccProduct();
mpb.setEnterpriseUser(enterpriseUser);
```

- On pages that you want to display the new field, you may find that a presentation bean is being used to provide the data. For example, on the **ProdMgrProdGen.jsp** page, product detail data is displayed using an object that implements the IPresProduct interface. This object is referred to as pb and is retrieved fromthe request passed to the JSP page. You can display the new field with code along these lines:

```
ph(((MatrixProductBean) pb.getIRdProduct()).getEnterpriseUser());
```

Similarly, you can customize business logic and JSP pages so that as end users browse products, they have the opportunity to contact the relevant product manager to request more information. For example, you could add a button next to each product in the product catalog, that would initiate an email enquiry directly to the product manager.

Modifying a Data Object

In this example, we add a field to the User data object to ensure that each user identifies the partner organization to which they belong when they log in. This is a mandatory field that must be entered when the user is created and it must be supplied by the user when they enter their login details.

Note that from the viewpoint of upgrading this implementation of the Comergent eBusiness System, these changes will cause problems in that any changes to the User and UserContact data object made in the next release of the Comergent eBusiness System will have to be manually introduced into this implementation.

1. Locate the main schema files under the project directory: they are in the **sdk_home/projects/project/WEB-INF/schema/** sub-directory.

2. Add a DataField element to the UserContact data object:

```
<DataField Name="Organization" Writable="y" Mandatory="y"
    ExternalFieldName="ORGANIZATION"/>
```

3. Add a DataField element to the User data object:

```
<DataField Name="Organization" Writable="y" Mandatory="y"
    ExternalFieldName="ORGANIZATION"/>
```

4. Add a DataElement for Organization to the **DsDataElements.xml** file:

```
<DataElement Name="Organization" Description="Organization name"
    DataType="STRING" MaxLength="16" />
```

5. Add a column to the CMGT_USER_CONTACTS table:

```
ALTER TABLE CMGT_USER_CONTACTS ADD (ORGANIZATION VARCHAR2(16))
```

Note: this is is the Oracle syntax: it is different for other database servers. If you have existing users in the CMGT_USER_CONTACTS table, then you must manually update their rows to add a non-null value to this column. If you are altering the table before populating the Knowledgebase, then you must modify the **UserContact** XML file by adding the Organization element to each UserContact element before you load the data.

6. Regenerate the DTDs using the merge target.

7. Modify the login pages so that there is an additional textfield in which users can enter their organization when they login:

```
<INPUT TYPE="text" NAME="organization" SIZE="16" MAXLENGTH="16">
```

For example, the enterprise use login page is **enterpriseMgr/Login/ FullPageLogin.jsp**.

8. Modify the user detail pages used to display user details and used when creating new users. Add an extra mandatory textfield as follows so that is displayed when viewing user details. The field label ("Organization") is wrapped in the text tag for localization.

```
<tr valign=top>
    <td class="Attribute"><cmgt:text id="*">
        Organization
    </cmgt:text></td>
    <td class="Attribute">
    <%if (isWritable){ %>
    <INPUT NAME="Organization" TYPE="Text"
        VALUE="<%=ph(replaceNullWithEmpty(userContact.-
            getOrganization()))%>"
        SIZE="16" MAXLENGTH="16">
    <%}%>
    </td>
</tr>
```

Optionally, you can add to the *checkInput()* Javascript function to verify that an organization has been provided when a user is being created:

```
if (f.Organization.value == "")
```

```
{
    alert("<cmgt:text id="*">Please provide organization for this
user.</cmgt:text>")
    return false;
}
```

9. Modify the login authentication logic to add the organization information
    when the user is authenticated. This requires changes to classes as follows:

    a. Modify the com.comergent.dcm.authentication.User class to add methods
        setOrganization() and *getOrganization()* that manage the Organization
        element of the user business object. Create a new form of the *getInstance()*
        method which takes three String parameters:

        ```
        static public User getInstance(String user, String password,
        String organization) throws ICCException,
        InvalidBizobjException
        {
            DataManager dm = DataManager.getDataManager();
            BusinessObject userBizobj = dm.getBusinessObject("User");
            DsElement root = userBizobj.expand();
            root.expand(userBizobj);
            DsElement el = root.getByName("UserLogin");
            el.setStringValue(user);
            el = root.getByName("UserAuthenticator");
            el.setStringValue(password);
            el = root.getByName("Organization");
            el.setStringValue(organization);
            return new User(userBizobj);
        }
        ```

    b. Extend the com.comergent.dcm.authentication.UserPasswordCredentials
        class by creating a MatrixCredentials class that has an additional member
        variable, m_organization, and a constructor MatrixCredentials(String
        username, String password, String organization). Its *verify()* method uses
        the *User.getInstance(String username, String password, String
        organization)* method to restore the user business object.

    c. Extend the com.comergent.dcm.authentication.LoginController class by
        creating a MatrixLoginController class. In this class, overwrite the
        getCredentials() method by adding code to retrieve the organization
        parameter from the request object and return a MatrixCredentials object.

10. In the **MessageTypes.xml** files, replace entries of the form:

    ```
    <ControllerMapping>LoginController</ControllerMapping>
    ```
    with:
    ```
    <ControllerMapping>MatrixLoginController</ControllerMapping>
    ```

11. Run the merge target to copy the modified files over to the ***sdk_home*/builds/ *project*/** directory:

```
sdk merge
```

Adding Functionality to an Application

In this next example, we show you how to add functionality to an application. We will enable users to add comments about products by customizing the Product Detail page and creating a new page for adding a comment and reading the comments of other users. This functionality could be used to help customers read product reviews written by other customers.

This customization will include

- creating a new data object called Comment

- generating the DataBean classes

- modifying the database schema

- modifying the **ObjectMap.xml** file

- modifying a JSP page and creating a new one

- writing new Controller classes

- modifying the **MessageTypes.xml** file

Comment Data Object

Thi section describes changes to the data services schema. Follow the guidelines in "Schema Files" on page 143 to ensure that you manage these changes in the recommended way.

We create a new data object called Comment:

1. Add a new business object declaration to the **DsBusinessObjects.xml** file:

```
<BusinessObject Name="Comment" Version="6.0"
    Description="Comment on product by user"/>
```

2. Add a new recipe to the **DsRecipes.xml** file:

```
<Recipe Name="Comment" Version="6.0" BusinessObject="Comment"
    Description="Comment Recipe">
    <DataObjectList>
        <DataObject Name="Comment"  Access="RWID"
         Ordinality="n" DataSourceName="ENTERPRISE"/>
    </DataObjectList>
```

```
</Recipe>
```

3. Create a new DataObject definition file called **Comment.xml**:

```xml
<?xml version="1.0"?>
<DataObject Name="Comment" Extends="C3PrimaryRW"
    ExternalName="MTRX_COMMENT"
    Access="RWID"
    ObjectType="JDBC"
    Version="6.0">
    <KeyFields>
        <KeyField Name="CommentKey" ExternalName="COMMENT_KEY"
            KeyGenerator="CommentKey"/>
    </KeyFields>
    <DataFieldList>
        <DataField Name="CommentKey"
            Writable="y" Mandatory="y"
            ExternalFieldName="COMMENT_KEY"/>
        <DataField Name="ProductID"
            Writable="y" Mandatory="y"
            ExternalFieldName="SKU_NAME"/>
        <DataField Name="Description"
            Writable="y" Mandatory="n"
            ExternalFieldName="DESCRIPTION"/>
    </DataFieldList>
</DataObject>
```

Note that:

•   This data object extends an existing data object, C3PrimaryRW. The
    C3PrimaryRW data object is a standard one provided by the Comergent
    eBusiness System. Do not modify this data object. All data objects that use
    the ACL mechanism to protect access to them must extend the
    C3PrimaryRW data object.

•   In general, use a project-specific table name for any database tables: this
    ensures that there is no likelihood of an upgrade overwriting the table and
    its data.

•   Its ordinality is set to "n" in its recipe: this means that a CommentListBean
    will also be created when the generateBean target is run. This list bean is
    used to hold a list of comments.

4. Create the following new DsDataElements in the **DsDataElements.xml** file:

```xml
<DataElement Name="Comment" Description="Product comment"
    DataType="HEADER"/>
<DataElement Name="CommentList" Description="Product comment list"
    DataType="LIST"/>
<DataElement Name="CommentKey" Description="Comment Key"
```

```
DataType="LONG" MaxLength="20"/>
```

Note that you do not have to add DataElements for ProductId and Description DataFields: you can re-use the DataFields already defined.

5.   Add a new key generator to the appropriate **DsKeyGenerators.xml** file (for example, **OracleKeyGenerators.xml**):

```
<KeyGenerator Name="CommentKey"
    KeyProcedureName="COMMENTKEY"
    GeneratorType="PROCEDURE" />
```

Generating the Comment and CommentList Data Beans

Rather than produce bean classes manually for the Comment and CommentList data beans, we can use the generateBean target to generate them for us:

1.   Run the generateBean target. If it runs correctly, then it will display a series of messages that it has successfully generated and compiled Bean classes for each data object. The compiled classes are in *debs_home*/**Comergent/WEB-INF/ classes/com/comergent/bean/simple/**.

2.   Check that the Java classes called CommentBean and CommentListBean are created and compiled.

Database Schema Modification

Modify the database schema definition as follows:

1.   Create a new table using the following SQL statements:

```
prompt 'About to create table MTRX_COMMENT'

drop table MTRX_COMMENT;

create table MTRX_COMMENT(
COMMENT_KEY number(20) NOT NULL,
SKU_NAME varchar2(120) NOT NULL,
DESCRIPTION varchar2(240),
UPDATE_DATE date,
UPDATED_BY number(20) NOT NULL,
CREATION_DATE date default sysdate,
CREATED_BY number(20) NOT NULL,
OWNED_BY number(20) NOT NULL,
ACCESS_KEY number(20),
ACTIVE_FLAG varchar2(1) default 'Y',
PRIMARY KEY (COMMENT_KEY));
```

2.   Create a new sequence for the business object:

```
drop sequence comment_key_seq;
```

```
create sequence comment_key_seq;
```

3. Create a new procedure for the sequence:

```
create or replace procedure commentkey(p_key out number)
as
begin
    select comment_key_seq.nextval
    into p_key
    from dual;
end commentkey;
/
```

Updating the ObjectMap.xml File

To ensure that the correct object is instantiated whenever you want to use a comment or comment list, then add the following elements to the **ObjectMap.xml** file:

```
<Object ID="com.comergent.bean.simple.IRdComment">
    <ClassName>com.comergent.bean.simple.CommentBean</ClassName>
</Object>
<Object ID="com.comergent.bean.simple.IAccComment">
    <ClassName>com.comergent.bean.simple.CommentBean</ClassName>
</Object>
<Object ID="com.comergent.bean.simple.IDataComment">
    <ClassName>com.comergent.bean.simple.CommentBean</ClassName>
</Object>
<Object ID="com.comergent.bean.simple.IRdCommentList">
    <ClassName>com.comergent.bean.simple.CommentListBean</ClassName>
</Object>
<Object ID="com.comergent.bean.simple.IAccCommentList">
    <ClassName>com.comergent.bean.simple.CommentListBean</ClassName>
</Object>
<Object ID="com.comergent.bean.simple.IDataCommentList">
    <ClassName>com.comergent.bean.simple.CommentListBean</ClassName>
</Object>
```

JSP Pages

Customizing the Product Detail JSP Page

The JSP page used to generate the HTML of the Product Detail page is the file *debs_home*/**Comergent/WEB-INF/web/en/US/catalog/ CatalogProductDetail.jsp**. In this example, we add a button that opens up a new window to display the comments for the product. We add it to the product detail table as a table cell as follows:

```
<td class="dataTable">
    <a href="JavaScript:getComments('<%= pj(pb.getProductID()) %>')">
```

```
    <img src="../images/btn_productComments.gif" border="0">
    </a>
</td>
```

The Javascript function *getComments()* is:

```
function getComments(productID)
{
    count=0;
    window.open("", "ChildDetailWindow_"+count, "directories=no,
        toolbar=no,menubar=no,scrollbars=yes,
        resizable=no,height=540,width=720");
    document.dataForm.action="<%= pu(link("catalog",
        "catProductComments")) %>";
    document.dataForm.target="ChildDetailWindow_"+count;
    document.dataForm.productID.value=productID;
    document.dataForm.submit();
    count++;
    return;
}
```

Creating the Product Comments Page

The page in the new window is generated by creating a new page *debs_home*/
Comergent/WEB-INF/web/en/US/catalog/
CatalogProductCommentsPopup.jsp.

To begin with, it just contains a form with a text area element into which users can
enter comments about the product:

```
<%
    IPresProduct pb = (IPresProduct) request.getAttribute("product");
%>

<p>Enter a comment about this product: <%= pj(pb.getName()) %></p>
<form method="POST" action="<%= link("adirect", "addComment",
    "productID=" + pb.getProductID()) %>">
    <textarea name="commentDescription" rows="4" cols="80">
    <cmgt:text id="*">Enter your comments here.</cmgt:text>
    </textarea>
    <input type="SUBMIT" name="COMMENT"
        value="<cmgt:text id="*">ADD COMMENT</cmgt:text>">
</form>
```

Note that the command parameter used is "addComment" and that we use the
product bean to retrieve the product key for this product. This object is passed in
through the request object. Note also that the product bean class is not
com.comergent.bean.simple.ProductBean, but rather
com.comergent.api.appservices.productService.IPresProduct. This is a common

way of wrapping a standard generated bean in a class that provides methods useful in presenting the bean in JSP pages. The Javadoc for this class is provided as part of the SDK.

Managing the Business Logic

We will need two controllers to manage this new functionality:

- a controller to process the catProductComments command executed to open the product comments window;

- a controller to process the addComment command executed to add a new comment.

There are two ways in which you can create the business logic to process each request:

- You can create a business logic class (BLC) and perform the necessary logic there. This approach is now deprecated.

- You can create a custom controller.

We recommend creating a custom controller in conjunction with a bizAPI class to manage any business or presentation logic that may be required. By convention, in Release 6.0 and higher, controllers are regarded as part of the reference package organization. In this example, we create the controller classes in the com.comergent.reference.apps.catalog.controller package: you must manually create the **com/comergent/reference/apps/catalog/controller/** hierarchy of directories under the **src/** directory of the project directory.

Try to structure the *execute()* method of the controllers so that the phases of the method are clear:

1. Extract parameters from request and perform any server-side validation.

2. Perform processing on the data objects.

3. Prepare data objects and presentation wrappers and marshall them to pass to the JSP page using the request and session objects.

4. Forward to JSP page.

Creating the CatalogProductCommentsController Class

Initially, simply to display the product comments page, we just pass the product to the JSP page. The CatalogProductCommentsController extends the ForwardController by overwriting its *execute()* method as follows:

```
public void execute() throws ControllerException, ICCException,
```

```
    IOException
{
    String productID = request.getParameter("productID");
    // Restore the product and attach it to the request
    com.comergent.appservices.productService.BizProductBean msBean =
    (com.comergent.appservices.productService.BizProductBean)
    com.comergent.dcm.util.OMWrapper.getObject(
    com.comergent.appservices.productService.BizProductBean.class);
    msBean.setProductID(productID);
    msBean.restore();
    msBean.restoreFeatures(true);
    msBan.restoreAssemblyItems();
    msBean.computePrice();
    request.setAttribute("product", msBean);
    callJSP();
}
```

Creating the CreateCommentController Class

To add a comment about a product, a user enters text into the text area of the form, and then clicks the ADD COMMENT button. We need to add business logic to the application so that this request is processed by the Comergent eBusiness System. We create a custom Controller class called CreateCommentController to process the request. This class extends the ForwardController class by overwriting its *execute()* method as follows:

```
public void execute() throws ControllerException, ICCException,
    IOException
{
    //Retrieve request parameter
    String productID = request.getParameter("productID");
    // create the new comment data bean
    CommentBean newComment = getNewComment(productID);
    // set the top level fields
    setCommentFirstLevelFields(newComment);
    // save the new comment
    DataContext temp_DataContext = new DataContext();
    newComment.persist(temp_DataContext);
    // Restore the product and attach it to the request
    com.comergent.appservices.productService.BizProductBean msBean =
    (com.comergent.appservices.productService.BizProductBean)
    com.comergent.dcm.util.OMWrapper.getObject(
    com.comergent.appservices.productService.BizProductBean.class);
    msBean.setProductID(productID);
    msBean.restore()
    msBean.restoreFeatures(true);
    msBean.restoreAssemblyItems();
    msBean.computePrice();
    request.setAttribute("product", msBean);
```

```
    callJSP();
}

protected CommentBean getNewComment(String s) throws ICCException
{
    String desc = request.getParameter("commentDescription");
    //Create comment bean
    CommentBean comment = (CommentBean)
        OMWrapper.getObject(
        com.comergent.bean.simple.CommentBean.class);
    comment.setProductID(s);
    comment.setDescription(desc);
    return comment;
}

protected void setCommentFirstLevelFields(CommentBean comment)
    throws ICCException
{
    ComergentSession session = ComergentAppEnv.getCurrentSession();
    User myUser = session.getUser();
    Long userKey = myUser.getUserKey();
    comment.setUpdatedBy(userKey);
    comment.setCreatedBy(userKey);
    comment.setOwnedBy(userKey);
}
```

Note the following:

- The new comment is created and persisted in the controller. Some applications manage all these activities in bizAPI classes.

- The controller must ensure that the product bean is restored and added to the request object. The **CatalogProductCommentsPopup.jsp** page is expecting to retrieve the product bean from the request to retrieve the product ID.

Using the ForwardController Class

Note that in this example, the CreateCommentController class extends the ForwardController class. Consequently, you must specify a JSP page in this MessageType element. If you subsequently want to specify a different JSP page, then you have only to change the **MessageTypes.xml** file: you do not have to modify and re-compile the CreateCommentController Java class. In addition, you do not have to manage the use of multiple JSP pages in the Controller source to manage different locales.

Updating the MessageTypes.xml File

We add these two message types to the **MessageTypes.xml** file as follows:

```
<MessageType Name="catProductComments">
    <JSPMapping>
        ../catalog/CatalogProductCommentsPopup.jsp
    </JSPMapping>
    <ControllerMapping>
        com.comergent.reference.apps.catalog.controller.-
            CatalogProductCommentsController
    </ControllerMapping>
</MessageType>
<MessageType Name="addComment">
    <JSPMapping>
        ../catalog/CatalogProductCommentsPopup.jsp
    </JSPMapping>
    <ControllerMapping>
        com.comergent.reference.apps.catalog.controller.-
            CreateCommentController
    </ControllerMapping>
</MessageType>
```

Add message type references to the appropriate message group (say, CatalogGroup) as follows:

```
<MessageTypeRef Name="catProductComments" />
<MessageTypeRef Name="addComment" />
```

Note that the addComment message type returns the user to the product comments page. The user must manually close this window when they have finished with it.

Modifying the Controller Classes

In addition to adding comments, users will want to read what other users have to say about a product. We need to display the comments using the same JSP page, **CatalogProductCommentsPopup.jsp**. To do this, we want to pass a bean to the page that contains a list of all of the comments made by users that relate to the product. We will use a list data object to do this. We use the list data bean automatically generated from the Comment data object definition (its Ordinality attribute is set to "n" in its Recipe element), and customize the **CatalogProductCommentsPopup.jsp** page to iterate through the list bean.

In general, you should bear in mind that the result of restoring a list data bean can be a list with many data beans. To display the resulting list on a browser page may give rise to unacceptable usability and performance problems. Consequently, you should consider using the pagination functionality supported by the Comergent eBusiness System. See "Pagination" on page 194.

We need to add code to the *execute()* method of the Controller classes CatalogProductCommentsController and CreateCommentController because they forward requests to the **CatalogProductCommentsPopup.jsp** page.

We are going to make use of a bizAPI interface ICommentList to retrieve the list of comments for the specified product. Note the use of the OMWrapper class to instantiate an instance of the class that implements the interface. The corresponding entry in the **ObjectMap.xml** file is:

```
<Object ID="com.comergent.api.apps.catalog.ICommentList">
    <ClassName>
    com.comergent.apps.catalog.bizAPI.CommentList
    </ClassName>
</Object>
```

The code for the ICommentList interface is:

```
package com.comergent.api.apps.catalog;

import com.comergent.bean.simple.IRdCommentList;

public interface ICommentList
{
    public abstract void setSKU(String sku);
    public abstract String getSKU();
    public abstract IRdCommentList getDataBean();
    public abstract IRdCommentList getCommentList(String sku);
}
```

The code for the CommentList class that implements the interface is:

```
package com.comergent.apps.catalog.bizAPI;

import com.comergent.api.apps.catalog.ICommentList;
import com.comergent.bean.simple.*;
import com.comergent.dcm.core.Global;
import com.comergent.api.dataservices.*;
import com.comergent.api.objmgr.OMWrapper;

public class CommentList implements ICommentList
{
    protected IDataCommentList clb;
    protected String sku;

    public CommentList()
    {
        clb = null;
    }

    public IRdCommentList getCommentList(String sku)
```

```
{
    try
    {
    this.sku = sku;
    clb = (com.comergent.bean.simple.IDataCommentList)
    OMWrapper.getObject(
        com.comergent.bean.simple.IDataCommentList.class);
    DsQuery temp_DsQuery =
        QueryHelper.newWhereClause("ProductID",
            DsQueryOperators.EQUALS_IGNORE_CASE, sku);
    clb.restore(new DataContext(), temp_DsQuery);
    }
    catch(Exception e)
    {
        Global.logInfo(e.toString());
    }
    return (IRdCommentList) clb;
}

public String getSKU()
{
    return sku;
}

public void setSKU(String sku)
{
    this.sku = sku;
}

public IRdCommentList getDataBean()
{
    return (IRdCommentList) clb;
}
}
```

Note the use of the DsQuery and QueryHelper classes to ensure that the *restore()* operation retrieves only those comments whose ProductID field is the product we are interested in.

Add the following lines to the *execute()* method of both **CatalogProductCommentsController.java** and **CreateCommentController.java** immediately before the *callJSP()* call:

```
ICommentList clBean = (ICommentList)
OMWrapper.getObject(
com.comergent.api.apps.catalog.ICommentList.class);
IRdCommentList readCLBean = clBean.getCommentList(productID);
request.setAttribute("comments", readCLBean);
```

Modifying the JSP Page

We must add a table to the JSP page that displays the comments, **CatalogProductCommentsPopup.jsp**. To do this we use a scriptlet as follows:

1. Add the following to the header of the page:

```
<%
    com.comergent.bean.simple.IRdCommentList commentListBean =
        (com.comergent.bean.simple.IRdCommentList)
            request.getAttribute("comments");
%>
```

2. Add the following to the body of the page:

```
<%
    ListIterator iter = commentListBean.getCommentIterator();
    request.setAttribute("commentList", iter);
%>
<cic:table datasourceRef="${commentList}" var="comment"
    showSelect="false" sortAscending="true" labelrowcss="label"
    rowcss="normal,alternate" >
    <cic:column width="10%" css="left">
        <cic:columnHeader><cic:span value="ID"/></cic:columnHeader>
            <cic:span value="${comment.commentKey}"/>
    </cic:column>
    <cic:column width="10%" css="left">
        <cic:columnHeader><cic:span value="CreatedBy"/>
        </cic:columnHeader>
        <cic:span value="${comment.createdBy}"/>
    </cic:column>
    <cic:column width="80%" css="left">
        <cic:columnHeader><cic:span value="Comment"/>
        </cic:columnHeader>
        <cic:span value="${comment.description}"/>
    </cic:column>
</cic:table>
```

The enumeration loop iterates through the commentListBean Iterator.

Customizing Access to the Business Objects

Our last example extends the previous one by demonstrating the use of the Comergent eBusiness System security mechanisms to manage access to business objects. By default, when any Version 5.0 data object (that is, whose Version attribute is set to "5.0" or higher) is persisted or restored, a security check is performed to verify that the current user is authorized to perform the action. You can use one of these approaches to manage access:

- "Access Policy Approach" on page 192

- "ACL Approach" on page 193

Release 6.4.1 and earlier releases have used the ACL mechanism whereas Release 6.7 and later releases primarily use the access policy mechanism. In general, you should use access policies for all your customization work.

Access Policy Approach

You can use an access policy to manage access to any resource in the Comergent eBusiness System. This section describes how to create an access policy so that only users who belong to the same partner as the user who made the comment can view the comment. See "Managing Access to Data Objects Using Access Policies" on page 93 for more information on access policies.

You want to define an access policy that expresses the requirement that users can see only comments made by users who belong to the same partner as them. In this case, the resource being managed by the access policy is the CommentBean, so begin by declaring the access policy as follows:

```
<AccessPolicy Name="CommentPolicy" PrincipalQualifier="UserRole">
    <Description>
        This policy determines that comments can be viewed only
        by users who belong to the partner as the creater of the
        comment.
    </Description>
    <ResourceClass>
        com.comergent.bean.simple.CommentBean
    </ResourceClass>
    <AccessChecker>
        <Principal>Partner.User</Principal>
        <ActionType>Restore</ActionType>
        <BooleanExpression>
            <ComparativeExpression Operator="Equals">
                <Term>service.usersRootPartnerKey</Term>
                <Term>service.ownersRootPartnerKey</Term>
            </ComparativeExpression>
        </BooleanExpression>
    </AccessChecker>
    <AccessChecker>
        <Principal>Partner.User</Principal>
        <ActionType>Create</ActionType>
        <BooleanExpression>
            <Always/>
        </BooleanExpression>
    </AccessChecker>
</AccessPolicy>
```

Note that each access policy must have a unique name. You use the PrincipalQualifier attribute to define what property is going to be compared in the access check. In this case UserRole will qualify the principal by check that at least one of the user's roles matches the value of the Principal element.

- The first access checker is used to check for read access to a comment. You can read this as saying: compare the root partner key of the current user to the root partner key of the owner of the resource, a comment. If they are equal, then permit the restore operation. Note the use of the usersRootPartnerKey service to retrieve the partner key of the root partner for the current user and the ownersRootPartnerKey service to retrieve the partner key of the root partner of the owner of the resource.

- The second access checker specifies who can create comments, in this case all partner users. The Always element always evaluates to true.

Add this new access policy to the **AccessPolicy.xml** configuration file and restart your servlet container. If you create comments as two users who belong to two different partners, then you will see that each cannot see the comments made by the other. However, two users from the same partner will be able to see each other's comments.

ACL Approach

When an object is created, a default ACL is applied to it unless the application creating the object specifies an ACL. Defined as the System Base ACL in the minimal data set, the default ACL allows the owner of the object to perform any action on the object, and allows users who belong to the same group as the owner to have read access. In our current example, the effect is to limit the comments that a user can see to comments made by themselves and by other users of the same group (try this!).

We now modify the controller classes to allow users to see all comments made by all users about a product. This entails a change to one line in the *getCommentList()* method in the CommentList bizAPI class described above:

Replace:

```
clb.restore(new DataContext(), temp_DsQuery);
```

with:

```
DataContext temp_DataContext = new DataContext();
if (Global.getBoolean("BusinessRule.ProductMgr.viewAllComments"))
{
    temp_DataContext.disableAccessCheck();
}
```

```
clb.restore(temp_DataContext, temp_DsQuery);
```

This additional code checks whether or not the business rule element ProductMgr.viewAllComments is set to true or false.

- If the business rule element is set to true, then the access check is disabled.

- If it is false, then the list of comments restored in the CommentList business object is filtered by the access check mechanism built into the *restore()* call.

In our example, when a comment is created, no ACL is set in the AccessKey field. Consequently, the default access control rules are applied: these give read access to all users who belong to the same group as the owner of the business object and deny access to all other users.

Modifying the BusinessRules.xml File

Add the following element to the ProductMgr element of the **BusinessRules.xml** file:

```
<viewAllComments ChangeOnlyAtBootTime="false"
    controlType="select" button="radio"
    multipleChoice="false"
    runtimeDisplayed="true"
    visible="true" boxsize="45"
    displayQuestion="Enable users to see all comments"
    displayOptions="true,true,false,false"
    defaultChoice="true"
    help="Allow users to see all comments made about a product">
    false
</viewAllComments>
```

If you now restart your Comergent eBusiness System and log in as an enterprise administrator with business rule manager rôle, then you see that there is now a new business rule that enables you to toggle this variable between true and false.

- If you leave the variable set to false, then you can verify that users see only comments that other users in their group have been made.

- By changing this variable to true, you can verify that users can now see all comments irrespective of who made them.

Pagination

In certain circumstances, restoring a list of data beans may mean that your list has more items than you want to display on a single browser page. To enable users to

browse through the complete list, you must use pagination to enable users to view one page after another. This section sketches briefly one approach to pagination using the example described in the previous sections.

You can make use of the *restoreToCache()* and *restoreFromCache()* methods to manage the restoration of the list data bean. The controllers must be passed in two parameters that enable the controller to calculate what subset of the list data bean to display on any particular page.

1. Modify the *execute()* method of the relevant controller classes as follows:

   a. Perform a check to see whether the list data bean has already been created using *restoreToCache()*. You can set a flag in the session or even just a request parameter for this purpose. Create a new comment list bean object.

      • If the flag has been set, then the *restoreToCache()* method has already been called and you do not need to call it again.

      • If the flag has not been set, then, using the comment list bean, call *restoreToCache(*) using the same Query business object used by the *getCommentList()* method above and specify the number of results per page as an int parameter.

   b. Retrieve the two parameters that determine which page of results should be displayed next. For example, PageNumber (the number of the current page) and PageCommand ("None", "Previous", or "Next" to specify whether the user wants to move back or forward through the pages).

   c. Calculate the number of the requested page. For example:

      • If PageNumber=7 and PageCommand=Next, then the user would like to see Page 8 of the result set.

      • If PageNumber=2 and PageCommand=Previous, then the user would like to see Page 1 of the result set.

   d. On the comment list bean created in Step a, call *restoreFromCache(int i)*, passing in the calculated page number from Step c.

   e. Set the following in the request object:

      • The resulting comment list bean restored in Step d.

      • The new page number as calculated in Step c.

      • As required, set attributes to determine whether there are previous or next pages when the new page is displayed. You can use a *restore-FromCache(int i)* call to check if there should be a next page (use the

value one greater than the calculated page number from Step c): it will return false if there is no more data.

2. Modify the **CatalogProductCommentsPopup.jsp** page to add Previous and Next links. Each of these links should be wrapped with a test to check whether the link should be displayed using the attributes created in Step e above. Each link should provide the following parameters:

   a. The PageCommand and PageNumber parameters.

   b. The standard cmd=detail parameter.

Note that the *restoreFromCache()* method restores data using the session and business object type to determine which cached data set to retrieve. Take care that you do not need to maintain two data sets of the same business object type in the same session.

Pagination Controller

You can make use of the PaginatedListController class: this provides a number of helper methods to manage paginated lists. It is found in the com.comergent.reference.apps.common.controller package. To use it, create a controller class that extends it, and implement the three methods:

- protected abstract IDataList restoreAndCacheList() throws ICCException

- protected abstract IDataList retrieveListFromCache()

- protected abstract String getListKey()

For an example of this usage, see the PriceListPartnerAssignmentListController class.

Developer Guidelines

This chapter describes guidelines that will help you maintain and customize your Comergent eBusiness System.

Overview

In working on either customizing your Comergent eBusiness System implementation or creating a Comergent eBusiness System application, bear in mind that your work may be tested in a number of different environments: such as on different application servers, against different database servers, and by users using different browsers. This chapter provides some helpful guidelines to ensure that your application works well and is as bug-free as it can be.

The following topics are items that you should keep in mind while developing applications for the Comergent eBusiness System. It is based on the experience of Comergent eBusiness System developers and a summary of types of bugs found in the releases of the Comergent eBusiness System.

Apart from the feature you are developing, you should also keep in mind to code and test to get coverage on these types of issues. Users of your Comergent eBusiness System or application may work with a combination of one or more of the following items and could potentially find a bug. When you are done with developing your application, you should review your code against this checklist.

Platform Variations

Browsers

If your Comergent eBusiness System application provides a Web-based user interface, then you must make sure that your application works properly and equally well in supported versions of both Netscape Navigator and Internet Explorer browsers.

Typically, if you test your application using only one browser version, then you may miss problems that using a different browser version may expose. These include HTML variations such as <DIV> that are supported by only one type of browser or Javascript methods that assume a browser-specific document object model (DOM). Note that there are also significant differences between the 4.7x and the 6.x and 7.x releases of Netscape Navigator.

See "Browser Usage" on page 203 for further issues relating to users working with a browser interface.

Databases

The Comergent eBusiness System supports three database server platforms: IBM DB2 Universal Server, Microsoft SQL Server, and Oracle Server. Make sure that you test your work against all three database server platforms while you work, and then you will ensure that once you deliver your application, it will run safely in any supported database platform environment.

Application Servers

In your development environment, you may make use of only one application server, such as JRun or Tomcat. Our experience has been that people work in one environment until the feature works, and then assume that it is going to work similarly with other application server. We have found variations between different application servers and so you should make sure that your application runs properly in other application server environments too.

Operating Systems

In principle, application servers written to conform to the J2EE standard insulate application developers from the operating system environment. In practice however, there are differences between the way in which application servers behave from one operating system to another. Consequently, you should test your application on different machines running different applications. At minimum, these should include a version of Windows and a version of UNIX.

Security

As you write your application, bear in mind that access to the application's functionality and business objects can and should be controlled so as not to expose sensitive information or allow unauthorized modification of data. The principal means to do this are:

- Access Policies

- ACLs

- Roles

Access Policies

In Release 7.0 and higher, you should use access policies to protect access to data objects. The use of ACLs is deprecated. See "Managing Access to Data Objects Using Access Policies" on page 93 for more information.

ACLs

Note:	The use of ACLs is deprecated

Data objects to which access must be managed must extend the C3PrimaryRW data object. You should make sure that whenever such a data object is created, that it has an appropriate ACL attached to it. Subsequent accesses to the data object should always be checked to verify that the user is authorized to act on the data object as intended.

Make sure that all of your data objects have gone through this ACL exercise. Make sure your application has implemented the security controls. Double-check that ACLs are set properly in reference data and minimal data sets.

Roles

Your application may need to be run by users with different roles. Some applications change behavior depending on the roles assigned to users and others do not. Typically you should develop and test your application with different users and roles in mind. However, it is very important that you consider all type of roles of users who can potentially access your application and make sure that it behaves as expected.

For example, testing using the reference data often means that you log in as ajones to run internal applications. Since ajones has both the enterprise user and enterprise administrator roles, your application may work as anticipated. What happens when

you log in as djones who is only an enterprise user? Similarly you should cover issues such as reading, writing, updating, and deleting objects created by users of different roles. For example, a customer support representative can read orders created by anonymous, registered, and direct commerce partner users: what privileges should such a user have on orders?

Be sure to test your work by logging as different users and stepping through the application functionality to verify that each user can perform what you intend they should be able to.

Encoding Data in JSP Pages

If you are writing JSP pages to develop the user interface of your application, then you need to take care to use the *ph()*, *pj()*, and *pu()* methods to address some security issues. For example, malicious users can enter scripting commands into data fields that execute when the data is next displayed. Make sure that you use the static methods provided to appropriately encode data strings in HTML pages. See CHAPTER 20, "JSP Pages" for more information.

General Application Issues

XML Messages

Your application has business logic and presentation layers. The business logic layer can be executed by posting XML messages into a Comergent eBusiness System, just as an HTTP request can be posted to the system. These XML messages are another aspect of your application, and they could become the APIs for your application, and mode of integration with other systems. Hence, it is very important that you ask yourself if there is a need to develop and test these XML messages.

Assembly and Configurations

If your application deals with products, then it is very important that you test it with products that are assemblies or which are configurable. For example, if you allow editing of sub-line items, then you should consider how the feature differs between assemblies and configurable products. You should be careful about display and editing of these.

You should also consider sending and receiving assemblies and configurations in XML messages. How will third-party software components deal with them, for example ERP systems, or Ariba and CommerceOne sites? If you allow editing of major line items, then should you allow editing of minor line items? How do you calculate prices for these types of products? These are all issues to consider for these types of products.

Internationalization

Resource Bundles

Have you programmed for an internationalized and localizable product before? The Comergent eBusiness System is a truly global product. You have to make sure that all user-visible pages behave correctly according to the locale of each user.

Make sure that you consider all data fields of business objects to determine whether they should be localized. Make sure that strings used to track the status of objects or that are displayed in drop-down lists are localized suitably. These two steps are important to make your application internationalized and ready for localization. See CHAPTER 32, "Internationalization" for more details.

Locales

Users may choose to work in their preferred locale as they interact with the Comergent eBusiness System. You are responsible to test your application with at least two locales. Typically, developers develop and test their applications using the en_EN locale. You should also normally test your application using the fr_FR locale as early as possible to catch potential localization issues. Including this in your testing again is one more important item to consider. Do not leave internationalization testing until the end of the development cycle because any problems found may require changes to business objects and lookup tables.

Retrieving Locales

You can retrieve a user's current locale using one of these calls:

```
Locale current_Locale = session.getLocale();
Locale current_Locale = ComergentI18N.getComergentLocale();
```

JSP Pages

Pay attention to character encoding. This is set in a meta element of JSP pages. It is commonly set to:

```
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
```

If you change this to accommodate a particular localization effort, then you must do it across all of the relevant pages.

Data

In general, you should test your application using both the scenarios in which only the minimal data has been installed and in which the reference data has been installed. In particular, take care to determine whether your application will require any additions or modifications to the minimal data: that is, must some data be present for your application to work which cannot be created using the Comergent eBusiness System?

Minimal Data

The minimal data set is designed to provide the absolute minimum of data that will enable a Comergent eBusiness System implementation to be started up and for users to create the full implementation using the Web user interface. During your development phase, you should consider whether your application requires any data that cannot be entered through the Web user interface. If this data is required for your application to run successfully, then this should be added to the minimal data set.

Currently, the minimal data set includes root nodes of business object hierarchies and ACLs for business objects. In addition, it contains the basic partner and group business objects, and defines the enterprise master price list.

Reference Data

You are responsible to maintain and test the reference data that should be used to test your application as part of a deployment of our reference system. Make sure that your testing allows for edge cases as part of the reference data and you should review it for consistency with the rest of the reference data set.

In Release 7.0.2 and higher, the reference data is designed to be applied as a layer on top of the minimal data set. You should review your reference data to verify that it makes the right separation between minimal and reference data. See "Minimal Data" on page 202 for more information about the minimal data set.

Sorting and Searching

If you are developing an application that provides a Web interface, then review it to check for places where users can do sorting or initiate a search. Check that the sort and search functionality works correctly and that it treats non-ASCII characters correctly.

Browser Usage

In general, Comergent eBusiness System applications are designed to work in a browser. Consequently, in writing your applications, you should take care to consider typical patterns of browser usage as described below:

- Cookies

- Enter Key

- Back and Forward Buttons

- Session Timeout

- Refresh Button

- Field Types and Lengths

Cookies

Users of your application can set up their browsers with cookies either on or off. Your application should function equally well with either setting. Take care to encode every URL to include session information, and take care not to store information in browser-side cookies. Nonetheless, you cannot assume that you have caught every potential pitfall, and so you should continue to test your applications with both cookies on and off.

Enter Key

If a user hits the Enter key while working in a form on the browser page, then the browser will submit the form. This is functionality that users expect and expect it to work like other applications on their desktop. If you have a user interface component containing form fields, then make sure that hitting the Enter key does not break anything. To fix problems with the Enter key is not trivial, and cannot be fixed in one place for all applications. Your form parameter validation and verification has to be designed with this kind of usage in mind. Make sure you address this when you design your user interface.

Back and Forward Buttons

Users are used to using the browser Back and Forward buttons as part of their navigation through a Web-based application. When these buttons are used, the browser may repost a request or reload a page from its memory. You should consider what effect this can have on your application's business logic.

In particular, there can be a number of issues when the Back button is used to navigate. Applications fail to function properly or at least confuse the user by not behaving as they expect. Consider this when developing your application.

Session Timeout

Browser users may also let their session timeout, either deliberately or accidentally. For example, a user is working with your application, but decides to go for lunch in the middle of completing a task. They come back to continue working with your application and their session has timed out. Now what? How should your application behave here? Typically, the user has to be routed to a suitable login page and then brought back to the application again.

Refresh Button

This is another area of concern for Web-based application developers. Users may use the **Refresh** button either because they are not sure that a particular action completed or has been successful or because they are impatient to see the results of their action. Web applications are also vulnerable to attacks that repeatedly post the same request to the application server.

You should ask yourself: If a user clicks the **Refresh** button, then what should happen? What would a user expect to happen? If a user sees something unexpected happen, then can they undo that effect? Make sure your application behaves the way user expects here.

Field Types and Lengths

Forms are the means by which users enter data into the Comergent eBusiness System. The data is saved into the Knowledgebase by inserting or updating records in the database tables. You should ensure that users are guided as much as possible to enter data of the correct form into each form field. However, your application must also handle situations in which users either accidentally or maliciously enter invalid data. These include: entering a negative number where only positive ones make sense; entering non-integers where only integers are expected; entering alphabetical characters instead of numeric characters.

As far as possible, you should apply client-side validation to form field entries so that bad data is caught as quickly as possible: it is better for the user and saves a server hit as well. However, you should also apply server-side checks and catch bad data gracefully to provide constructive feedback to users.

In particular, when using forms, make sure that you control the maximum length of form field entries in order that users do not enter more data into a field than your application can hold. On the server-side pay particular attention to cases where

users might be able to "attack" your application by entering massively long strings as field values.

Developer Testing

Database Requests and User Operation

When you are done developing your application, use the Comergent eBusiness System logging capabilities to assess your application's performance. The application server log can show you how long it takes your application to service each user request, and what resource usage it takes.

For example, you can use the log viewer tool to check how many database operations your application is making for each request. This is important. This test allows you to verify that the database access is as you expect or makes you aware of how the application interacts unexpectedly with the database server. Developers sometimes find that an apparently simple persist or restore operation gives rise to a database-intensive operation such as a very time-consuming query.

API and Exceptions

As part of your application development you may develop some APIs for other modules to use. It is important that you consider possible error conditions and that your application throws appropriate exceptions. See CHAPTER 33, "Exceptions" for general information about recommended exception policies.

Take care to consider also that the reference data you create is largely used to test the successful running of your application. You should also consider designing for error conditions that are not present in the minimal and reference data sets, but which may arise if data is loaded from some external source.

Javadoc

If you are developing APIs, then it is very important that you provide reasonable Javadoc comments for use by client applications. Do not forget to add package-level documentation using **Package.html** files in each package directory.

HTML Validation

HTML validator tools can check that the HTML produced by your JSP pages are in conformance to the HTML 4.0 standard. It is possible that the HTML produced by your application is not conformant. You should check for this and fix problems. It will also improve the performance of your application.

Threads

In some situations, you may want to initiate a process as a separate thread. For example, typically, a user may execute a request that initiates a long-running process such as index generation. In these situations, you may want to handle this by creating a new thread that executes the task while the original request is completed and a response is returned to the user.

When you do this, the new thread process can continue to run even when the main servlet container process is terminated. To prevent this, you should always call the *setDaemon(true)* method after creating a thread. For example:

```
Thread t = new Thread(...);
t.setDaemon(true);
```

This ensures that these threads will be terminated at the same time as the servlet container is shut down.

File Uploads

You may need to enable the upload of files into the Comergent eBusiness System: typically, this is how partner users will create templates and logos in the Comergent eBusiness System, and it can also be used to upload data such as campaign mailing lists and leads.

You must consider the following:

Forms for File Upload

Use the standard file upload HTML elements to create a file upload form:

```
<FORM ACTION="" METHOD="POST" NAME="UploadLeadForm"
    onSubmit ="return uploadFile()">
<INPUT TYPE="hidden" NAME="cmd" VALUE="LeadUploadStatus">
<INPUT TYPE="hidden" NAME="leadFile" VALUE="">
<INPUT TYPE="file" NAME="leadFileName" VALUE="">
<A HREF="javascript:uploadFile()"><IMG ALIGN="TOP"
    SRC="../htdocs/shared_images/uploadUglyButton1.gif"
    WIDTH=81 HEIGHT=22 ALT="Upload" BORDER="0">
</A>
</FORM>
```

Note the use of a Javascript method *uploadFile()* in the form: it is referenced twice to manage both the case when a user clicks the Upload button and when they hit enter in the file field.

In the JSP page, the *uploadFile()* method looks like this:

```
function uploadFile()
{
    if (trim(document.UploadLeadForm.leadFileName.value) != '')
    {
        document.UploadLeadForm.encoding="multipart/form-data";
        document.UploadLeadForm.leadFile.value=
            document.UploadLeadForm.leadFileName.value;
        document.UploadLeadForm.leadFile.action="<cmgt:link
            app='*'></cmgt:link>"
        document.UploadLeadForm.submit();
        return true;
    }
    else
    {
        alert("You must enter a file name.");
        setFocus();
    }
}
```

The important thing to notice is that the encoding is set to "multipart/form-data". When this request is submitted to the Comergent eBusiness System, some pre-processing is performed on the the request before passing it to the controller used to process the request. The controller can retrieve the uploaded file by calling the FileUploadCache class: this class provide methods to retrieve the uploaded file as an InputStream or InputStreamReader object. For example, this code fragment retrieves the file uploaded from the form described above:

```
String FileID = request.getParameter("leadFile");
InputStream is = FileUploadCache.getInputStream(FileID);
```

From this point on, the controller can determine what to do with the uploaded file as your business logic requires.

Saving Files on the Comergent eBusiness System

Bear in mind that when you save a file in the Comergent eBusiness System, you must bear in mind how the file will need to be accessed in the future. For example:

• If the file is a GIF file, then is it to be displayed to users?

• Is it a data file that can be discarded once it has been processed?

• Will it need to be maintained in a versioned manner?

In general, you should use the *adjustFileName()* method of the ComergentAppEnv class to save files to the file system. This method provides standard ways to specify

what sort of file you are saving and this determines where the file is saved. See "ComergentAppEnv Class" on page 27 for more details.

File Processing

Bear in mind that file processing can be both processor-intensive and error-prone. You need to consider the possibility of offloading the file processing task to a separate thread and what to do if errors occur that mean some or all of the file data is invalid.

One technique to offload processing to a separate thread is to attach the file to message and post it back into the Comergent eBusiness System so that it is processed as a separate request. For example, the following code does this for uploaded lead files:

```
LeadCreateListResultBean result =
    postComergentXMLMessage(is, format);
processResult(result);
```

Here, the *postComergentXMLMessage()* method is used to post the message to the Comergent eBusiness System and to return a result.

```
private LeadCreateListResultBean postComergentXMLMessage(
    InputStream is, String format) throws Exception
{
    ConverterFactory cf = ConverterFactory.getConverterFactory();
    Converter converter =
        cf.getIncomingConverter("LeadCreateListRequest", format);
    converter.setSource(is); // pass the stream that you have read
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    converter.setTarget(out);
    converter.convert();
    String tmp = out.toString();
    String msgURL = CronManagerHelper.getCronMessageURL();
    LocalPost iccPost = new LocalPost(msgURL, null);
    ComergentDocument replyDoc = iccPost.postString(tmp);
    XMLResponseAccessor xmlAccessor = new
    XMLResponseAccessor(ConverterFactory.getNativeMessageCategory());
    BizObjTable retBusObjTable =
        xmlAccessor.xmlReplyToBeans(null, replyDoc);
    LeadCreateListResultBean bean = (LeadCreateListResultBean)
        retBusObjTable.getBean(LeadCreateListResultBean.class);
    return bean;
}
```

Summary

The application development in Comergent eBusiness System involves a number of considerations apart from the actual feature development of your application. By using the topics described above as a checklist, you will be able to meet many of the issues before your customers do!

User Interface and Style Guidelines

Overview

As you work on tailoring the Comergent eBusiness System or on developing a new Comergent eBusiness System application, bear in mind that your work should be consistent with the overall look-and-feel of the Comergent eBusiness System user interface. This chapter provides a summary of the guidelines used by all of the current applications. If you use these to guide your work, then your customizations to existing applications and new applications will be immediately familiar to users and enable them to quickly become comfortable using them.

This chapter also covers the use of the following UI components:

- calendar widget: a flexible, re-usable UI component to support locale-specific date selection and entry. See "Using the Calendar Widget" on page 217.

- tree viewer: a flexible, re-usable UI component to display tree structures of data objects. See "Using the Tree Viewer" on page 220.

- entity picker: a flexible, re-usable UI component to display sets of entities from which selections can be made. See "Using the Entity Picker" on page 222.

Tables and Data Lists

General

When you display a list of business objects to a user, be consistent in how the list behaves:

- If a user creates a new object, then by default the newly-created item should appear at the top of the list.

- If a list is displayed that has no items on it (such as a search that matches no objects), then display a message that makes it explicit that the list has zero items.

Columns

In general, you should use the following left-to-right display order for the table columns:

- Check boxes (if any): if the user can act on the list of business objects by checking or unchecking a check box against each item on the list, then these check boxes should appear as the far left column. As far as the user is concerned, the action is performed on all checked rows of the table simultaneously.

  The action buttons should be displayed at both the top and bottom of the table, preferably above and below the check boxes column. Provide buttons for "Select All" and "Deselect All".

  Examples: Assigning price lists to partners, copying product inquiry lists.

- Name: The first text column should be the "primary name or key" column. Entries in this column must uniquely identify each business object: if the user is able to navigate to the detail of the business object, then this column should provide the link to the detail page.

  Examples: Partner name on the Partner Profile List page, product ID on the Product Catalog page, price list name on the Price List List page.

- Additional information columns: provide other informational columns as required by the application to the right of the Name column. In general, you should make the columns sortable so that the user can sort and group the items by useful attributes.

  Examples: Type, Level, and Category on the Partner Profile List page, Full Names, Roles on the User List page.

- Action buttons: If actions can be performed on individual items, then there should be an action button for each possible action on each row, and these should all be in the far right column of the table. Note that if there is only one relevant action, then you can also display the action as a link: for example, **<u>Delete</u>**. The title of the column should be "Actions".

  A legend to identify the action buttons should be displayed at the top of the page, immediately above the table and preferably above the Actions column. If there is no space at the top of the page, then place the legend at the foot of the table.

  Examples: **Duplicate Promotion** and **Delete Promotion** on the Promotion List page, **Delete Product** and **Assign Rules** on the Price List Detail page.

- Text alignment in columns. Follow these rules where possible:

  - Left justify names, product IDs and other unique identifiers, and text that may vary in length such as description fields.

    Examples: Partner on the Partner Profile List page, Name and Description on the Price List List page.

  - Center fields whose values do not vary in length or which come from a fixed set of values (typically determined at the time of implementation).

    Examples: Level and Type on the Partner Profile List page, Currency and Type on the Price List List page.

  - Right justify fields whose values are numbers and prices. Use a fixed-width font for such fields.

- Sorting by columns: Sort order is indicated by an up or down arrow next to the table heading text. Clicking the table heading text should have the effect of sorting the table by ascending values of that column. If the table is already sorted by the column (as indicated by the arrow), then clicking on either the heading text or the arrow should toggle the sort order between ascending and descending.

  In general, all columns that contain text or status information should be sortable. The field used to sort a column should always be visible.

Formatting

Use the standard text format for names, descriptions, and other text attributes for each column. Names of people should be displayed as Family name (Last name), Given name (First name) and sorting by the column should sort by Family name.

Dates should be displayed in the numerical format (for example, 10/23/2002, not October 23rd, 2002) determined by the user's current locale.

Buttons

Use a consistent convention between buttons and their resulting actions. These are the standard buttons: use these wherever possible.

- **Apply**
- **Assign**
- **Cancel**
- **Delete**
- **Details**
- **Done**
- **Edit**
- **OK**
- **Remove**
- **Save**
- **Update**

Forms

HTML forms are the chief means by which users will add or modify information in the Comergent eBusiness System. Make sure that your forms are clear and well-designed so that a user can enter the information easily. The more guidance you can provide users so that they enter correct and valid information in each field, the better the application works.

In particular, where possible:

- Provide a clear label for each text field and other input components such as drop-down lists, list boxes, radio buttons, and check boxes. Provide explanatory text where appropriate.

- If a field may only take one of a certain number of values, then provide the values as a drop-down list. In general, consider whether the values are to be defined in the JSP page or as the result of a database query: there may be localization issues to address in either case.

- Clearly mark required fields with an asterisk ("*") and provide a legend that reads "* Required Fields".

Text Fields

When a user is creating a new business object, then fields should be blank unless a default value will be used if none is provided by the user. If a default value will be used, then display the default value at the time the user is viewing the form. Validate each text field to determine that the user-entered value is appropriate. Constrain text fields so that users are not permitted to enter more characters than allowed by the business object definition.

If there is space, then provide additional information to help the user enter correct information: for example, "Enter a number with no more than two decimal places." or "Enter the telephone number in the form (xxx) yyy zzzz.".

Drop-Down Lists and List Boxes

Drop-down list and list box values should not be provided in a random order. Users should see the same list of values in the same order each time they visit the page. Where possible, sort values in a drop-down list or list box so that the user can easily find values. For example, order values by alphabetical order (such as US states) or by some well-understood ordering (Gold, Silver, Bronze). Pay attention to internationalization issues that may mean that values are displayed in a different order for a different locale.

Where a drop-down list or list box values provide additional information as part of the text string, then use a fixed-width font. For example:

```
MOD:    Workstation
MOD:    Server
MG:     Computers
OCSA:   Monitor
```

If no default value is provided, then use the text "-- Select --" (or "--" for short values) to indicate to a user that they must select a value from a drop-down list. If the field is optional, then display a blank entry rather than an optional value.

Workflow Conventions

In general, create an object when a user clicks **Add** or **New**. Display the new object with pre-filled fields and then allow the user to edit the object.

Users should specify an assignment of one object to others and then click **Assign**; they should not have to click a button simply to be taken to a page on which they perform the assignment.

If a user wants to modify an object, then typically they should click **Edit** to view the object details. Once they have changed values of fields, they should click **Update** to apply the changes.

Popup Windows

Popup windows are used to enable users to enter information in situations where you do not want to disrupt the main page flow. Follow these guidelines:

- Popup windows should have titles that describe the purpose of the window. For example: "Change Locale".

- Action buttons should be displayed in the lower right of the window.

- Do not use the windows "Popup" or "Dialog" in the text or title of these windows.

Search and Find Windows

When displaying the results of a search, display the search criteria at the top of the page. For example: "Search result for the product ID: MXWS-7550". Include all the search criteria used such as Partner Name and Partner Type.

Registration Pages

Make sure that you enable users to enter their registration information naturally using clearly understood fields. For example:

- Address1

- Address2

- City

- State/Province

- Postal Code

Telephone numbers should be left justified and followed localizable formats. Types (Business, Fax, Cell, and so on) should appear in a separate column or as a drop-down list.

Using the Calendar Widget

The calendar widget is a UI component which can be used to enable users to specify dates. By providing a simple point-and-click component, users can select the desired date from a calendar. The format of the calendar matches the user's chosen locale preferences and automatically validates their selection to prevent them from selecting dates such as March 43rd. It also provides the usability cues that help users to select the correct date using information such as days of the week.

The calendar widget should be used instead of the three drop-down lists currently used to input the date. The calendar widget can also replace any current text fields used to input a date.

There are two supported components to the calendar widget:

- A calendar popup that allows the user to graphically select a date from a calendar: most of the code is in **calendarPopup.js**

- Date formatting and verification of the date entered into the date text field: the code is in **date.js**.

To Use the Calendar Widget

1. Include these JavaScript files in your JSP page.

```
<SCRIPT LANGUAGE="JavaScript" SRC="../js/PopupWindow.js">
</SCRIPT>
<SCRIPT LANGUAGE="JavaScript" SRC="../js/CalendarPopup.js">
</SCRIPT>
<SCRIPT LANGUAGE="JavaScript" SRC="../js/date.js">
</SCRIPT>
<SCRIPT LANGUAGE="JavaScript" SRC="../js/I18N.js">
</SCRIPT>
```

2. Add an input text field and anchored image as follows:

```
<INPUT type="text" size="10" name="<Name of field>" value=""> <A
href="javascript:popupCal.showCalendar('anchor1',
'<Name of form>.<Name of field>','<%=ComergentI18N.getLocaleDate-
Pattern()%>')" NAME="anchor1" ID="anchor1">
<img src="../images/calendar.gif" border="0" alt="Calendar"></A>
```

The name of the input text field must match the name specified in the *showCalendar()* method so that the user's selection is set correctly in this field once it is picked.

3. Define three hidden form elements with the following names.

```
<INPUT TYPE="HIDDEN" NAME="<Name of field>Date" VALUE="">
<INPUT TYPE="HIDDEN" NAME="<Name of field>Month" VALUE="">
<INPUT TYPE="HIDDEN" NAME="<Name of field>Year" VALUE="">
```

4. Add an onsubmit=*processDate*() call to the FORM element. The onSubmit function *processDate()* must at minimum invoke the *extractDateFromDateField()* function provided in the **date.js** Javascript package. This function converts the data from the text field and populates the three hidden input fields. In addition, you can add other processing to the date field such as comparing it to the system date.

5. When processing the form, you retrieve the values of the selected day, month, and year by calling:

   • request.getParameter("<*Name of field*>Date"): as an integer between 1 and 31.

   • request.getParameter("<*Name of field*>Month"): as an integer between 0 and 11.

   • request.getParameter("<*Name of field*>Year"): as an integer, say "2003".

To Replace Three Drop-down Lists

Some current Comergent eBusiness System applications use three text fields in forms to enable users to enter and modify dates. In general, you should work to replace these text fields with an instance of the calendar widget as follows:

1. Include these JavaScript files in your JSP page.

```
<SCRIPT LANGUAGE="JavaScript" SRC="../js/PopupWindow.js">
</SCRIPT>
<SCRIPT LANGUAGE="JavaScript" SRC="../js/CalendarPopup.js">
</SCRIPT>
<SCRIPT LANGUAGE="JavaScript" SRC="../js/date.js"></SCRIPT>
<SCRIPT LANGUAGE="JavaScript" SRC="../js/I18N.js"></SCRIPT>
```

2. Comment out the three drop-down lists and replace them with a text field and a calendar widget icon. Here is the example code. The "anchor1" anchor is used to position the popup calendar near the text field.

```
<INPUT type="text" size="10" name="CreateStartDateField" value="">
<A href="javascript:popupCal.showCalendar('anchor1','query-
Form.CreateStartDateField','<%=ComergentI18N.getLocaleDatePat-
tern()%>')" NAME="anchor1" ID="anchor1">
<img src="../images/calendar.gif" border="0" alt="Calendar"></A>
```

The text field in this case is named CreateStartDateField.

3. Define the three hidden form elements with the names of the three drop-down lists that are being replaced.

```
<INPUT TYPE="HIDDEN" NAME="CreateStartDate" VALUE="">
<INPUT TYPE="HIDDEN" NAME="CreateStartMonth" VALUE="">
<INPUT TYPE="HIDDEN" NAME="CreateStartYear" VALUE="">
```

4. Populate these three form fields from the date text field defined in Step 3.

> **Note:** It is important to note that the *extractDateFromDateField()* function requires the field names follow the following naming convention. The primary design goal of the *extractDateFromDateField()* function is to reduce the amount of code needed to verify and set the three date variables to one Javascript function call.

```
function verifyDateField()
{
    var theDate;
    theDate = extractDateFromDateField("Creation Start Date",
        "queryForm.CreateStart",
        "<%=ComergentI18N.getLocaleDatePattern()%>",
        false, true);
    if (theDate == false) return false;
}
    Name = <dateForm>
    Text field = <dateForm> + "DateField"
    Year value = <dateForm> + "Year"
    Month value = <dateForm> + "Month"
    Date value = <dateForm> + "Date"
```

Naming Example
If Name is 'CreateStart", then the name of the text field is "CreateStartDateField", and *extractDateFromDateField()* will extract the date and set the following hidden fields: CreateStartDate, CreateStartMonth, and CreateStartYear.

To Replace a Text Field by the Calendar Widget
Some current Comergent eBusiness System applications use a single text field in forms to enable users to enter and modify dates. In general, you should work to replace each such text field with an instance of the calendar widget as follows:

1. Include these JavaScript files in your JSP page.

```
<SCRIPT LANGUAGE="JavaScript" SRC="../js/PopupWindow.js">
</SCRIPT>
<SCRIPT LANGUAGE="JavaScript" SRC="../js/CalendarPopup.js">
</SCRIPT>
<SCRIPT LANGUAGE="JavaScript" SRC="../js/date.js"></SCRIPT>
<SCRIPT LANGUAGE="JavaScript" SRC="../js/I18N.js"></SCRIPT>
```

```
Comment out the text field and replace them by a text field and a
calendar icon.
<INPUT type="text" size="10" name="CreateStartDateField" value="">
<A href="javascript:popupCal.showCalendar('anchor1','query-
Form.CreateStartDateField','<%=ComergentI18N.getLocaleDatePat-
tern()%>')" NAME="anchor1" ID="anchor1">
<img src="../images/calendar.gif" border="0" alt="Calendar"></A>
```

The text field in this case is named as 'CreateStartDateField', which is the name of the original text field.

Notes

The call to *showCalendar()* takes these arguments:

1. The Anchor: the position where the calendar will pop up.

2. The Form element: the calendar widget reads the date from this element and opens the calendar to show the current date. If there is no value in this form element, then it opens the calendar with the current date; otherwise the current displayed date is shown as selected. Also on selecting a date by clicking on the calendar this field is populated from the calendar.

3. The Format String: the calendar expects a date format and returns the date in the form field in the same format.

Using the Tree Viewer

A useful UI component provided with the Comergent eBusiness System is a tree viewer component. It provides a means to display complex hierarchical information in the form of a tree of expandable and collapsible nodes. You can use the tree viewer to display hierarchies such as the product catalog and the model group hierarchy. Different object types can be displayed in the hierarchy which can be represented by different icons and which execute different message types when selected.

You must implement the tree view as a frame. The frame must be populated using a message type that is processed by a controller that extends the TreeViewController class as described in Step 2 below.

Follow these steps to create a tree view for your hierarchy:

1. Create a tree view class that implements the TreeViewEntity interface. This interface extends the PresentationEntity interface and among the methods you must implement are the main "tree" methods:

   • *getID()*

- *getName()*

- *getDisplayName()*

- *getType()*

- *getChildren()*

- *getTopLevelEntities()*

2. Create a controller that will process the request to populate the frame used to display the tree. This controller must extend the abstract TreeViewController class and you must implement the *newTreeViewEntity(String s)* method. This method must return an instance of the tree view class created in Step 1. The String parameter may not be used: this will depend on your implementing class.

   The *execute()* method of the TreeViewController class invokes the *init()* method which returns as a String the TreeViewEntity returned by the *newTreeViewEntity()* method. The String rendering is set as the value of a request attribute named "TreeView.CodeBody".

3. Create the JSP page used to display the tree within a browser page. The tree must be created within a frame on the JSP page. The frame should be populated using the TreeView message type. For example:

```
<FRAME src='<%= link("productMgr", "TreeView",
"TreeView.ParamFile=../js/cmgtProdMgrTreeViewParam.js&Tree-
View.MsgType=productMgr&TreeView.Cmd=ProdMgrPCHierarchy") %>'
name="TreeView" FRAMEBORDER='no' NORESIZE SCROLLING="no">
```

Note that the *link()* method passes a standard message type, TreeView, whose definition is:

```
<MessageType Name="TreeView">
    <JSPMapping>
        ../uiComponent/TreeViewFrameSet.jsp
    </JSPMapping>
    <ControllerMapping>
        com.comergent.dcm.caf.controller.ForwardController
    </ControllerMapping>
```

```
</MessageType>
```

You must pass as parameters the name of the parameter Javascript file **cmgtProdMgrTreeViewParam.js** and the name of the message type defined in Step 5 as the TreeView.Cmd parameter. The parameter file defines the mapping between node types and the icons used to represent them.

You must ensure that the parent page defines a *Dispatch((modName, msg, val)* Javascript method. This method defines the module name, message type, and a value (of the selected tree node). This method is invoked when users select nodes in the tree: typically, it is used to populate a detail frame that displays information about the selected node.

4. Create a JSP page that will be used to render the tree view. This JSP page must have the following scriptlet:

```
<%= (String) pu(request.getAttribute("TreeView.CodeBody")) %>
```

This scriptlet retrieves the TreeViewEntity in the form of the String created in Step 2. It places a hidden frame of content in the tree view frame and is accessed to retrieve the tree model data.

5. Create a message type that specifies the controller created in Step 2 and the JSP page created in Step 4. For example:

```
<MessageType Name="ProdMgrPCHierarchy">
    <JSPMapping>
        ../uiComponent/TreeViewCode.jsp
    </JSPMapping>
    <ControllerMapping>
        com.comergent.appservices.productService.controller.-
            TVProductCategoryController
    </ControllerMapping>
</MessageType>
```

Using the Entity Picker

The entity picker provides a UI component that can be used to help users of the Comergent eBusiness System select data objects from a number of possibilities. It supports these views:

• Hierarchy: navigate the object hierarchy to make selections

• Search: perform a search using specified criteria

• Flat list: select from a list of all of the objects, using pagination to move through a long list

For any particular task, you should decide which of the views you want to support: your picker can support one or more of them, but need not offer them all.

Follow these steps to create an hierarchy entity picker for your needs:

1. Create a tree view class that implements the TreeViewEntity interface. This interface extends the PresentationEntity interface and among the methods you must implement are the main "tree" methods:

    *   *getID()*

    *   *getName()*

    *   *getDisplayName()*

    *   *getType()*

    *   *getChildren()*

    *   *getTopLevelEntities()*

2. Create a controller class that extends the EntityPickerHierarchyViewController class. You must implement the *newTreeViewEntity(String s)* method. This method must return an instance of the tree view class created in Step 1. The String parameter may not be used: this will depend on your implementing class.

3. Create a message type that is to be used to display the EntityPicker window. Typically, this is part of a URL that is the HREF attribute of a **Browse...** button. For example, you might define the following message type:

```
<MessageType Name="EPMainFrame">
    <JSPMapping>
        ../uiComponent/EntityPickerFrame.jsp
    </JSPMapping>
    <ControllerMapping>
        com.comergent.dcm.caf.controller.ForwardController
    </ControllerMapping>
</MessageType>
```

Use this in conjunction with a form such as:

```
<FORM name="picker" method="post"
    action='<%=link("productMgr", "EPMainFrame")%>'>
    <INPUT type="hidden" name="EPModule" value="visualModeler">
    <INPUT type="hidden" name="EPHierarchyView"
        value="DisplayMyHierarchyObjectPicker">
    <INPUT type="hidden" name="EPParam"
        value="../js/cmgtVM_MGMPParam.js">
    <INPUT type="hidden" name="SingleSelect" value="true">
```

```
</FORM>
```

The form must define the EPModule, EPHierarchyView, and SingleSelect parameters.

4. Create the message type specified by the EPHierarchyView parameter of Step 3. This message type must map to the controller class created in Step 2 and to the JSP page to be used to render the hierarchy: for example, **EntityPickerHierarchyFrame.jsp**.

```
<MessageType Name="DisplayMyHierarchyObjectPicker">
    <JSPMapping>
        ../uiComponent/EntityPickerHierarchyFrame.jsp
    </JSPMapping>
    <ControllerMapping>
        com.comergent.apps.productMgr.controller.-
            MyHierarchyPickController
    </ControllerMapping>
</MessageType>
```

Follow these steps to create a search entity picker for your needs:

1. Create a search controller to process the search. The search controller must extend the EntityPickerController and implement the following methods:

    *   *newPresentationEntity()*

    *   *constructQuery()*

    *   *getDataContext()*

    *   *getSortFields()*

    *   *getAscending()*

    These methods are used to determine the presentation entity to be used, together with the form of the query that will retrieve the objects.

2. Use the **EntityPickerSearchFrame.jsp** to display your Search frame. In passing the request to this JSP page, specify the following paremeters:

    *   EPParam parameter to specify the parameters Javascript file to be used by the page. This sets the icon images to be used by the page.

    *   EPModule parameter specifies the application module.

    *   EPSearchConsole specifies the message type to be used to display the search console. The search console is the frame that specifies the search criteria and search values.

- EPSearchView specifies the message type to be used to display the search results.

3. Create message types for the search console and search view message types defined in Step 2. For example:

```
<MessageType Name="EPProdSearchView">
    <JSPMapping>
        ../uiComponent/EntityPickerListView.jsp
    </JSPMapping>
    <ControllerMapping>
        com.comergent.appservices.productService.controller.-
            EPProductSearchViewController
    </ControllerMapping>
</MessageType>
<MessageType Name="EPProdSearchConsole">
    <JSPMapping>
        ../productMgr/ProdMgrEPProdSearchConsole.jsp
    </JSPMapping>
    <ControllerMapping>
        com.comergent.dcm.caf.controller.ForwardController
    </ControllerMapping>
</MessageType>
```

Use search controller created in Step 1 as the controller in the search view message type.

4. Create the JSP pages declared in Step 3. The search console must provide the search criteria you support to search on the objects (such as name, ID, and so on).

Follow these steps to create a flat list entity picker:

1. Create a controller that extends the EntityPickerController. You must implement the following methods:

   - *newPresentationEntity()*

   - *constructQuery()*

   - *getDataContext()*

   - *getSortFields()*

   - *getAscending()*

   These methods are used to determine the presentation entity to be used, together with the form of the query that will retrieve the objects.

2. Create a message type that maps to the controller created in Step 1 and a JSP page that supports pagination. For example:

```
<MessageType Name="EPProdListView">
    <JSPMapping>
        ../uiComponent/EntityPickerListView.jsp</JSPMapping>
    <ControllerMapping>
        com.comergent.appservices.productService.controller.-
            EPProductListViewController
    </ControllerMapping>
</MessageType>
```

Images

Templates for images used in the administration pages are provided as Photoshop template (***.psd**) files.

CHAPTER 20   *JSP Pages*

This chapter presents a detailed description of the JSP pages of the Comergent eBusiness System and how they may be customized. It covers:

- "Wait JSP Pages" on page 244
- "Redirecting to Full Page Access" on page 245

JSP Page Location

The JSP pages are installed into the ***debs_home*/Comergent/** directory and subdirectories that correspond to the Comergent eBusiness System applications. Each supported locale has its own directory structure which replicates the application subdirectories.

For example, the JSP pages used by the Partner Manager application in the English-United States locale are installed in the ***debs_home*/Comergent/WEB-INF/web/en/US/partnerMgr/** directory.

Page Structure

Almost all of the JSP pages reflect the same basic structure:

- Standard file comment template: optional for external page development, but helpful if you are programming in the Comergent eBusiness System. Start each page with the following DOCTYPE declaration:

  ```
  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
  ```

- Page directives: this should include the JSP pages described in "Included JSP Pages" on page 231. Any custom tag libraries must be referenced in this section. You should include the following on every JSP page:

  ```
  <% page contentType="text/simple; charset=UTF-8" %>
  ```

- Import statements: Try to import only those packages and classes that are needed. Imported packages should be ordered from the most fundamental to more specific ones.

  Use the page directive as follows to import packages:
  ```
  <@ page import="com.comergent.package.*" %>
  ```

- Attribute parameters declaration: This section should include all the attributes and parameters that the JSP page is expecting. The name, class, type, and scope should be specified. This section should serve as documentation for the JSP page. JSP pages that do not expect attributes should include an empty section.

  ```
  <%--
  ```

```
      ** Request Attributes
      InventoryCollectionBean inventoryCollectionList
      Vector of InventoryCollectionListBean$InventoryCollectionBean
           inventoryCollectionVector
      BigDecimal key
      String name
      ** Request Parameter
      ** Session Attributes
      ** Applications Attributes
--%>
```

- Bean referencing section: This section contains the actual definition of the beans used in the JSP page. The section should be delimited by comments that mark the beginning and end of the section as follows:

```
<%-- $$BEGIN USE BEAN --%>
<%
MyBean myBean =
    (MyBean) session.getAttribute("myBean");
%>
<%-- $$END USE BEAN --%>
```

- Scriptlet declaration: This is an optional section for declarations to define variables and methods at the class-scope level of the generated JSP servlet. Declarations made between these tags are accessible from other declarations and scriptlets in your JSP page and from other servlets. You should avoid declaring member variables in JSP pages.

  This section should be delimited with comments to mark the beginning and end as follows:

```
<%-- $$BEGIN DECLARATIONS --%>
<%!
      int numOfRequests=0;
%>
<%-- $$END DECLARATIONS --%>
```

Attention:	Remember that objects are shared between multiple threads being executed in the same instance of a servlet. To guard against sharing violations, synchronize objects where necessary.

- General scriptlet section: In this section, all the calculations that could be performed up front should be done. The results to be displayed should be assigned to variables:

```
<%
storefrontOrderStateMachine =
    StorefrontAPI.getFactory().getStorefrontOrderStateMachine();
Vector transitions =
```

```
      storefrontOrderStateMachine.getValidStateTransitionsForLine(
      orderStatus, lineStatus);
int noTransitions = transitions.size();
%>
```

- HTML section: This section comprises the mixture of HTML and JSP scriptlets that together generate the Web page. This section should be written in compliance with the XHTML Transient DTD, which can be summarized as follows:

  All tag and attribute names should be in lower case (for example <html>, <head>, <body>, and so on). All attributes values should be inside quotes (for example, color="red", width="1", and so on). Tags should be nested correctly as follows:

  ```
  <p> This <b> is </p> </b> #wrong
  <p> This <b> is </b> </p> #correct
  ```

  All tags must have a matching closing tag or be closed. Almost all HTML tags have closing tags except for tags such as the br, hr, and input tags. The Comergent Technologies-recommended way to handle those tags are:

  The br tag should be written as follows:

  ```
  <br/> #wrong: this will cause Netscape trouble.
  <br /> #correct: note the space between the 'br' and the slash
  ```

  Input tags should be as follows:

  ```
  <input name="user" type="text" />
  ```

- Head section: this section must include any meta information and declare the cascading stye sheet to be used.

- Javascript section: try to use the standard Javascript libraries provided by the Comergent eBusiness System. See "Javascript" on page 235.

- Body section: JSP scriptlets in the body section should be reduced to a minimum, which includes code that cannot be calculated in the declaration or general scriptlet sections. The following are the suggestions for different points:

  Loops: Normal Java looping instructions (for, while do, and do while) should be used for loops, the tag cmgt:list is now deprecated due to performance problems. An optional section should follow the loop

instruction where the values are calculated for that iteration and assigned to variables. Those variables are referenced in the loop body using expressions.

Conditionals: the Java if statement should be used for conditionals, inline conditionals could be used if it will improve readability.

Expressions: expressions should be reduced to variable referencing or inline conditionals.

Tags: The tags cmgt:encode, cmgt:link, cmgt:list, cmgt:if, cmgt:getProperty, jsp:getProperty, and jsp:setProperty are deprecated. These tags are replaced by Java utility methods. See "Scriptlets" on page 233.

HTML attributes values: if tags or expressions are used to calculate HTML tags attributes values, then XML quotes should be used as follows:

Wrong way:
```
<input name="InvCollection.<%=partnerKey %>" type="checkbox"
    value="<%=(partnerKey==1)?"checked":""%>" />
```

Correct way:
```
<input name="InvCollection.<%=partnerKey %>" type="checkbox"
    value='<%=(partnerKey==1)?"checked":""%>' />
```

- Comments: in general, you should use the JSP comment tag:

```
<%-- jsp:useBean id="pgno" type="String" scope="page" --%>
```

This comment is visible to readers of the JSP page, but does not generate HTML visible to readers of the generated Web page. Use the standard HTML comment tags to embed comments in the HTML page.

Avoid using comments of the form:

```
<!-- jsp:useBean id="tempBean"/ -->
```

Some servlet containers misinterpret this syntax.

Included JSP Pages

Comergent eBusiness System JSP pages declare the same error page and an included JSP page:

- Error page: On installation, the standard error page, **error.jsp**, is installed in *debs_home*/**Comergent/WEB-INF/web/en/US/**. It provides a brief error message together with the error stack trace. The stack trace is printed

within HTML comment tags `<!--` and `-->` so that it is not visible to the user.

• The included JSP page, **cmgtinclude.jspf**, is installed in *debs_home***/Comergent/WEB-INF/web/en/US/common/**. It is used to declare the ComergentSession and ComergentRequest objects and to provide standard scriptlet methods. See "Scriptlets" on page 233. It also declares the Comergent tag library. If you include **cmgtinclude.jspf**, then do not declare the tag library anywhere else in your page.

You can use a flag in the **cmgtinclude.jspf** JSP page to generate useful debugging information in each generated HTML page. In the *writeDebugInfo()* method, set the doDebug variable to true. This causes a comment block to be generated at the top of each HTML page along these lines:

```
<!-- START_OF_WRITE_DEBUG_INFO
    controller=com.comergent.apps.catalog.AdvisorController
    jsp=null
    uri=/Comergent/en/US/catalog/ProductAdvisor.jsp
    request's params
        sortCriteria=default
        OP=
        cmd=advisorWizard
        pathIndex=0
        advisorCmd=continue
END_OF_WRITE_DEBUG_INFO -->
```

Note that for security reasons you should set this flag to false in any production system. Note also that included JSP pages do not necessarily get re-compiled. You may have to delete the compiled servlet from the servlet container's working directory to be sure that the debugging information stops being generated.

See "JSP Fragments" on page 242 for information about including JSP fragments.

Using the Session Context

The Comergent eBusiness System manages session information when it is interacting with external systems. To do this, a ComergentSession object is used to wrap the servlet container's session object. Consequently, the standard use of the session context in the jsp:useBean tag is deprecated.

Instead of using:

```
<jsp:useBean name="myBean" type="com.comergent.bean.MyBean"
    scope="session" />
```

You must use:

```
<%
    com.comergent.bean.MyBean myBean =
        (com.comergent.bean.MyBean) session.getAttribute ("myBean");
%>
```

You can continue to use:

```
<jsp:useBean name="myBean" type="com.comergent.bean.MyBean"
    scope="request" />
```

and

```
<jsp:useBean name="myBean" type="com.comergent.bean.MyBean"
    scope="application" />
```

If you do use the <jsp:useBean> tag, then make sure that the Bean class satisfies these conditions:

•   it must have a default public constructor

•   it must provide accessor methods for all of its class variables

Scriptlets

You use scriptlets to manage the dynamic generation of the HTML in the JSP pages:

•   to transform Strings

•   to provide looping and conditional logic

•   to provide standard macros

The JSP page, **cmgtinclude.jspf**, provides several standard methods that almost all JSP pages use. You should always include this page in any JSP pages that you modify or create. It provides:

•   *cmgtText()*: this method is used to localize text used in scriptlets. Use it to return a localized string as follows:

    cmgtText(textID, textString)

The textID is used to retrieve the corresponding text string from the appropriate resource bundle. The textString parameter is returned if no matching id is found in the resource bundle. It is also used as the default value of the ***_en_US.properties** file if the tool provided by Comergent Technologies is used to generate the properties file. For example:
    <%

```
...
    out.println(cmgtText("*","My text"));
...
%>
```

A second form of the method takes an additional array argument:

```
cmgtText(textID, textString, objectArray)
```

Use this form when a text string uses a number of values which may be re-arranged from one locale to another. For example:

```
<%
...
    String[] values = new String[2];
    values[0] = userBean.getFirstName();
    values[1] = userBean.getLastName();
    out.println(cmgtText("*","My first name is {0} and my second
        name is {1}", values));
...
%>
```

The default value of this property is "My first name is {0} and my second is {1}", but a different locale may re-arrange this to use, say in French, "Mon nom de famille est {1} et je m'appelle {0}".

- *cmgtTextBundle()*: this method is used in the same way as *cmgtText()*, but an extra parameter enables you to specify the name of the resource bundle to be used.

```
cmgtTextBundle(textID, textString, bundle)
cmgtTextBundle(textID, textString, objectArray, bundle)
```

- *formatPrice()*: this method is used to display currency information.

- *link()*: this method is used to generate URLs in the generated Web pages. There are several different forms of this method, the most common of which is:

```
link(String app, String cmd, String param)
```

In this form, the first String parameter sets the application, the second the cmd parameter, and the third defines any parameters to be set in the URL. For example:

```
link("advisor", "addProduct", "productKey=12");
```

will generate:

```
http://<server>:<port>/Comergent/advisor?cmd=addProduct&
    productKey=12
```

- *ph()*: use this method to convert HTML characters into their escape sequence (for example, "<" to "<"). Use this method to encode all dynamically-created text in the body of a JSP page.

- • *pj()*: use this method to convert Javascript characters to their escape sequence (for example, " " to "%20"). Use this method to encode all dynamically-created text used in Javascript scripts.

- • *pu()*: use this method to present Java objects without encoding.

Implicit Objects

Take care not to use the names of the implicit objects in your scriptlets. In particular, do not use exception as the name of an Exception object.

Javascript

Some standard Comergent eBusiness System Javascript functions are provided in Javascript files in the *debs_home*/**Comergent/en/US/js/** library directory. Note that each supported locale (*la_CO*) must have a corresponding *la*/*CO*/**js/** sub-directory under *debs_home*/**Comergent/**.

You can include these Javascript functions in a JSP page using elements along these lines:

```
<script language="JavaScript" type="text/javascript"
    src="../js/genericUtil.js"></script>
```

Note that the Javascript file **com_Main.js** defines most of the widely used Javascript functions, and in particular is the one that invokes the pickStyleSheet() function. See "Cascading Style Sheets" on page 238 for more information about cascading style sheets.

Forms

A CSS, **form.css**, should be used to display HTML forms. This ensures that all the Comergent eBusiness System forms present a consistent look-and-feel. A form is built up of a number of <div> elements, one for each row, each of which has the class "row". Individual rows will typically have two <span> elements: one for the label (whose class is "label") and one for the value input field (whose class is "value").

If the value input field is a drop-down list or list box, then you can populate the field using <div> elements within the value <span> element. You can use the cic:select and cic:options tags to create a drop-down list, and the rendered attribute to control what fields get displayed.

Use the fieldset element to draw a border around the form: this helps to group the form elements visually.

Example

The following fragment of a JSP page provides an example of the use of these styles and tags. Note the use of the cic:options tag to populate a drop-down list of values from a LookupResult array. In this example, the array has been populated like this:

```
LookupResult[] currencyList =
    CommerceUtils.getListOfValidCurrencyLookupResults(partnerKey);
```

The "faded" class is used to make text fainter.

```
<form name="newList" method="post" target=""
    action="<%=link("*", comergentRequest.getMessageType())%>">
<fieldset class="userinfobox">
<div class="row">
    <span class="label"><span class="faded">
    <cmgt:text id='cmgt_commerce/UserInfoBox_9'
        bundle='commerce.UserInfoBoxResources'>Customer Type:
    </cmgt:text></span>
    </span>
    <span class="value">
    <div id="showVertical">
        <span><%=ph(currentVertical)%></span>
    </div>
    <div id="editVertical" style="display:none;">
        <cic:span rendered="${empty verticalList}"
        value="${localizedUserInfoBox['NA']}"/>
        <cic:select rendered="${not empty verticalList}"
            name="verticalMarkets">
        <cic:options var="vertical" valueRef="${vertical.code}"
            labelRef="${vertical.string}"
            datasourceRef="${verticalList}"
            selectedValue="${currentVerticalCode}"/>
        </cic:select>
    </div>
    </span>
</div>
<div class="row">
    <span class="label"><span class="faded">
    <cmgt:text id='cmgt_commerce/UserInfoBox_10'
    bundle='commerce.UserInfoBoxResources'>Currency:</cmgt:text>
    </span>
    </span>
    <span class="value">
    <div id="showCurrency">
```

```
            <%=ph(currentCurrency)%>
        </div>
        <div id="editCurrency" style="display:none;">
            <cic:span rendered="${empty currencyList}"
                value="${localizedUserInfoBox['NA']}"/>
            <cic:select rendered="${not empty currencyList}"
                name="currencyList">
            <cic:options var="currency" valueRef="${currency.code}"
            labelRef="${currency.string}" datasourceRef="${currencyList}"
            selectedValue="${currentCurrencyCode}" />
            </cic:select>
        </div>
        </span>
</div>
<div class="row">
    <span class="label">
    <span class="faded"><cmgt:text id='cmgt_commerce/UserInfoBox_11'
    bundle='commerce.UserInfoBoxResources'>Last Modified:</cmgt:text>
    </span></span>
    <span class="value">
        <span><%=ph(updatedDate)%></span>
    </span>
</div>
</fieldset>
<input type="hidden" name="Operation" value="Edit">
</form>
```

Form Submits

Make sure that you use Javascript correctly to handle the processing of form data. If you write the Javscript incorrectly, then it is possible for form data to be submitted twice.

Users may initiate a form *submit()* call either by clicking a submit button or by hitting **Enter** while in the form. If you include an *onSubmit=function()* call in your form definition, then either make sure that the function itself does not include a *submit()* call, or if it does, then make sure that you return false immediately after the submission. The form of the onSubmit attribute should follow this form:

```
<form ... onSubmit="return myEventHandler()" ... >
```

In general, write the function so that it must explicitly return true or false whatever path of execution is taken.

Frames

In general, the Comergent eBusiness System deprecates the use of frames in its pages. JSP pages should be written with the assumption that the pages may be

displayed within a frameset displayed bya Web server or other application. Consequently, when you specify navigation links from one page of the Comergent eBusiness System to another, you should use "_self" or "" in setting the target. For example:

```
<cic:outputLink target=""
    href="${cic:link('*', 'ApprovalDataDisplay',
    cic:concat('ShoppingCartKey=',
    approval.dataBean.shoppingCartKey), false)}">
```

The Comergent eBusiness System can be run in two modes controlled by the InFrameEnvironment system property:

*   If you set the value of this property to "false" (the default value), then the Comergent eBusiness System runs in the whole browser window and it provides the top-level navigation bar.

*   If you set the value of this property to "true", then the Comergent eBusiness System can run in one frame of a frameset, and the top-level navigation bar is suppressed.

However, if you use frames, then bear in mind that the child frames should be populated by using URLs that point to the Comergent eBusiness System.

For example, suppose that you wish to generate a Web page by using two frames. Each frame must be generated by a JSP page:

```
<FRAMESET rows="100,*" border="0" framespacing="0" frameborder="NO">
    <FRAME name="navigation"
    src="<%= link("enterpriseMgr", "HomeNavDisplay") %>"
        MARGINWIDTH="0" MARGINHEIGHT="0" scrolling="no">
    <FRAME name="data"
        src="<%= link("enterpriseMgr", "HomeDataDisplay") %>"
        MARGINWIDTH="10" MARGINHEIGHT="0" scrolling="Auto">
</FRAMESET>
```

The *link()* method is used to generate dynamically the URLs that populate the frames. The application and command parameter ensure that the right path and parameters are generated as parts of each URL.

Cascading Style Sheets

The Comergent eBusiness System makes extensive use of cascading style sheets (CSS) to ensure a uniform look-and-feel user interface across applications. You should use CSS for the following reasons:

- CSS is a standard layout language for the Web. It provides a powerful mechanism to manage a constent look-and-feel for a Web site across many pages.

- It is easy to author by hand.

- Using CSS reduces bandwidth usage because a single style sheet can specify styles for multiple pages. Once a browser has cached the style sheet it does not need to be downloaded each time a reference is made to it.

- It helps to separate style from content.

The main cascading style sheets are incorporated into JSP and HTML pages using the Javascript function *pickStyleSheet()* provided in the **com_Main.js** Javascript file.

If you want to make changes to the look-and-feel of Comergent eBusiness System Web pages, then you should take care to modify the appropriate cascading style sheet rather than making changes on individual JSP or HTML pages. Do not specify styles within individual elements: this will make it very difficult to maintain a consistent look-and-feel.

If you need to introduce a new CSS for an application, then create a directory for the application under *debs_home*/**Comergent/en/US/css/***application*/, and place your new style sheets in this location. References to the style sheets should take this form:

```
<link rel="stylesheet" href="../css/application/custom.css"
    type="text/css">
```

By modifying the cascading style sheet for a particular locale, you can customize the user experience for the locale.

Comergent eBusiness System Style Sheets

The following stylesheets are provided out-of-the-box:

- **basestylesheet.css**: provides generic styles for thebasic HTML elements such as HTML, BODY, and so on.

- **buttons.css**: provides the styles for buttons. See "Buttons" on page 240 for more information.

- **calendar.css**: used to display calendars in forms.

- **color-csr.css** and **color-customer.css**: used to render colors for pages that are viewed by both customer users and enterprise customer service representatives.

- **data-table.css**: used to render tables displayed to users.

- **form.css**: used to display forms to users. See "Forms" on page 235 for more information.

- **widget.css**: used to display widgets to users.

In addition, there are the following legacy style sheets maintained to support compatability with previous releases of the Comergent eBusiness System:

- **ie_main.css**

- **internal.css**

- **nn_main.css**

In general, you should avoid customizing these style sheets because their use is deprecated.

Buttons

In general, you can use a combination of CSS and JSP tags to display buttons so that they are consistently and efficiently displayed in the Comergent eBusiness System. Each button can be displayed by specifying its class and its display text.

A CSS, **buttons.css**, provides styles for the following types of buttons commonly used in the Comergent eBusiness System:

TABLE 7. **Button Styles**

Button	Style	Background Image
Normal	normal-button	**images/btn_normal_*x*.gif**
Thin	thin-normal-button	**images/btn_thin_normal_*x*.gif**
Focus	focus-button	**images/btn_focus_*x*.gif**
Thin Focus	thin-focus-button	**images/btn_thin_focus_*x*.gif**
Mini	mini-button	**images/btn_mini_resolve.gif**

For example:

```
<a class="normal-button-small"
href="<%=link("*", "WorkspaceDataDisplay")%>">
```

```
<%=ph((String) localized.get("Cancel"))%>
</a>
```

or, using the cic:output tag:

```
<cic:outputLink rendered="${!(isCSR)}"
css="normal-button normal-button-small right"
href="javascript:changeQuantities();" >
<cic:span value="${localized["Update']}"/>
</cic:outputLink>
```

Depending on the text for the button, you should select the appropriate width for the button. Except for the mini-button style, each style of button supports the following sizes:

- small: width 54px

- medium: 71px

- large: 120px

- jumbo: 150px

Tables

Tables of data should be displayed using a combination of the JSP tags and CSS stylesheets to ensure a common look-and-feel of tables across the Comergent eBusiness System. See "cic:table Tag" on page 387 for a description of the cic:table tag.

The tag makes use of some pre-defined styles, notably the data-table style for the general table class, and the "normal" and "alternate" classes to manage the display of rows in tables.

Securing JSP Pages from Cross-Scripting Attacks

Bear in mind the possibility of a cross-scripting attack by a malicious user in which the user enters a scripting command through a text field on one of your Web pages. To protect against this sort of attack, you must use the methods provided to encode user-entered data before it is displayed in the browser's Web page.

In scriptlets, use the *ph()*, *pj()*, and *pu()* methods described in "Scriptlets" on page 233. In tags, use the encode tag (<cmgt:encode>) described in "encode Tag" on page 353.

JSP Fragments

If you create a re-usable JSP page fragment that can be used in several places in different JSP pages, then you can use a static include. However, note that in Release 7.0 and higher, the use of static includes is deprecated: instead, consider using dynamic includes using the <jsp:include tag>.

Note:	If there are no child elements of the jsp:include tag, then use the single tag form of the jsp:include tag: <jsp:include ... />. Using the closing tag can cause problems for some servlet containers such as Tomcat 5.5.x.

If you do use a static include, then follow these guidelines:

*   the naming convention is to give these files the jspf file suffix (for example, **PopupEditDetails.jspf**). See "JSP Page Naming Conventions" on page 242 for further information.

*   Make sure that these fragments do not include the **cmgtinclude.jspf** page.

*   Include in the JSP pages at the appropriate position:

    ```
    <%@ include file="../includes/PopupEditDetails.jspf" %>
    ```

Debugging JSP Pages

While creating new JSP pages or modifying existing JSP pages, you may find it helpful to insert the following at the head of the JSP page:

```
<%@ page buffer="1024kb" %>
```

The buffer attribute helps you to make full use of the errorPage attribute of the page directive, so that you can forward the output to a user-friendly page for debugging.

You should remove this tag before deploying the JSP pages into a production environment.

JSP Page Naming Conventions

JSP page file names should follow the following format. This file naming convention scheme requires that the containing directory encode the "application name". For example, for those JSP pages belonging to Lead Management, the JSP page files are in the **leadMgmt/** directory or a sub-directory. Optionally, the application name can be prepended to the page name. File names should be mixed case, with the first letter of the file name capitalized.

Standard Naming Convention

Where possible, you should follow the following naming convention for your JSP pages. Each JSP page should be named: PageName + [FrameName] + (".jsp" or ".jspf") where:

- PageName = Descriptive name of the page (e.g. "LeadAssignments", "Assignments", and so on)

- FrameName = Description of the frame that this page represents (for example, "Frameset", "Body", "Header", "Popup", and so on)

- Use the suffix jsp for JSP pages that are used to generate complete HTML pages or frames. Use the jspf suffix for JSP pages that are to be used as re-usable fragments of other pages.

Note the following guidelines:

- Prepend the word "Common" for files used across multiple applications.

- Use "Frameset" as the FrameName for the parent frame.

- Use "Popup" as the FrameName for popup windows and dialogs.

Examples

The following table provides some examples of JSP page names.

TABLE 8. JSP Page Names

JSP Page	Description
LeadAssignmentsFrameset.jsp	Lead Assignments main frame
LeadAssignmentsBody.jsp	Lead Assignments body frame
LeadCommonHeader.jsp	Common Leads header frame
LeadPartnerAdminList.jsp	Single frame page
LeadPartnerNotes.jsp	Single frame page
CloseLeadPopup.jsp	Popup single frame page

Resources

Many entities in the Comergent eBusiness System can have resources associated with them: examples include products, features, and questions in *C3* Advisor. To access these, you can use a scriptlet along these lines:

```
PresFeatureEntryBean feb = (PresFeatureEntryBean) fEntries.get(m);
```

```
<%
    if (feb.getResourceValue("image") != null)
    {
%>
<img src="<%= feb.getResourceValue("image") %>">
<%
    }
%>
```

The parameter for *getResourceValue()* is the name of the resource type.

Wait JSP Pages

There are times when a user initiates a process that takes a few seconds to complete, and you may want to provide a page that displays a waiting message to indicate that the Comergent eBusiness System is processing the request.

You can create a page to be displayed as follows:

1. Create a message type along these lines:

    ```
    <MessageType Name="WaitPageDisplay">
        <JSPMapping>
            ../common/WaitPageDisplay.jsp
        </JSPMapping>
        <ControllerMapping>
            com.comergent.dcm.caf.controller.ForwardController
        </ControllerMapping>
    </MessageType>
    ```

2. Create a **WaitPageDisplay.jsp** JSP page in the *debs_home***/Comergent/WEB-INF/web/en/US/common/** directory like this:

    ```
    <%@ page contentType="text/html; charset=UTF-8" language="java"
        errorPage="../error.jsp" %>
    <%@ include file="../common/cmgtinclude.jspf"%>
    <%
        String appToRun = request.getParameter("appToRun");
        String cmdToRun = request.getParameter("cmdToRun");
        String originalRequest = "true";
        if (request.getParameter("originalRequest")!=null &&
            request.getParameter("originalRequest").trim().
            equalsIgnoreCase("true"))
        {
            originalRequest = "false";
        }
    %>
    <html>
    <head>
    ```

```
                    <title>Please Wait</title>
                    <script type="text/javascript" language="JavaScript"
                        src="../js/com_Main.js"></script>
                    <script type="text/javascript" language="JavaScript">
                    <!--
                     <%@ include file="../common/pickStyleSheet.jsp"%>
                    //-->
                    </script>
                    <script>
                    <!--//
                    function init()
                    {
                        document.location="<%=link(appToRun, cmdToRun,
                            "originalRequest="+originalRequest, true)%>";
                    }
                    //-->
                    </script>
            </head>
            <body topmargin="0" leftmargin="0" marginheight="0"
                marginwidth="0" bottommargin="0" rightmargin="0"
                onLoad="javascript:init();">
                <center>
                <font color="red">
                    Please wait ...
                </font>
                </center>
            </body>
            </html>
```

3. Change the URL that executes the initial request from:

```
<%=link("application", "command")%>
```

to:

```
<%=link("application", "WaitPageDisplay",
"appToRun=application&cmdToRun=command", true)%>
```

Redirecting to Full Page Access

The Comergent eBusiness System makes use of frames as it displays pages to users of the system. In certain circumstances, a user may let their session expire, but then click a link or button in a frame. If this happens, then the Comergent eBusiness System must redirect the user to an appropriate login page, but this must be displayed as a full browser page rather than just in the frame containing the clicked element.

To handle this situation, the Comergent eBusiness System makes use of the **FullPageLoader.jsp** JSP page. This page is designed to resubmit the user's request as if it came from the top level of the browser rather than from a frame.

When the user's request is received by the server, the DispatchServlet recognizes an error condition (because the user's session has expired), and so invokes the *sendError()* method of the ComergentResponse class. This method determines if the requested message type is for a full page or for a frame, and if it is for a frame, then it invokes the *sendErrorInFrame()* method of the ComergentResponse class. This method calls the *localRedirect()* method and specifies the PageLoader message type. The PageLoader message type is associated with the **FullPageLoader.jsp** JSP page, and so the request is forwarded to this page.

This page makes use of a form whose action element is:

```
<FORM NAME="LoaderForm" ACTION="<%=link(appName)%>"
    METHOD="Post" TARGET="_top">
...
</FORM>
```

When the page is loaded by the browser, the form is automatically submitted and so the browser now regards the response from the form submission as being for the whole browser page. Hidden parameters within the form provide values for the parameters associated with the user's original request, and so the server now executes the request: typically by forwarding the request to the appropriate login page.

Online Help

This chapter presents a detailed description of the framework used to provide Online Help for the Comergent eBusiness System.

Architecture

The Online Help system is built using the JavaHelp 2.0 framework developed for Web-based applications. See the *JavaHelp 2.0 System User's Guide* for a basic overview of the framework and its API.

When a user points their browser to the Comergent eBusiness System and logs in, they are presented pages that provide them with access to all of the functionality supported by your implementation of the Comergent eBusiness System. At any time, the intention is to provide access to the Online Help through the **Help** button.

When users click the **Help** button, a Javascript function *showHelp()* opens a secondary window known as the Help window. The Help window is populated using a URL that points to a JSP page, **help.jsp**. The typical form of this URL is:

```
http://server:port/Comergent/en/US/help.jsp?id=introduc_htm_196006
```

FIGURE 7. Help Window

The Help window comprises a Navigation panel on the left and a Content panel on the right. Typically, if a user click a folder or page icon in the Navigation panel, then the appropriate page is displayed in the Content panel. The Navigation panel can provide one or more views (such as a Table of Contents view and a Search view): each view is supported by a JSP page and an XML configuration file.

The locale information of the user is reflected in the *la/CO/* part of the path of the URL. See "Localization" on page 253 for information on how to localize Online Help. The id parameter of the URL is used to determine what HTML page is used to populate the Content pane of the Help window.

When the user first accesses the Help window, a ComergentHelpBroker object is instantiated and added to their session. This object determines what content page is displayed when the user clicks a link in the Navigation panel, and keeps track of the user's actions so that if the user closes the Help window and then returns to it, the user's current view and content page is re-displayed. The ComergentHelpBroker uses a HelpSet object to determine what views and content is provided by the Online Help system, and the HelpSet object is initialized using a Help Set file: **Administration.hs**.

The content of the Help window is determined by a set of configuration files as described in "Configuration Files" on page 249.

Configuration Files

The **HelpTopicsMap.xml** configuration file maintains the basic mapping between the page that the user sees and which topc ID should be used to invoke the Online help. It provides mappings in this form:

```
<Topic>
    <Page>PricingDetail</Page>
    <URL>pricinga_htm_180685</URL>
</Topic>
```

Typically, the JSP page will specify the value of the Page element, and then a call to the HelpUtil class *getID()* method will return the value of the topic ID:

```
String helpTopicID =
    com.comergent.dcm.util.HelpUtil.getID(helpTopicPage);
```

For example:

```
String helpTopicID =
    com.comergent.dcm.util.HelpUtil.getID("PricingDetail");
```

With the exception of the **HelpTopicsMap.xml** configuration file, all of the Online Help configuration and content files are stored in the **Comergent/en/US/htdocs/ help/** directory and its sub-directories. The starting point for the Online Help system is its HelpSet file: this is an XML file that describes the help set: the mapping between IDs and content pages and what views are provided. The default HelpSet file is **Administration.hs**.

The HelpSet file can be loaded by the HelpSet class from any URL: if you use a standard HTTP URL, then you can use the URL as the "base" URL to retrieve the content files; if you use a file URL, then you must resolve the location of the content files relative to the URL used to access the **help.jsp** page.

HelpSet File

A standard HS file looks like this:

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE helpset
    PUBLIC "-//Sun Microsystems Inc.//DTD JavaHelp HelpSet Version
1.0//EN"
    "http://java.sun.com/products/javahelp/helpset_1_0.dtd">

<helpset version="1.0">
    <title>Administration</title>
    <maps>
        <homeID>introduc_htm_196006</homeID>
        <mapref location="Administration.jhm" />
```

```
        </maps>
        <view>
            <name>Contents</name>
            <label>Administration</label>
            <type>javax.help.TOCView</type>
            <data>administ.xml</data>
        </view>
        <view>
            <name>Search</name>
            <label>Search</label>
            <type>javax.help.SearchView</type>
            <data engine="com.sun.java.help.search.DefaultSearchEngine">
                JavaHelpSearch
            </data>
        </view>
</helpset>
```

The maps element specifies the location of the mapping file (in this case, **Administration.jhm**) that maps topic IDs to content HTML files. Each type element specifies the JSP page that will be used to display its view together with a data element that specifies how the data for the JSP page is to be retrieved.

Mapping File

The mapping file determines the relationship between the help IDs and the content pages:

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE map
  PUBLIC "-//Sun Microsystems Inc.//DTD JavaHelp Map Version 1.0//EN"
  "http://java.sun.com/products/javahelp/map_1_0.dtd">

<map version="1.0">
  <mapID target="defaultID" url="default.htm" />
  <mapID target="menu_closed" url="images/menu_folder_closed.gif" />
  <mapID target="menu_open" url="images/menu_folder_open.gif" />
  <mapID target="introduc_htm_196006" url="introduc.htm" />
  <mapID target="introda2_htm_211183" url="introda2.htm" />
...
</map>
```

Table of Contents File

The table of contents view is controlled by the **administ.xml** configuration file.

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE toc
  PUBLIC "-//Sun Microsystems Inc.//DTD JavaHelp TOC Version 1.0//EN"
  "http://java.sun.com/products/javahelp/toc_1_0.dtd">
```

```
<toc version="1.0" categoryclosedimage="menu_closed"
    categoryopenimage="menu_open" topicimage="topic" >
    <tocitem text="Online Help">
        <tocitem target="introduc_htm_196006"
            text="CHAPTER 1 Introduction" expand="false">
        <tocitem target="introda2_htm_211183"
            text="Managing the Sales Channel" expand="false"/>
        <tocitem target="introda3_htm_218019"
            text="Using C3 Partner.com" expand="false"/>
...
</toc>
```

The toc element contains a nested set of tocitem elements that can be organized
hierarchically to provide the Table of Contents elements. Each tocitem's target
attribute provides the target ID for its content and a text attribute whose value is
displayed in the Table of Contents view.

Search Files

The configuration files used to provide the Search functionality are contained in the
JavaHelpSearch/ sub-directory. Typically, these are generated files using a utility
such as the jhindexer utility that comes with JavaHelp 2.0.

Tag Library

A small tag library is used to help render the Navigation panel: this is provided by
the **jh.jar** library file and specified by the **jhlib.tld** tag library descriptor file.

Customizing Online Help

You can customize the Online Help for your implementation of the Comergent
eBusiness System in different ways. This section describes some of these in
ascending order of complexity.

Page Format

You can change the look-and-feel of the Online Help pages by making
modifications to the **help.css** cascading style sheet file.

Screen Shots

If you have changed the look-and-feel of the Comergent eBusiness System pages,
then you can take new screen shots of the new pages, and overwrite the
corresponding GIF files provided by the out-of-the-box Online Help. These are all
located in the **images/** sub-directory under the **help/** directory.

Content Pages

If your implementation has made changes to Comergent eBusiness System that go beyond the look-and-feel of the pages, such as adding new fields on administration pages, then you can update the corresponding HTML files and merge them into the deployed WAR file.

For example, suppose that you want to change the description of the *To Assign an Account to an Enterprise Node* task which is covered in the **account3.htm** content file. Simply use the customize target of the SDK to extract this file into your project, and edit the file as required. When you next build your project, the modified file will be built into the Online Help.

Adding Content Files

If your changes to the Comergent eBusiness System mean that you need to make greater changes to the Online Help, then you may need to create new HTML content pages. When you do so, you should bear in mind the following factors:

• Where in the overall flow of the Online Help do you want to insert the new content? New pages will need to be added to the TOC file (**administ.xml**) and you may need to fix the **Next** and **Previous** links of pages on either side so that users clicking through pages will get the right sequence of pages.

• What topic IDs should be used to point to the new content? Edit the JHM file to add these topic IDs and their corresponding references.

• Re-run the Search utility (such as jhindexer) to ensure that the new content is searchable.

Adding Views

It is possible to add new views to the Navigation panel. For example, you might want to provide access to the Index through a view, or you may want to provide a view that lists all the tasks supported by the Comergent eBusiness System or frequently asked questions.

To add a view, you must add a new view element to the **Administration.hs** HelpSet file, and provide the corresponding JSP page used to render the view. For example, you can use the following view element to add an Index view:

```
<view>
    <name>Index</name>
    <label>Index</label>
    <type>javax.help.IndexView</type>
    <data>adminind.xml</data>
```

```
</view>
```

The **adminind.xml** file would provide the list of index entries. Its standard format is:

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE index
  PUBLIC "-//Sun Microsystems Inc.//DTD JavaHelp Index Version 1.0//
EN"
  "http://java.sun.com/products/javahelp/index_1_0.dtd">

<index version="1.0">
  </indexitem>
  <indexitem text="$">
    <indexitem target="advvisu2_htm_232095"
        text="notation in models"/>
    <indexitem target="pricing6_htm_202726"
        text="used to denote shared price lists"/>
...
</index>
```

The **javax.help.Indexview.jsp** page can be used to render this view.

Bear in mind that custom views will require new JSP pages and often new tag classes to render the content of the new view. You must also consider how to create the content for the new view or how to link to existing content from the new view.

Localization

You can provide Online Help for as many as different locales as your implementation of the Comergent eBusiness System supports. To add support for a new locale, say *la_CO*, follow these steps:

1. As part of the addition of the new locale to the Comergent eBusiness System, you should have already created a new directory structure for the new locale under the **Comergent/** directory: say, **Comergent/***la***/***CO***/**, and copied over all of the static content (including the Online Help files) from the **Comergent/en/ US/** directory.

2. In the **Comergent/***la***/***CO***/htdocs/help/** directory, modify the **administ.xml** file by translating the text attribute of each tocitem.

3. Translate each HTML file in **Comergent/***la***/***CO***/htdocs/help/** for your new locale.

4. Make other changes as appropriate to the HTML pages as described in "Customizing Online Help" on page 251.

Note that the current content for the Online Help system is generated automatically using WebWorks Publisher from the FrameMaker files that are used to create the *Comergent eBusiness System Administration Guide*. If your translation project is intended to re-create the entirety of the Online Help system for a new language, then it will be more efficient to translate the source FrameMaker files, and then use WebWorks to re-generate the Online Help for the new language.

Data Services Guidelines

This chapter covers a number of topics related to the use of the data services layer of the Comergent eBusiness System:

How to Specify a Query

This section describes the use of the DsQuery and QueryHelper classes to create and modify complex SQL queries that can be used in *restore()* operations. Data beans and the IData interfaces support *restore()* methods that take a DsQuery object

as a parameter. These parameters enable you to specify a more complex SQL query to be used to retrieve data from the data source.

DsQuery classes are intended to replace the use of the com.comergent.dcm.qbe.Query class which is now deprecated. In general, you should exclusively use the DsQuery class to create complex queries.

For example, using the QueryHelper to build a DsQuery, you can specify that only data objects whose values of a particular data field match specified values should be returned, or you can specify the sort order in which data objects should be returned.

A new QueryHelper class exists to support the creation of custom search queries. It contains a number of methods that return DsQuery instances. These methods can either create new WHERE clauses, or combine existing ones. The resulting DsQuery tree is passed as a parameter to the DataBean *restore()* method.

The following methods are provided by the QueryHelper class:

- newWhereClause Methods
- newSubQuery Method
- joinWhereClauses Method
- addWhereClause Method
- addSubQuery Method
- freeQuery Method

The following methods are provided by the DsQuery class:

- debugPrintTree Method

DataBean classes provide the following method to help you debug problems with restore and persist calls.

- debugPrintSql Method

QueryHelper Methods

The QueryHelper class supports the following methods:

newWhereClause Methods

```
public static DsQuery newWhereClause(String elementName,
    int comparisonOperator,
    Object value)
```

This method creates a WHERE clause element of the form:

```
(element comparisonOperator value)
```

Using an ArrayList of Values

```
public static DsQuery newWhereClause(String elementName,
    int comparisonOperator,
    ArrayList values)
```

This method will create a WHERE clause of the form:

```
((element comparisonOperator value1) OR (element comparionsOperator
value2) OR ...)
```

This may be transformed to:

```
(element IN (value1, value2...))
```

newSubQuery Method

```
public static DsQuery newSubQuery(String elemName,
    String subqueryDataObjNm
    DsQuery subquery)
```

This method creates a WHERE clause element of the form:

```
(element IN (SELECT...))
```

It uses the first key of the DataObject as output from the subquery.

```
public static DsQuery newSubQuery(String elemName,
    String subqueryDataObjNm,
    String returnElemName,
    DsQuery subquery)
```

This method differs in that it allows the application to specify which element value to return from the sub-query Data Object.

```
public static newSubQuery(String elementName,
    String subqueryDataObjName,
    String returnElement,
    DsQuery subquery)
```

This method enables you to specify the return element name from a sub-query as opposed to the other forms of this method which return the first key element.

joinWhereClauses Method

```
public static DsQuery joinWhereClauses(DsQuery left,
    int logicalOperator,
    DsQuery right)
```

This method applies a logical operator to two previously created DsQuery elements. If you have created the following:

```
q1 = QueryHelper.newWhereClause("CartKey", DsQueryOperators.EQUALS,
    new Long(1));
q2 = QueryHelper.newWhereClause("CartName", DsQueryOperators.EQUALS,
    "My Cart");
```

then

```
query = QueryHelper.joinWhereClauses(q1, DsQueryOperators.OR, q2);
```

will generate a SQL WHERE clause of the form:

```
((CART_KEY = 1) OR (CART_NAME = 'My Cart'))
```

Joining a List of WHERE Clauses

You can provide a list of DsQuery objects and these will be joined using the specified operator as follows:

```
public static DsQuery joinWhereClauses(int logicalOperator,
    ArrayList nodes)
```

This method applies a logical operator to an ArrayList of previously created DsQuery elements. If you have created the following:

```
q1 = QueryHelper.newWhereClause("CartKey", DsQueryOperators.EQUALS,
    new Long(1));
q2 = QueryHelper.newWhereClause("CartKey", DsQueryOperators.EQUALS,
    new Long(2));
q3 = QueryHelper.newWhereClause("CartKey", DsQueryOperators.EQUALS,
    new Long(3));
nodeArrayList.add(q1);
nodeArrayList.add(q2);
nodeArrayList.add(q3);
```

then

```
query = QueryHelper.joinWhereClauses(DsQueryOperators.OR,
    nodeArrayList);
```

will generate a SQL WHERE clause of the form:

```
((CART_KEY = 1) OR ((CART_KEY = 2) OR (CART_KEY = 3)))
```

addWhereClause Method

You can add to WHERE clauses as follows:

```
public static DsQuery addWhereClause(DsQuery query,
    int logicalOperator,
    String elementName,
    int comparisonOperator,
    Object value)
```

This is a helper method, equivalent to:

```
q1 = QueryHelper.newWhereClause(oldQuery, element,
    comparisonOperator, value);
query = QueryHelper.joinWhereClauses(oldQuery, logicalOperator, q1);
```

This method adds a WHERE clause element of the form:

```
(oldQuery) logicalOperator (element comparisonOperator value)
```

addSubQuery Method

```
public static DsQuery addSubquery(DsQuery query,
    int logicalOperator,
    String element,
    String subqueryDataObjNm,
    DsQuery subquery)
```

This is a helper method, equivalent to:

```
q1 = QueryHelper.newSubquery(element, subqueryDataObjName, subquery);
q2 = QueryHelper.joinWhereClauses(oldQuery, logicalOperator, q1);
```

This method adds a subquery of the form:

```
(oldQuery) logicalOperator (element IN (SELECT...))
```

It uses the first key of the DataObject as output from the subquery.

```
public static DsQuery addSubquery(DsQuery query, int logicalOperator,
    String elementName,
    String subqueryDataObjName,
    String returnElement,
    DsQuery subquery)
```

This form of the method enables you to specify the returned element as opposed to the first key element.

freeQuery Method

```
public void freeQuery(DsQuery query)
```

This method releases all nodes of a DsQuery tree to the pool for reuse.

Attention:	It is important that the application invoke this method when the query is no longer needed.

DsQuery Methods

The following methods are provided by the DsQuery class.

debugPrintTree Method

```
public void debugPrintTree(String heading, PrintStream stream)
```

This method is for debugging. It prints a formatted dump of the DsQuery tree. The heading parameter is a mandatory text string that will prefix all printed lines.

DataBean Methods

The following method is available on the DataBean class.

debugPrintSql Method

```
public void debugPrintSql(DataContext context,
    DsQuery query,
    String heading,
    PrintStream stream)
```

This method is for debugging. It generates the resulting SQL query and writes a formatted version to the PrintStream. The heading parameter is a mandatory text string that will prefix all printed lines. This method has to be on the DataBean in order to have appropriate context to generate the SQL.

Using LIKE Calls

There is no explicit LIKE operator. You can use EQUALS or EQUALS_IGNORE_CASE and transform to the equivalent of a LIKE clause using wild card characters.

For example, suppose that you have a case where the data in the Knowledgebase is in mixed case, and the condition you want in the WHERE clause is 'toupper(PartnerName) like "%ABC%"'. In this case, you use the EQUALS_IGNORE_CASE operator: for example:

```
DsQuery temp_DsQuery =
    QueryHelper.newWhereClause("PartnerName",
    DsQueryOperators.EQUALS_IGNORE_CASE, "*ABC*");
```

If you use "*", then you can also use other wildcard characters that the database server will understand: such as "_" and "%" with Oracle and "_" with SQL Server.

Note that using * to the left of a String in the value parameter to indicate that you want to match occurrences of a string with any preceding string can significantly impact the performance of the query and hence of the application.

Examples

Example 1: Simple Search Query

Search for Partner Name containing "micro" in any case.

```
DataContext context = new DataContext();
PartnerBean partner = new PartnerBean();
```

```
DsQuery qry = QueryHelper.newWhereClause("PartnerName",
    DsQueryOperators.EQUALS_IGNORE_CASE, "*micro*");
partner.restore(context, qry);
qry.free();
```

During the restore operation, this will generate:

```
SELECT CMGT_PARTNERS.PARTNER_KEY…
    FROM CMGT_PARTNERS
    WHERE UPPER(CMGT_PARTNERS.PARTNER_NAME) LIKE '%MICRO%';
```

Example 2: Search Query with Three Values

Search for Partner in Territory 1, 2, or 3.

```
DataContext context = new DataContext();
Integer i1 = new Integer(1);
Integer i2 = new Integer(2);
Integer i3 = new Integer(3);
ArrayList values = new ArrayList (3);
values.add(i1);
values.add(i2);
values.add(i3);
PartnerBean partner = new PartnerBean();
DsQuery qry = QueryHelper.newWhereClause("TerritoryKey",
    DsQueryOperators.EQUALS, values);
partner.restore(context, qry);
qry.free();
```

During the restore, this will generate:

```
SELECT CMGT_PARTNERS.PARTNER_KEY…
    FROM CMGT_PARTNERS
    WHERE CMGT_PARTNERS.TERRITORY_KEY IN (1, 2, 3);
```

Example 3: Sub-query with Two Values

Search for Partners in State "CA" and in City "San Francisco".

```
DataContext context = new DataContext();
DsQuery query;
PartnerBean partner = new PartnerBean();
// match the State
query = QueryHelper.newWhereClause("State",
    DsQueryOperators.EQUALS, "CA");
    // Add a match on City
query = QueryHelper.addWhereClause (query, DsQueryOperators.AND,
    "City", DsQueryOperators.EQUALS, "San Francisco");
// Convert it to a subquery of PartnerAddress
query = QueryHelper.addSubQuery("AddressKey",
    "PartnerAddress", query);
partner.restore(context, query);
```

```
query.free();
```

During the restore operation, this will generate:

```
SELECT CMGT_PARTNERS.PARTNER_KEY…
    FROM CMGT_PARTNERS
    WHERE CMGT_PARTNERS.ADDRESS_KEY IN
    (SELECT CMGT_ADDRESSES.ADDRESS_KEY
    FROM CMGT_ADDRESSES
    WHERE CMGT_ADDRESSES.STATE = 'CA'
    AND CMGT_ADDRESSES.CITY = 'San Francisco');
```

Example 4: INTERSECT with Two Values

Find products for two different features.

Attention:	The INTERSECT operator can only be applied to subqueries, and those subqueries must return identical result columns.

```
DataContext context = new DataContext();
DsQuery q1, q2, finalQuery;
ProductBean product = new ProductBean();
// specify the first Feature and convert it to a subquery
q1 = QueryHelper.newWhereClause("FeatureKey",
    DsQueryOperators.EQUALS, fKey1);
q1 = QueryHelper.newSubQuery("ProductId",
    "ProductXFeature", q1);
// specify the second Feature and convert it to a subquery
q2 = QueryHelper.newWhereClause("FeatureKey",
    DsQueryOperators.EQUALS, fKey2);
q2 = partner.newSubQuery("ProductId",
    "ProductXFeature", q2);
    // INTERSECT the 2 subqueries
finalQuery = QueryHelper.joinWhereClauses(q1,
    DsQueryOperators.INTERSECT, q2);
product.restore(context, query);
finalQuery.free();
```

During the restore, this will generate:

```
SELECT CMGT_PRODUCTS.PRODUCT_ID…
    FROM CMGT_PRODUCTS
    WHERE CMGT_PRODUCTS.PRODUCT_ID IN
    INTERSECT
    ((SELECT CMGT_PRODUCT_X_FEATURE.PRODUCT_ID
    FROM CMGT_PRODUCT_X_FEATURE
    WHERE CMGT_PRODUCT_X_FEATURE.FEATURE_KEY = ?)
    INTERSECT
    ((SELECT CMGT_PRODUCT_X_FEATURE.PRODUCT_ID
    FROM CMGT_PRODUCT_X_FEATURE
```

```
WHERE CMGT_PRODUCT_X_FEATURE.FEATURE_KEY = ?)
```

Example 5: Using Two Sub-queries

Assume that we have a Partner data object which has an Address child data object and a Vertical child data object. If we want to find all Partners which have Partner.type = "XYZ" and Address.city = "Boston" and Vertical.name = "Government":

```
DataContext context = new DataContext();
DsQuery query, subq1, subq2;
PartnerBean partner = new PartnerBean();
// Specify the Partner.type
q1 = QueryHelper.newWhereClause("Type",
    DsQueryOperators.EQUALS, "XYZ");
// Specify the Address.city
subq1 = QueryHelper.newWhereClause("City",
    DsQueryOperators.EQUALS, "Boston");
// Tell it to use the "Address" DataObject for the subquery
subq1 = QueryHelper.newSubQuery("AddressKey",
    "Address", subq1);
// Join the 1st where clause with the subquery
query = QueryHelper.joinWhereClauses(query,
    DsQueryOperators.AND, subq1);
// Specify the Vertical.Name
subq2 = QueryHelper.newWhereClause("Name",
    DsQueryOperators.EQUALS, "Government");
// Tell it to use the "Vertical" DataObject for the subquery
subq2 = QueryHelper.newSubQuery("VerticalKey",
    "Vertical", subq1);
// Add this new subquery to the existing one.
query = QueryHelper.joinWhereClauses(query,
    DsQueryOperators.AND, subq2);
partner.restore(context, query);
finalQuery.free();
```

During the restore, this generates:

```
SELECT CMGT_PARTNERS.PARTNER_KEY…
    FROM CMGT_PARTNERS
    WHERE CMGT_PARTNERS.TYPE = 'XYZ'
    AND CMGT_PARTNERS.ADDRESS_KEY IN
    (SELECT CMGT_ADDRESSES.ADDRESS_KEY
    FROM CMGT_ADDRESSES
    WHERE CMGT_ADDRESSES.CITY = 'Boston')
    (SELECT CMGT_VERTICALS.VERTICAL_KEY
    FROM CMGT_VERTICALS
    WHERE CMGT_VERTICALS.NAME = 'Government');
```

Note: This works even if the Partner Business Object does not have Vertical or Address data objects included as children. The new query mechanism is capable of generating subqueries from DataObjects that are not referenced in the data object.

Example 6: Subquery Using the Child of a Child

Assume that you have an Order data object which has a LineItem child data object which itself has a SerialItem child data object. If we want to find all Orders which have SerialItem.type = "XYZ", then code along these lines will work:

```
DataContext context = new DataContext();
DsQuery query;
OrderBean order = new OrderBean();
// Specify the Partner.type
query = QueryHelper.newWhereClause("Type", DsQueryOperators.EQUALS,
    "XYZ");
// Tell it to use the new WHERE clause in a subquery of the
// "Vertical" DataObject
query = QueryHelper.newSubQuery("LineItemKey", "SerialItem", subq1);
// Tell it to use the subquery results in a subquery of the
// "LineItem" DataObject
query = QueryHelper.newSubQuery("OrderKey", "LineItem", query);
order.restore(context, query);
```

During the restore, this will generate:

```
SELECT CMGT_ORDERS.ORDER_KEY…
FROM CMGT_ORDERS
WHERE CMGT_ORDERS.ORDER_KEY IN
    (SELECT CMGT_ORDER_LINES.ORDER_KEY
    FROM CMGT_ORDER_LINES
    WHERE CMGT_ORDER_LINES.LINE_ITEM_KEY IN
        (SELECT CMGT_ORDER_SERIAL_ITEMS.LINE_ITEM_KEY
        FROM CMGT_ORDER_SERIAL_ITEMS
        WHERE CMGT_ORDER_SERIAL_ITEMS.TYPE = 'XYZ'));
```

How to Specify Sort Order

The following methods are available on the DataBean class:

- addSort Method

- insertSort Method

- clearSort Method

They provide a simple interface for adding sort criteria, or modifying existing criteria. The sort criteria are preserved following a restore operation. This allows an existing sort to be qualified by adding or inserting additional criteria. If this is not desired, then the clearSort method can be used to drop the existing criteria.

addSort Method

```
public void addSort(String sortBy, boolean ascending)
```

This method appends to the sort order.

insertSort Method

```
public void insertSort(String sortBy, boolean ascending)
```

This method inserts into the sort order. This is setting the high order sort entry.

clearSort Method

```
public void clearSort()
```

This method clears all current sort settings.

Example 1: Sort on One Element Ascending

```
AddressListBean addrList = new AddressListBean();
AddrList.addSort("State", DsConstants.ASCENDING);
AddrList.restore(contetx, null);
```

Example 2: Sort on One Element Ascending, One Element Descending

```
AddressListBean addrList = new AddressListBean();
AddrList.addSort("State", DsConstants.ASCENDING);
AddrList.addSort("City", DsConstants.DESCENDING);
AddrList.restore(contetx, null);
```

Example 3: Add New High Order Sort

```
AddressListBean addrList = new AddressListBean();
AddrList.addSort("City", DsConstants.DESCENDING);
AddressList.restore();
// Now modify it to first sort on "State"
AddrList.insertSort("State", DsConstants.ASCENDING);
AddrList.restore(contetx, null);
```

Query Constants

DsQueryOperators

The DsQueryOperators interface defines string constants for both logical and comparison operators specified in a DsQuery.

```
package com.comergent.dcm.dataservices;

public interface DsQueryOperators
{
    public static final int NOT_SET   = -1;
    public static final int AND        = 0;
    public static final int OR         = 1;
```

```
     public static final int INTERSECT    = 2;
     public static final int MAX_LOGICAL_OP = 2;
     public static final int MIN_COMPARISON_OP = 3;
     public static final int EQUALS    = 3;
     public static final int EQUALS_IGNORE_CASE = 4;
     public static final int NOT_EQUALS = 5;
     public static final int GT      = 6;
     public static final int GE      = 7;
     public static final int LT      = 8;
     public static final int LE      = 9;
     public static final int LIKE    = 10;
     public static final int CONTAINS  = 11;
     public static final int SUBQUERY  = 12;
     public static final int NOT_IN_SUBQUERY  = 13;
     public static final int LEFT_OUTER_EQUALS = 14;
     public static final int INVALID      = 15;
}
```

DsConstants

The DsConstants interface defines string constants for use by Data Services clients. This class extends DsQueryOperators in order to provide a single source, but not break existing code.

```
package com.comergent.dcm.dataservices;

public interface DsConstants extends DsQueryOperators
{
     public static final boolean ASCENDING = true;
     public static final boolean DESCENDING = false;
     public static final int NO_LIMIT = -1;
     public static final String NULL = "NULL";
}
```

Using UpdateHelper and DsUpdate

There are times when you need to update multiple data objects with the same change. The DsUpdate class provides a mechanism to do this as described in this section. In general, you should use the UpdateHelper class to work with DsUpdate because it provides a number of methods that help construct DsUpdate objects.

You use an instance of the DsUpdate class in conjunction with the DataContext class. You use the *addFieldUpdate()* method of a DsUpdate object to specify which field should be updated and with what value. Each field to be updated is represented by a DsUpdateField object. When *persist()* is invoked on a data bean, and the associated DataContext has a specified DsUpdate object, then the fields of the databean are updated with the values as specified by the DsUpdateField objects.

The UpdateHelper class provides three methods that return IDsUpdate interfaces. These methods can be used to request an Update, Delete (logical delete), or Erase (database delete) operation.

The following steps are required to issue a request:

1. Use the QueryHelper to create a DsQuery that specifies the WHERE clause for the request.

2. Obtain an instance of the DataBean for which the update operation is to be performed.

3. Use one of the methods on the UpdateHelper to specify the desired type of operation and obtain an IDsUpdate interface.

4. Optionally specify additional SET statements (for updates only) using the IDsUpdate interface.

5. Invoke *persist()* on the IDsUpdate.

Note the following:

•   All UpdateHelper methods are static. The UpdateHelper does not need to be directly instantiated.

•   The IDsUpdate interfaces cannot currently be enlisted in a Transaction or ActiveTransaction.

For a detailed examples of how to create complex WHERE clauses, please refer to "DsQuery Methods" on page 259.

UpdateHelper Methods

newDelete Method

```
public static IDsUpdate newDelete(IData bean, IDsQuery query)
```

This method creates a MARK INACTIVE request of the form:

```
UPDATE TABLE SET ACTIVE_FLAG = 'N' WHERE where clause from query
```

newErase Method

```
public static IDsUpdate newErase(IData bean, IDsQuery query)
```

This method creates a DELETE request of the form:

```
DELETE FROM TABLE WHERE where clause from query
```

newUpdate Method

```
public static IDsUpdate newUpdate(IData bean, String elementName,
```

```
    int updateOperator, Object value, DsQuery query)
```

This method will create an UPDATE request of the form:

```
UPDATE TABLE SET COLUMN = COLUMN UPDATEOPERATOR VALUE WHERE where
clause from query
```
Or for an assignment operator:

```
UPDATE TABLE SET COLUMN = VALUE WHERE where clause from query
```

IDsUpdate Methods

addFieldUpdate Method
```
public void addFieldUpdate(String elementName, int updateOperator,
    Object value)
```
This method adds a "SET" clause to the request. This must be invoked once for each additional Element to be updated.

persist Method
```
public void persist() throws ICCException
```

This method persists and commits the request.

depugPrint Method
```
public void debugPrint(String heading, PrintStream stream)
```

This method performs a formatted dump of the internal DsUpdate and DsQuery structures.

debugPrintSql Method
```
public void debugPrintSql(String heading, PrintStream stream)
    throws ICCException
```

This method dumps the SQL that would be generated by the specified DsUpdate request.

Operators

You can use the following operators in the update operation:

• ASSIGN

• ADD

• SUBTRACT

• MULTIPLY

• DIVIDE

Example

Suppose that you want to update all partner profiles for partners whose name includes the string "micro" by setting their partner type to "2".

```
PartnerBean partner = new PartnerBean();
DsQuery query = QueryHelper.newWhereClause("PartnerName",
    DsQueryOperators.EQUALS_IGNORE_CASE,
    "*micro*");
IDsUpdate update = UpdateHelper.newUpdate(partner, PartnerType,
    DsUpdateOperators.ASSIGN, new Long(2), query);
update.persist();
```

During the persist operation, this will generate the following SQL:

```
UPDATE CMGT_PARTNERS
SET PARTNER_TYPE = 2
WHERE UPPER(PARTNER_NAME) LIKE '%MICRO%';
```

Oracle Hints

If your Knowledgebase is running on an Oracle database server, then you can take advantage of Oracle Hints to improve the performance of your system. This section describes how to use Oracle Hints. It covers:

- What are Oracle Hints?

- What support is available for Oracle Hints?

- When should I use Oracle Hints?

- How do I specify an Oracle Hint for the primary query?

- How do I specify an Oracle Hint for a sub-query?

- What is the Oracle Hints syntax?

What are Oracle Hints?

The Oracle database server provides a "Hints" mechanism that is used to provide information to the Oracle Query Optimizer on how best to execute a SQL request. These "Hints" are embedded directly in the SQL to be executed.

What support is available for Oracle Hints?

Beginning in Release 6.3, the Comergent eBusiness System has added support for "Hints" in both the primary query and in sub-queries specified using the DsQuery mechanism.

When should I use Oracle Hints?

Oracle Hints should be used if there are performance issues with a specific database query. The Oracle Query Optimizer can be used to evaluate execution plans and estimated costs for a query. The query execution plan can then be fine-tuned using the hints mechanism and the appropriate hint can then be provided at execution time.

How do I specify an Oracle Hint for the primary query?

The DataBean and IData interface now provide the following method to specify an Oracle Hint:

```
public void setQueryHint(String queryHint)
```

This method is available on all generated Bean classes and all generated IData interfaces. It adds the specified queryHint to the SQL statement immediately following the SELECT keyword.

How do I specify an Oracle Hint for a sub-query?

The Comergent eBusiness System added an overload of the *newSubQuery()* method to the QueryHelper class that includes a Hint parameter. The full method signature is:

```
public static DsQuery newSubQuery(String elementName,
    String subqueryDataObjName, String returnElement,
    DsQuery subquery, boolean showInactive, String queryHint)
```

The queryHint is inserted immediately following the SELECT keyword for the specified subquery.

What is the Oracle Hints syntax?

For detailed information on the Oracle Hints syntax, please refer to the version-specific Oracle documentation. Since the Comergent eBusiness System directly inserts the provided string into the SELECT statement, we can support all possible hints.

Stored Procedures

This section describes the use of stored procedures in data objects.

* What support is available for Stored Procedures in Release 6.0?

* What Stored Procedure support has been added in Release 6.3?

* What are the limitations on Stored Procedure support?

- How do I map a data object to a database stored procedure?

- Examples

  - Sample DataObject using Output Parameters

  - Sample DataObject using Result Parameters

  - Sample Oracle Stored Procedure returning a Result Set

What support is available for Stored Procedures in Release 6.0?

- Only Oracle stored procedures are supported.

- Input and Output parameters are supported.

- IN/OUT parameters and result sets are not supported.

What Stored Procedure support has been added in Release 6.3?

In Release 6.3 and later, the Comergent eBusiness System has added support for the following:

- Microsoft SQL Server stored procedures are now supported.

- The Comergent eBusiness System supports returning result sets from stored procedures.

What are the limitations on Stored Procedure support?

The Comergent eBusiness System does not support IN/OUT parameters. These are not available in all supported database servers. The Comergent eBusiness System does not support the use of output parameters and result sets within the same data object. It is possible for a data object to specify a child data object that also references a stored procedure. The Comergent eBusiness System does not support Collection-based parameters such as Oracle's support of Vectors of values.

Database stored procedures do not support variable parameter lists. This imposes some limitations on qualifying what data will be returned by a stored procedure. Data objects can only be mapped to a single database stored procedure. This means that it is not possible to use one stored procedure for retrieval of data and a different stored procedure for persistence. Non-result set based stored procedures can be written so that a parameter indicates the type of operation to be performed, but this significantly complicates the logic of the stored procedure.

Due to the limitations of database stored procedures, it is not possible to persist changes to data retrieved using the stored procedure result set mechanism.

Database stored procedures do not support variable parameter lists. This imposes some limitations on qualifying what data will be returned by a stored procedure.

How do I map a data object to a database stored procedure?

Tying a data object to a stored procedure can be accomplished as follows. In the XML data object definition file for the data object:

1. Specify the stored procedure name in the ExternalName attribute of the DataObject. For Oracle, the stored procedure name should be prefixed by its package name.

2. Specify the SourceType attribute for the data object and assign it a value of "2".

3. Specify all input parameters as data fields with a ParameterType attribute value of "IN".

4. Ensure all input parameters are also specified as key fields.

5. Specify all output parameters as data fields with a ParameterType attribute value of "OUT". You must also specify an ExternalFieldName attribute for these fields. This is not used and so you can set any value for the attribute.

6. Specify all result parameters as data fields with a ParameterType attribute value of "RESULT". Note that the result set is returned directly by the procedure call. The result parameters correspond to the columns that comprise the result set.

Examples

Sample DataObject using Output Parameters

```
<DataObject Name="SampleProcedure"
    ExternalName="SAMPLE_PROCEDURE.SEARCH_BY_DESC"
    Access="RWID" Ordinality="n" ObjectType="JDBC" SourceType="2"
    Version="6.3">
    <KeyFields>
        <KeyField Name="ProductID"/>
    </KeyFields>
    <DataFieldList>
        <DataField Name="ProductID" Writable="y" Mandatory="n"
            ParameterType="IN"/>
        <DataField Name="Name" Writable="n" Mandatory="n"
            ParameterType="OUT" ExternalFieldName="placeholder" />
        <DataField Name="Description" Writable="n" Mandatory="n"
            ParameterType="OUT" ExternalFieldName="placeholder" />
    </DataFieldList>
</DataObject>
```

Sample DataObject using Result Parameters

```
<?xml version="1.0"?>
<DataObject Name="SampleProcedure"
    ExternalName="SAMPLE_PROCEDURE.SEARCH_BY_DESC" Access="RWID"
    Ordinality="n" ObjectType="JDBC" SourceType="2" Version="6.3">
    <KeyFields>
        <KeyField ExternalName="DESCRIPTION"/>
        <KeyField ExternalName="ROWNUM"/>
    </KeyFields>
    <DataFieldList>
        <DataField Name="SearchString" Writable="y" Mandatory="n"
            ParameterType="IN"/>
        <DataField Name="MaxRowCount" Writable="y" Mandatory="n"
            ParameterType="IN"/>
        <DataField Name="ProductID" Writable="n" Mandatory="n"
            ParameterType="RESULT"/>
        <DataField Name="Name" Writable="n" Mandatory="n"
            ParameterType="RESULT"/>
        <DataField Name="Description" Writable="n" Mandatory="n"
            ParameterType="RESULT"/>
    </DataFieldList>
</DataObject>
```

Sample Oracle Stored Procedure returning a Result Set

```
CREATE OR REPLACE PACKAGE
sample_procedure AS
TYPE search_rec IS RECORD(
sku_name cmgt_product.sku_name%TYPE,
NAME cmgt_product_locale.NAME%TYPE,
DESCRIPTION cmgt_product_locale.DESCRIPTION%TYPE);
TYPE search_result
IS REF CURSOR RETURN search_rec;
FUNCTION
search_by_desc(desc_str in varchar2, max_row_count in number) RETURN
search_result;
END sample_procedure;
/
show errors

CREATE OR REPLACE PACKAGE BODY
sample_procedure AS
FUNCTION search_by_desc(desc_str in varchar2, max_row_count in number)
RETURN search_result IS
rc search_result;
BEGIN
OPEN rc for
SELECT cmgt_product.sku_name,
cmgt_product_locale.NAME,
```

```
cmgt_product_locale.DESCRIPTION
FROM cmgt_product, cmgt_product_locale
WHERE rownum < max_row_count
AND UPPER(cmgt_product_locale.DESCRIPTION)
LIKE desc_str
AND cmgt_product_locale.sku_name
= cmgt_product.sku_name(+)
AND cmgt_product_locale.locale
= 'en_US'
ORDER BY cmgt_product.sku_name;
RETURN rc;
END;
END sample_procedure;
/
show errors
```

Pagination

This section describes how you can use the built-in pagination capabilities to handle large lists of data objects. An example of its usage is provided in "Pagination" on page 194.

How do I get a Paginated Result Set?

When a DataListBean *restore()* is invoked, the first page of results is returned immediately. If there is more than a single page of data, then page files will be created asynchronously.

How do I tell if I have more than one page of results?

The DataListBean *moreResults()* method will return true.

How do I tell if there are more pages in the page set?

The DataListBean *morePages()* method will return true.

What happens if I ask for a page that does not exist?

If the page is still being created, then the DataListBean *getNextPage()* method will wait. If the pagination set has been completely built and the requested page does not exist, then *getNextPage()* will return false.

If I make changes, then will they appear in the page files?

If you perform a *persist()* on a paginated DataListBean, then the changes are first persisted to the database, and only then is the relevant page file rewritten.

How do I control the number of results per page?

By default, the number of results per page is controlled by the NumPerCachePage element in the **DataServices.xml** property file. A value of "-1" indicates no limit. This can be overridden by specifying a DataContext during restore.

Is there a limit on the number of page files?

By default, the maximum number of results is controlled by the MaxResults element in the **DataServices.xml** property file. A value of "-1" indicates no limit. This can be overridden by specifying a DataContext during restore.

When are the page files deleted?

Page files can be explicitly released by invoking DataListBean *freeCache()* method. By default, they will be released as soon as the session terminates. They will also be reused by a subsequent restore request from the same session on the same DataListBean.

Can I have multiple paged result sets in the same session?

Normal behavior is to support one result page set per DataListBean type per Session. A subsequent attempt to restore the same DataListBean would normally overwrite a previously created one.

It is possible to create multiple page sets by naming the page set using a DataContext during the restore. If the page set is not named, then it will overwrite an existing page set for that data bean and session combination.

Can I control where the page files are written?

Yes, the directory path for page files is controlled by the rsCachePath element in the **DataServices.xml** property file.

Performance Optimization

This section describes factors that come into play when considering performance issues associated with the data services layer of the Comergent eBusiness System. Bear in mind that these are guidelines and that they should be complemented by a thorough understanding of the database server in use.

Optimizing Ad Hoc Queries

The new QueryHelper class exists to support the creation of ad hoc search queries. It contains a number of methods that return DsQuery instances. These methods can either create new WHERE clauses, or combine existing ones. The resulting DsQuery tree is passed as a parameter to the DataBean restore method.

The order in which selection criteria are added using the QueryHelper directly translates to how the resulting WHERE clauses are generated. RDBMS provide proprietary tools that can be used to evaluate the execution cost of queries. These tools can be used to determine the best ordering of WHERE clauses. There may be additional efficiencies to be gained through the use of sub-queries and the ordering of these sub-queries. The optimal WHERE clause for a specific request is dependent on:

1. The RDBMS

2. The available indices (and the type of each index)

3. The amount of data in each table to be joined

4. The use of RDBMS cost-based vs. syntax-based query optimizers

5. How recently database statistics have been generated (where appropriate for the specific RDBMS)

6. The uniqueness of the selection criteria

7. Whether the RDBMS has cached any of the relevant tables or indices

8. The on-disk distribution of database tables and indices

Due to the number of factors that influence query performance, query tuning recommendations should be obtained from your proprietary database documentation. Query tuning is typically a combination of WHERE clause and index optimization.

While we have attempted to create appropriate indexing for common queries issued by the Comergent software, customizations may involve additional selection criteria that will perform more efficiently with additional database indexing. We strongly recommend that the DsQuery mechanism be used whenever non-key queries are issued.

Optimizing Data Retrieval Sizes

The DataContext mechanism can be used to set the number of records per page and the maximum number of records to retrieve.

If a fast initial page display is required, then setting a smaller number of records per page will result in a faster initial response. Remaining data will be automatically retrieved using a background thread.

To limit overall processing overhead, the maximum number of records can be limited. This is especially appropriate in areas such as retrieving a list of products. It is unlikely that an end user will want to scroll through several thousand products.

A more reasonable approach is to limit the number of products to 100 or even 50, and then to allow the user to specify additional selection criteria.

Left-Outer and Equi-Joins

Our default join mechanism is the left-outer join. This mechanism provides the behavior expected by most application developers. We have discovered several cases where an RDBMS generated a sub-optimal execution plan for some queries that used left-outer joins. If a specific query is having performance problems, then you may want to determine if there are left-outer joins that can be converted to equi-joins. For additional information, please refer to"Join Types" on page 278.

Reference and Child Data Objects

Reference DataObjects are intended for 1-to-1 relationships. Child DataObjects are typically used for 1-to-many relationships. There are some circumstances in which you may wish to use a Child DataObject to represent a 1-to-1 relationship.

Remember that Child DataObjects use a lazy link mechanism. This does result in a separate query being issued for the child data, but it also means that the data is not retrieved until it is directly referenced. For data that is infrequently referenced the Child DataObject may be more appropriate.

Using Distinct Tables for Customer Extensions

There has been some concern raised regarding the concept of storing custom data in separate tables. In most cases, this new data has a 1-to-1 relationship to existing Comergent tables. This allows the use of Reference DataObjects resulting in a single query to obtain the combined data. In addition, it should be possible to access the custom table by indexed primary key resulting in a very low retrieval cost.

While adding an additional table to a query will always have an impact on the retrieval cost, we feel that cost should be minimal in most cases. The key benefit of using distinct tables for custom data is the ease of upgrade to new releases of the Comergent software and schema. By using a separate table combined with our XML schema inheritance, the customer can be isolated from both database and XML schema changes. The Object Manager also makes these customizations transparent to the Comergent software. This combination of factors dramatically simplifies the upgrade process.

Using Stored Procedures

In some extreme cases, there may be a significant performance advantage if RDBMS stored procedures are used in place of a dynamic query.

Stored procedures have the following advantages:

1. The database can normally cache the SQL execution plan.

2. There may be less network traffic required to execute the stored procedure than for an equivalent SQL statement.

3. It may be possible to execute more complex SQL statements.

Stored procedures have the following disadvantages:

1. There is additional maintenance overhead for each stored procedure that is created. This may result in additional work during product upgrades.

2. Stored procedures do not easily handle variable parameter input. This can be accomplished but it tends to result in an extremely complex and possibly inefficient stored procedure.

3. Currently, data beans can only be mapped to a single stored procedure. This means that it is not possible to map to one procedure for data retrieval and a second procedure for update.

In general, we suggest that stored procedures only be considered if they can provide a significant performance benefit. In most cases, with appropriate indexing, the DsQuery mechanism is capable of providing very efficient query execution. For further information, please refer to "Stored Procedures" on page 270.

Oracle Hints

Release 6.3 added support for Oracle hints as described in "Oracle Hints" on page 269. We have found very few instances where this has provided performance improvements that could not be accomplished by restructuring the DsQuery. However, this mechanism is available if query analysis shows it can provide a significant benefit.

Join Types

What is an Equi-Join?

An equi-join is a table join that requires corresponding values to exist in the joined tables. For example given the following two tables:

TABLE 9. Table T1

C1	C2
A	1
A	2

TABLE 9. Table T1 (Continued)

C1	C2
A	3
B	2

TABLE 10. Table T2

C1	C2
1	X
3	Y
4	X

The following query:

```
SELECT * FROM T1, T2 WHERE T1.C2 = T2.C1
```

will return:

TABLE 11. Result Set Table

T1.C1	T1.C2	T2.C1	T2.C2
A	1	1	X
A	3	3	Y

Notice that there are no values returned for T1.C2 = 2 or T1.C1 = B. This is due to the mandatory join requirement.

What is a Left-Outer Join?

A left-outer join will return all rows from the left-side table even when the right-side table in a join does not contain any rows that match the join criteria.

If the above query is revised to use left-outer join syntax as follows:

```
SQL Server: SELECT * FROM T1 LEFT OUTER JOIN T2 ON T1.C2 = T2.C1
Oracle Syntax: SELECT * FROM T1, T2 WHERE T1.C2 = T2.C1 (+)
```

will return:

TABLE 12. Result Set Table

T1.C1	T1.C2	T2.C1	T2.C2
A	1	1	X
A	2		

TABLE 12. Result Set Table (Continued)

T1.C1	T1.C2	T2.C1	T2.C2
A	3	3	Y
B	2		

What is a Right-Outer Join?

A right-outer join is similar to the left-outer join, except that it returns rows in the right-side table even if there are no matching rows in the left-side table.

If the above query is revised to use right-outer join syntax as follows:

```
SQL Server: SELECT * FROM T1 RIGHT OUTER JOIN T2 ON T1.C2 = T2.C1
Oracle Syntax: SELECT * FROM T1, T2 WHERE T1.C2 (+) = T2.C1
```

will return::

TABLE 13. Result Set Table

T1.C1	T1.C2	T2.C1	T2.C2
A	1	1	X
A	1	3	Y
		4	Z

What is a Cross Join?

A cross-join is the cross product of the rows in both tables. It can be achieved by not specifying any selection criteria. This type of join is rarely of any practical use.

If the above query is revised to use cross join syntax as follows:

```
SELECT * FROM T1, T2
```

will return:

TABLE 14. Result Set Table

T1.C1	T1.C2	T2.C1	T2.C2
A	1	1	X
A	1	3	Y
A	1	4	Z
A	2	1	X
A	2	3	Y

T1.C1	T1.C2	T2.C1	T2.C2
A	2	4	Z
A	3	1	X
A	3	3	Y
A	3	4	Z
A	2	1	X
A	2	3	Y
A	2	4	Z

What is our Default Join Mechanism?

By default, the Comergent eBusiness System uses a left-outer join because this is typically what an application wants, and because this mechanism can perform lazy evaluation of joins for child data objects.

Which Joins do we Support?

We support the left-outer join and the equi-join.

How do I tell the Data Services Layer to use an Equi-Join?

An equi-join is specified by adding the following attribute to the Relationship element in the data object definition XML file:

```
JoinOperator="EQUI"
```

Remember that an equi-join will not return rows unless a match is found for the join criteria.

Transactions

This section describes the support provided by the Comergent eBusiness System for transactions: database actions that span one or more atomic operations. In general, you use the Transaction class to manage situations in which several data objects must be persisted together, and if one fails, then they should all fail.

Default Transaction Support

All business objects and generated data beans that reference one or more database tables provide implicit transaction integrity. Any *persist()* operation performed on a business object or data bean ensures that all resulting INSERT, UPDATE, and DELETE requests occur within the context of a single physical database

transaction. If any of the resulting operations fail, then all requests are rolled back. Note that if a data bean has children, then these are also persisted within the same transaction.

Support Using the Transaction Class

In certain circumstances it may be necessary to persist multiple distinct business objects or data beans within the same transaction. This can be accomplished through the use of the Transaction class. See "Transaction Class Methods" on page 282 for details.

When you use the Transaction class, data beans are enlisted into a Transaction object. When the *commit()* method is invoked on the Transaction object, *persist()* operation are invoked on the individual data beans:

• If any of the *persist()* operations fail, then they are all rolled back.

• If all of the *persist()* operations succeed, then they are all committed together.

Support Using the ActiveTransaction Class

In extraordinary circumstances it may be necessary to ensure that the persistence of multiple data beans occur within the same transaction, and that these changes are persisted immediately to allow restore requests within the same transaction to see the changes in real time. This can be accomplished using the ActiveTransaction class.

It must be noted that the ActiveTransaction class uses long running transactions that can have a negative impact on performance and concurrent access. ActiveTransactions should only be used as a last resort when all other possibilities have been discarded. See "How to use the ActiveTransaction Class" on page 284 for more details.

Transaction Class Methods

The Transaction class changes the behavior of the underlying data beans. A *persist()* call to a data bean that is enlisted in a Transaction will be a no-op. The actual persistence of changes will occur when the Transaction *commit()* method is invoked. This allows the use of short duration database transactions, which minimize locking and improve concurrent access. The Transaction class provides the following public methods:

• public Transaction(): the default constructor

• public void enlist(BusinessObject bizObj) throws ICCException: this method is used to enlist a business object in an existing Transaction.

- public void enlist(TransactionSupport bean) throws ICCException: this method is used to enlist a data bean in an existing Transaction. The data bean must implement the TransactionSupport interface. All generated data beans automatically implement this interface.

- public void commit() throws ICCException: this method will persist all changes made to the enlisted objects and if successful it will then commit those changes. If any persist operations fail, then the entire transaction will be rolled back.

- public void rollback() throws ICCException: this method restores all enlisted objects, discarding all changes.

To manage the logical transaction you must keep a reference to the Transaction instance.

If a business object or data bean is enlisted in a Transaction, then the *persist()* method will be a no-operation method. Invoking the transaction class's *commit()* method will first persist the changes and then commit the database transaction.

The Transaction class is extremely lightweight and should incur minimal overhead. This technique does not use a two-phase commit mechanism due to performance, concurrency, and database maintenance issues.

Limitations

1. Changes applied to a business object cannot be recovered if the server fails prior to committing the transaction. This is an inherent limitation of any transaction mechanism.

2. Transactions that span multiple database servers cannot be guaranteed.

3. Transactions that involve non-database server data sources cannot be guaranteed.

Sample Usage

An application that wants to update multiple business objects within the context of a transaction will perform the following steps along the lines of this sample code extract:

```
ShoppingCartBean scBean = new ShoppingCartBean();
OrderBean oBean = new OrderBean();
Transaction trans = new Transaction();
trans.enlist(scBean);
trans.enlist(oBean);
...
Restore and make any changes to scBean and oBean.
```

```
Changes can be made in any order.
...
trans.commit();
```

See "Detailed Commit Functionality Description" on page 284 for more information on how the commit operation is managed when one or more data sources are involved.

How to use the ActiveTransaction Class

The ActiveTransaction methods and usage are identical to those of the Transaction class. The difference lies in the database transaction duration and persist timing.

When a *persist()* operation is invoked against a data bean that is enlisted in an ActiveTransaction, this immediately begins a database transaction and applies all update operations. This database transaction will be used by all enlisted data beans that use the same Data Source. The ActiveTransaction *commit()* will commit the database transaction.

In order to provide this functionality, the ActiveTransaction exhibits the following behavior:

An active database connection is established and tied up for the duration of the ActiveTransaction. Since the transaction duration is indeterminate, it may remove the connection from the available pool for several minutes. The connection will also remain tied up if the ActiveTransaction is not committed or rolled back. This makes the ActiveTransaction unsuitable for browser based usage.

An active database transaction is established and tied up for the duration of the ActiveTransaction. Depending on the nature of the database operations that are performed, this may result in restrictive locks being held against database resources. In the worst case, these locks may limit concurrent access for the duration of the transaction.

Because the ActiveTransaction uses long duration database transactions, there is an increased risk of database deadlocks.

ActiveTransactions should only be used as a last resort, and even then only for administrative or message based applications.

Detailed Commit Functionality Description

This section describes how a commit operation is managed ina transaction.

Commit with one Database Server

The *commit()* method will apply updates to all enlisted objects within the same transaction and connection. If the updates succeed, then it will commit the transaction. If any of the updates fail, then the transaction will be rolled back.

Commit with Multiple Database Servers

The commit method will apply updates to all enlisted objects. It will not be possible to apply these updates within the same transaction. Only if the all of the updates succeed will it commit each physical transaction. If any of the updates fail, then all transactions will be rolled back.

Commit with a Database Server and non-Database Server Data Source

The *commit()* method will apply updates to all enlisted objects that reference a database server. If the updates succeed, then it will then apply updates to the non-database server objects. If those updates succeed, then and only then, will it commit the original updates.

This leaves open a small window for failure, but the size of that window is minimized. The failure could only be connection-related. Logical failures should be caught by this technique.

CHAPTER 23 *Resources*

This chapter describes how to use resources to enhance the end-user experience.

Overview

Resources provide a general mechanism to attach attributes to data objects without specifying the attributes as data fields. Typically, you use resources in situations where some, but not all data objects of a particular type, need an attribute such as datasheet or image associated to them.

For example, you might want to associate product datasheets with some products, though not with all products, or an image with some *C3* Advisor questionnaire pages, but not with all.

Each resource has a resource type: out of the box, the following resource types are supported:

- image
- datasheet
- longtext
- video
- audio

More resource types can be created at implementation time simply by adding them to the **ResourceTypeList** XML data file. Each data object can have zero or more resources, but it can have only one resource of a particular type.

When you want to retrieve resources for a particular data object, you first retrieve the resource key, and then restore a resource list data object as follows:

```
DataContext dataContext = new DataContext();
Long resourceKey = msBean.getResourceKey();
ResourceListBean resourceListBean =
(com.comergent.bean.simple.ResourceListBean)
OMWrapper.getObject("com.comergent.bean.simple.ResourceListBean");
DsQuery dsQuery = QueryHelper.newWhereClause("ResourceKey",
    DsQueryOperators.EQUALS, resourceKey);
resourceListBean.restore(dataContext, dsQuery);
QueryHelper.freeQuery(dsQuery);
```

To retrieve the value of a resource of a particular type (say, an image), you must iterate through the ResourceListBean to identify the appropriate resource:

```
ListIterator resourceList = resourceListBean.getIterator();
ResourceBean resourceBean = null;
while (resourceList.hasNext())
{
    resourceBean = (ResourceBean) resourceList.next();
    if (resourceBean.getResourceTypeKey().intValue() == 1020)
    {
        break;
    }
}
if (resourceBean != null)
{
    String valueString = resourceBean.getResourceValue();
}
else
{
    /* Handle the case where there is no image resource */
}
```

Note that helper class and beans have been created in various parts of the Comergent eBusiness System. For example, ResourceHelper is a class that is used by presentation beans to retrieve resources by type. It supports methods such as:

String getResourceValue(String type)

The abstract class ProdServResource provides a wrapper around this class and presentation beans such as BizConditionBean in the productService packages extend this class to provide easy access to resources.

JSP Page Layer

Resources can be used to "decorate" pages used to display data objects. For example, if you associate a resource of type image with a questionnaire page, then you can display the image on the questionnaire page by adding the following to the **AdvisorBody.jsp** JSP page:

```
<IMG SRC="<%= qpb.getResourceValue("image") %>"/>
```

In this example, you are using the fact that the IPresQueryPage class extends the IProdServResource interface and implements the getResourceValue() method.

Data Services Layer

At the data services level, data objects are linked to resources using a ResourceKey data field: this maps to a RESOURCE_KEY column in the table that underpins the data object. For example, the Product data object declares a ResourceKey data field and this maps to the RESOURCE_KEY column of the CMGT_PRODUCT table.

Resources are maintained in the CMGT_RESOURCE and CMGT_RESOURCE_LOCALE tables. The RESOURCE_KEY column in a data object table is a foreign key to the RESOURCE_KEY column of the CMGT_RESOURCE table: this and the RESOURCE_TYPE_KEY column comprise the primary key for the CMGT_RESOURCE table: that is, a resource key and resource type uniquely determine the resource and its value.

A resource key is automatically generated for a data object when the first resource is assigned to the data object. The resource key value is unique across all data objects in the Comergent eBusiness System: no other data object will use that resource key, and only resources that are assigned to that data object will have that resource key.

CHAPTER 24   *State Machines*

In many situations, a data object has a state field which is used to manage the data object's life cycle. This chapter provides an overview of how to use state machines to manage the life cycle: what state transitions are permitted and what happens as a data object transitions from one state to another. It covers:

*   "Overview" on page 291
*   "State Machine Configuration Files" on page 293
*   "Customizing a State Machine" on page 296

Overview

Release 6.4 introduces the ability to manage data objects and their transition from one state to another using state machines. Each state machine manages the transitions for a specific data object, such as orders or order line items. The state machine is specifed as an XML file. See "State Machine Configuration Files" on page 293 for the structure of a state machine file.

You instantiate a state machine for a data object by calling the *getStateMachine(String s)* method of the StateMachineFactory class. This method returns an object that implements the IStateMachine interface. This is the state machine class that will manage state transitions for the data object. For example:

```
IStateMachine sm = smf.getStateMachine("OrderStateMachine");
```

An additional parameter can be passed in to specify the state machine for a particular partner:

```
IStateMachine sm = smf.getStateMachine("OrderStateMachine", key);
```

When a state machine is instantiated, it reads in the corresponding **StateMachine.xml** configuration file. This file specifies the precise processing that each input request should undergo, and how the data object should be moved from one state to another. The mapping from state machine to **StateMachine.xml** file is defined in the **StateMachineList.xml** configuration file.

Each Input element specifies the actions that should be taken when the request is made by calling the *performInput()* method of the state machine. This method has a parameter to specify the ID of the input. Typically, there will be a Helper class, such as the OrderStateMachineHelper class that provides a mapping between the input name, such as "ORDER INPUT XML PLACE" and an integer value (in this case, 25). The Roles attribute of the Input element determines which users can perform the action.

The signature of the IStateMachine state machine interface *performInput()* method is:

```
performInput(Long inputId, Object obj, IRdUser user, Hashtable ht)
```

The second parameter in the method call is used to pass in the object that will be processed by the business logic. The Hashtable parameter is used to pass in any other processing parameters that the handler classes may need.

If the input is not valid for this state, then an InputFailedException is thrown, and this should be caught and handled by the apropriate business logic. If the input is valid, then the ActionHandler classes are called, in the order listed in the ActionHandlerList element. Each ActionHandler must implement the IActionHandler interface by providing the *performInputAction()* method: this is the busines logic used to process the input request.

The signature of the IActionHandler interface *performInputAction()* method is:

```
performInputAction(IInput input, Object obj, IRdUser user,
    Hashtable ht)
```

The second parameter in the method call is used to pass in the object that will be processed by the handler. The Hashtable parameter is used to pass in any other processing parameters that the handler class may need.

If any of the ActionHandlers throw an InputFailedException, then the processing of the input request is stopped and the object stays in its current state. If all the

ActionHandlers succeed, then the object is moved into the state specified by the NextState element.

The ActionEvents declared in the ActionEventList element are fired: these should be handled by the EventBus (see CHAPTER 9, "Events" for more information). You should use ActionEvents to trigger processing that should not affect the main business logic used to process the action, but which should happen if the business logic successfully completes. For example, sending out email notifications in the event that an order is successfully placed can be handled using ActionEvents.

State Machine Configuration Files

StateMachineList.xml Configuration File

The state machines are declared in the **StateMachineList.xml** configuration file. By default, its locations is *debs_home***/Comergent/WEB-INF/statemachines/** and its location is specified in the StateMachines element of the **Comergent.xml** configuration file.

Each state machine is defined by specifying the name of its **StateMachine.xml** configuration file. For example:

```
<StateMachine>EnterpriseOrderStateMachine.xml</StateMachine>
```

It is possible to define different state machines for different storefront partners using the StorefrontStateMachines element. For example, the following example declares the state machine for the storefront partner whose partner key is "21":

```
<StorefrontStateMachines StorefrontKey="21"
    StorefrontName="Allnet">
    <StateMachine>AllnetOrderStateMachine.xml</StateMachine>
</StorefrontStateMachines>
```

For a given storefront, if no StateMachine element is defined, then the corresponding state machine of the DefaultStateMachines element is used.

StateMachine.xml Configuration File

Each **StateMachine.xml** configuration file defines a state machine for a particular application. It specifies how state transitions should be processed for a data object. It comprises a StateMachine element and a child StateList element that contains a set of State elements: each State element specifies through its child elements what the valid inputs are for the state, and how each input should be processed.

The StateMachineName attribute of the StateMachine element is used by the StateMachineFactory class to retrieve a named state machine.

The value, *lookupType*, of the LookupType element of the **StateMachine.xml** is used to retrieve the lookup codes from the CMGT_LOOKUPS table.

- Each State element must correspond to a lookup code whose lookup type is *lookupType*State so that user-viewable names can be displayed for each supported locale.

- Each Input element must correspond to a lookup code whose lookup type is *lookupType*Input.

Note that for legacy reason, some data objects (Order and RFQ) use their Status lookup type. For example, there are lookup codes for the OrderStatus and OrderInput lookup types and these are used by the OrderStateMachine because it declares its LookupType by:

```
<LookupType>Order</LookupType>
```

A typical State element looks like this:

```
<State Name="Open" Start="true">
  <Description>
    This is the initial State in the Ordering flow.
  </Description>
  <InputList>
    <Input Name="ORDER INPUT USER PLACE"
        Roles="Partner.DirectCommerceUser;Registered.User;
           Enterprise.CustomerServiceRepresentative">
    <Description>This is the "Place" action.</Description>
    <NextState>Order Submitted</NextState>
        <ActionHandlerList>
            <ActionHandler>
    com.comergent.apps.orderMgmt.orders.bizAPI.OrderPlaceHandler
            </ActionHandler>
            <ActionHandler>
    com.comergent.apps.orderMgmt.orders.bizAPI.OrderPersistHandler
            </ActionHandler>
            <ActionHandler>
    com.comergent.apps.orderMgmt.orders.bizAPI.SaveDiscounts
            </ActionHandler>
            <ActionHandler>
    com.comergent.apps.orderMgmt.orders.bizAPI.WriteHistoryHandler
            </ActionHandler>
        </ActionHandlerList>
        <ActionEventList>
            <ActionEvent>OrderPlaceEmailEvent</ActionEvent>
        </ActionEventList>
    </Input>
    <Input Name="ORDER INPUT XML PLACE"
        Roles="Partner.DirectCommerceUser;Registered.User">
```

```
<Description>This is the "Place" action.</Description>
<NextState>Order Submitted</NextState>
<ActionHandlerList>
      <ActionHandler>
com.comergent.apps.orderMgmt.orders.bizAPI.PreProcessXMLPlaceHandler
      </ActionHandler>
      <ActionHandler>
com.comergent.apps.orderMgmt.orders.bizAPI.OrderPlaceHandler
      </ActionHandler>
      <ActionHandler>
com.comergent.apps.orderMgmt.orders.bizAPI.OrderPersistHandler
      </ActionHandler>
      <ActionHandler>
com.comergent.apps.orderMgmt.orders.bizAPI.SaveDiscounts
      </ActionHandler>
      <ActionHandler>
com.comergent.apps.orderMgmt.orders.bizAPI.WriteHistoryHandler
      </ActionHandler>
   </ActionHandlerList>
   <ActionEventList>
      <ActionEvent>OrderPlaceEvent</ActionEvent>
   </ActionEventList>
</Input>
  </InputList>
</State>
```

The Roles attribute of the Input element is used to check that only appropriate users can act on the object when it is in a given state. Note that the list of roles is delimited by semi-colons (";").

Action Events

When an input has finished being processed, the state machine can also fire action events. These are events that can be broadcast to other parts of the Comergent eBusiness System so that they can take appropriate actions if need be. Action events are propagated using the event bus framework. see CHAPTER 9, "Events" for details.

You declare the action events using the ActionEventList element: an input can declare zero or more ActionEvent elements. Each such event must be declared in the **events.xml** configuration file. For example, for the ActionEvent declared above, there should be the corresponding entry in the **events.xml** configuration file:

```
<event
   class="com.comergent.api.apps.orderMgmt.orders.OrderPlaceEvent">
   <description>This event is fired when an Order is placed
   </description>
   <consumers-list>
```

```
      <consumer>
com.comergent.reference.apps.orderMgmt.orders.OrderEmailEventHandler
      </consumer>
      <consumer>
com.comergent.reference.apps.orderMgmt.orders.OrderAtStorefrontEmai-
lEventHandler
      </consumer>
      <consumer>
com.comergent.reference.apps.salesContracts.SalesContractOrder-
EventHandler
      </consumer>
   </consumers-list>
</event>
```

Customizing a State Machine

In your implementation of the Comergent eBusiness System you may need to modify the way in which a state machine works for a data object. Typically, this will involve one or more of the following:

*   "Changing the Business Logic associated with a Change in State" on page 296

*   "Changing the Available State Transitions" on page 297

*   "Adding a New State" on page 298

Changing the Business Logic associated with a Change in State

If you want to modify the logic that is executed when an object changes from one state to another, then you must modify the list of ActionHandler elements associated with the state and input.

Example

For example, suppose that you want to replace the SaveDiscount ActionHandler class with a custom class in the logic that is executed when an Order in the Open state receives the ORDER INPUT USER PLACE input.

1.   First, you must create your custom class, say the CustomSaveDiscount class. This class must implement the IActionHandler interface as described above.

Then you modify the ActionHandlerList element by changing the corresponding ActionHandler element in the list of ActionHandlers:

```
<Input Name="ORDER INPUT USER PLACE"
    Roles="Partner.DirectCommerceUser;Registered.User;
        Enterprise.CustomerServiceRepresentative">
```

```
<Description>This is the customized "Place" action.</Description>
<NextState>Order Submitted</NextState>
    <ActionHandlerList>
        <ActionHandler>
com.comergent.apps.orderMgmt.orders.bizAPI.OrderPlaceHandler
        </ActionHandler>
        <ActionHandler>
com.comergent.apps.orderMgmt.orders.bizAPI.OrderPersistHandler
        </ActionHandler>
        <ActionHandler>
com.comergent.apps.orderMgmt.orders.bizAPI.CustomSaveDiscounts
        </ActionHandler>
        <ActionHandler>
com.comergent.apps.orderMgmt.orders.bizAPI.WriteHistoryHandler
        </ActionHandler>
    </ActionHandlerList>
</Input>
```

Changing the Available State Transitions

If you want to modify the available state transitions for a state, then you must
modify the InputList element by adding or removing the input associated with the
state transition.

Example

For example, suppose that you want to allow an order that is is in the Open state to
receive an input that suspends the order. Because we want to use a new input, we
must create a corresponding new lookup code along these lines:

```
<LightWeightLookup state="INSERTED">
    <LookupType state="INSERTED">OrderInput</LookupType>
    <LookupCode state="INSERTED">1020</LookupCode>
    <Locale state="INSERTED">en_US</Locale>
    <Description state="INSERTED">ORDER INPUT SUSPEND</Description>
</LightWeightLookup>
```

Then, in the State element for Open orders, add an input element along these lines:

```
<Input Name="ORDER INPUT SUSPEND"
    Roles="Partner.DirectCommerceUser;Registered.User;
        Enterprise.CustomerServiceRepresentative">
<Description>This is the customized "Place" action.</Description>
<NextState>Order Suspended</NextState>
    <ActionHandlerList>
        <ActionHandler>
com.comergent.apps.orderMgmt.orders.bizAPI.OrderSuspendHandler
        </ActionHandler>
        <ActionHandler>
com.comergent.apps.orderMgmt.orders.bizAPI.WriteHistoryHandler
```

```
            </ActionHandler>
        </ActionHandlerList>
    </Input>
```

Note that in this example, we want the next state to be a new state (Order Suspended) that does not yet exist in the system, and so we must also add this state as described in "Adding a New State" on page 298.

Adding a New State

You may sometimes need to add a new state for an object to the ones that come with the Comergent eBusiness System. To do so, you must consider what inputs from other states should put an object into the new state, and what valid inputs will change an object's state from the new state to one of the existing ones.

First, you should add the new lookup codes to reflect the new state and new inputs. Having done so, you will need to add Input elements to the existing State elements to support the transitions to the new state, and you will need to add a new State element that defines the valid inputs that the new state supports.

Example

For example, suppose that we want to add an Order Suspended state for the Order object. We want to allow for Open orders to be placed in the Order Suspended state, and the only valid input we want to support for an order in the Order Suspended state is to "reopen" the order by putting it back into the Open state.

1. First, we add the new state to the Order state lookup codes:

```
<LightWeightLookup state="INSERTED">
    <LookupType state="INSERTED">OrderStatus</LookupType>
    <LookupCode state="INSERTED">1020</LookupCode>
    <Locale state="INSERTED">en_US</Locale>
    <Description state="INSERTED">Order Suspended</Description>
</LightWeightLookup>
```

Note the use of the OrderStatus lookup type as noted above.

2. Next, we add the input lookup codes needed to get orders into and out of the new state:

```
<LightWeightLookup state="INSERTED">
    <LookupType state="INSERTED">OrderInput</LookupType>
    <LookupCode state="INSERTED">1020</LookupCode>
    <Locale state="INSERTED">en_US</Locale>
    <Description state="INSERTED">ORDER INPUT SUSPEND</Description>
</LightWeightLookup>
<LightWeightLookup state="INSERTED">
    <LookupType state="INSERTED">OrderInput</LookupType>
```

```
    <LookupCode state="INSERTED">1030</LookupCode>
    <Locale state="INSERTED">en_US</Locale>
    <Description state="INSERTED">ORDER INPUT REOPEN</Description>
</LightWeightLookup>
```

3. Now we modify the Open State element of the order state machine by adding the new input that is used to suspend an open order:

```
<Input Name="ORDER INPUT SUSPEND"
    Roles="Partner.DirectCommerceUser;Registered.User;
        Enterprise.CustomerServiceRepresentative">
<Description>This is the customized "Place" action.</Description>
<NextState>Order Suspended</NextState>
    <ActionHandlerList>
        <ActionHandler>
com.comergent.apps.orderMgmt.orders.bizAPI.OrderSuspendHandler
        </ActionHandler>
        <ActionHandler>
com.comergent.apps.orderMgmt.orders.bizAPI.WriteHistoryHandler
        </ActionHandler>
    </ActionHandlerList>
</Input>
```

4. Now we add a new State element to say what inputs an order in the Order Suspended state can accept:

```
<State Name="Order Suspended" Start="true">
  <Description>
    This is the suspended State in the Ordering flow.
  </Description>
  <InputList>
    <Input Name="ORDER INPUT REOPEN"
        Roles="Partner.DirectCommerceUser;Registered.User;
            Enterprise.CustomerServiceRepresentative">
    <Description>This is the "reopen" action.</Description>
    <NextState>Open</NextState>
        <ActionHandlerList>
            <ActionHandler>
com.comergent.apps.orderMgmt.orders.bizAPI.OrderReopenHandler
            </ActionHandler>
            <ActionHandler>
com.comergent.apps.orderMgmt.orders.bizAPI.OrderPersistHandler
            </ActionHandler>
            <ActionHandler>
com.comergent.apps.orderMgmt.orders.bizAPI.SaveDiscounts
            </ActionHandler>
            <ActionHandler>
        </ActionHandlerList>
        <ActionEventList>
            <ActionEvent>OrderReopenEmailEvent</ActionEvent>
```

```
        </ActionEventList>
    </Input>
  </InputList>
</State>
```

CHAPTER 25 *Widgets*

This chapter describes how to use Comergent widgets in your Comergent eBusiness System applications. Widgets are primarily used for *C3* Portal pages, but may be used throughout the system.

Overview

A widget is a custom tag that can be used in a JSP page (referred to here as the container JSP page) to provide a UI component in the generated Web page. It is self-contained in that the widget tag may be moved from one part of a JSP page to another without effecting the look-and-feel of the rest of the page and the widget content is generated independently of the rest of the page.

Widgets work by making a separate call to the Comergent eBusiness System which executes a different message type to the rest of the container JSP page. By executing a distinct message type, you can call a different controller and use a distinct JSP page to generate the output for the widget content. The controllers used by widgets should always extend the WidgetController class. This controller extends the IncludeController class which invokes the *include()* method on the RequestDispatcher as opposed to the more commonly used *forward()* method.

Widgets work as HTML includes and so they re-use the same cascading style sheets as the rest of the Web page. Care must be taken in defining Javascript

functions and form variables to avoid name duplication with other elements of the Web page. See "Guidelines" on page 302.

Widget Tag

The widget tag takes these named attributes:

- height: the height of the widget element in pixels

- name: the name of the widget is the message type executed by the Comergent eBusiness System

- width: the width of the widget element in pixels

Note that the height and width parameters may not always be realized in the Web page seen by the user. These set the height and width of the HTML table attributes, but the user's browser renders the page in the light of all of the HTML it receives for the Web page.

In addition, you can define any number of parameters in the body of the tag. These are passed to the controller and BLC through the request object. Each parameter takes the form *name*=*value* and each pair is separated by an ampersand ("&") from the next. For example:

```
<cmgt:widget name="MyOrdersWidget" height="300" width="500">
    numberOfRows=5&orderBy=lastupdate
</cmgt:widget>
```

Guidelines

Follow these steps to create a widget:

1. Define the message type in the appropriate **MessageTypes.xml** file. Typically, it will be of this form:

    ```
    <MessageType Name="MyOrdersWidget">
        <ControllerMapping>
            com.comergent.apps.orderMgmt.orders.controller.-
                MyOrdersWidgetController
        </ControllerMapping>
        <JSPMapping>
            ../orderMgmt/Orders/MyOrdersWidget.jsp
        </JSPMapping>
    </MessageType>
    ```

2. Create the controller class declared in the ControllerMapping element to process the request. You should extend the WidgetController class by

overwriting its *execute()* method. By calling the WidgetController's method *callJSP()*, you can forward the request to the JSP page defined by the JSPMapping element.

3. Create the JSP page declared in the JSPMapping element. You must follow these rules in writing this JSP page:

   • Widgets should use the widget style for display: this ensures that they have a common look-and-feel throughout the Comergent eBusiness System.

   • Begin and end the HTML content of this page with <table> and </table>. All widgets define their content in the context of a table. Use only HTML tags and syntax that are valid in an HTML table.

   • The JSP page must include the **cmgtinclude.jspf** page through the standard JSP <@ include> directive.

   • Adopt a naming convention for all parameters, functions, and variables in the JSP page so that they do not conflict with other usages on the container JSP page. We recommend a convention that prepends the name of the widget before the object name. For example: *MyOrdersWidget_function* or *MyOrdersWidget_formName*.

   • Take care that actions performed on the widget's JSP page do not act on the container page. Links or buttons on a widget's page should take a user to a new Web page.

4. You can retrieve parameters defined in the widget tag through the request object. For example, using the example above, you can retrieve the value of the orderBy parameter by:

```
String temp_ParameterString = request.getParameter("orderBy");
```

As with any request parameter, it is returned as a String, so you must parse a parameter to recover a numeric value.

5. You can retrieve session information through a standard call to the ComergentSession object. In particular, you can retrieve the session user by the following:

```
User temp_User = comergentSession.getUser();
```

6. Add the widget to the container JSP page with the following:

```
<% out.flush(); %>
<cmgt:widget name="widgetName"
    height="heightInPixels" width="widthInPixels">
    parameter1=value1&parameter2=value2&...&parametern=valuen
```

```
</cmgt:widget>
```

Note that you must flush the contents of the output stream before the widget. If you do not do this, then the HTML generated by the widget is dispatched to the browser before the rest of the HTML stream.

7. Make any desired modifications to the **ErrorWidget.jsp** JSP page. This page is displayed in the area defined for the widget if an unhandled error condition arises.

Integrating a Widget in a Portal Page

If you have your own portal application and you want to add a Comergent eBusiness System widget into a portal page, then you can do this as follows:

1. Create your Comergent eBusiness System widget by following the steps described in "Guidelines" on page 302.

2. In your portal application, declare the portlet using the syntax required to set up a portlet. The portlet type of Comergent eBusiness System portlets is HTML.

3. To specify the URL required to provide the portlet content, use a URL like this:

```
http://<server:port>/Comergent/en/US/direct/matrix?login=user-
name&passwd=password&cmd=directLogin&LoginData-messageType=MyO-
rdersWidget&LoginData-displayBorders=false&LoginData-
entryPoint=direct&validate=true
```

The login and password parameters need to be generated dynamically based on the user, and so you will need to be able to retrieve the username and password for the user to identify themselves to the Comergent eBusiness System.

The LoginData-messageType parameters is used to specify the widget's message type. If other parameters are required by the widget, then pass them in as part of the URL too.

Example

In this example, we show how the MyOrdersWidget is used to display a small panel of current orders to a user.

Container JSP Page

The container JSP page has the following text:

```
<% if (isDirect) { %>
    <% out.flush(); %>
    <cmgt:widget name="MyOrderWidget" height="300" width="500" >
        numberOfOrders=5
    </cmgt:widget>
<% } %>
```

Note the use of the boolean variable isDirect to determine whether the user is a direct commerce user (and hence a user who can create orders).

MessageTypes.xml Entry

The following entry is added to the direct commerce user message group:

```
<MessageType Name="MyOrdersWidget">
    <ControllerMapping>
        com.comergent.apps.orderMgmt.orders.controller.-
            MyOrdersWidgetController
    </ControllerMapping>
    <JSPMapping>
        ../orderMgmt/Orders/MyOrdersWidget.jsp
    </JSPMapping>
</MessageType>
```

WidgetController

A new class MyOrdersWidgetController is created which extends the WidgetController class. Its *execute()* method overwrites the *execute()* method of the WidgetController class with:

```
public  void execute() throws
    ControllerException, ICCException, IOException
{
    try
    {
        String numberOfRowsStr = null;
        int numberOfRows = DEFAULT_NUMBER_OF_ROWS;
        numberOfRowsStr = request.getParameter("numberOfRows");
        if (numberOfRowsStr != null && !(numberOfRowsStr.equals("")))
        {
            numberOfRows = Integer.parseInt(numberOfRowsStr);
        }
        boolean bAscending = false;
        IOrderFactory fac = OrdersAPI.getFactory();
        IRdLightWeightOrdersList orderedCarts =
        fac.getListOfOrders(numberOfRows, "UpdateDate", bAscending);
```

```
        request.setAttribute("orderedCarts",
            (IRdLightWeightOrdersList) orderedCarts);
    }
    catch (Exception e)
    {
        sendWidgetError(e);
    }
    // Dispatch
    callJSP();
}
```

The final call *callJSP()* forwards the request to the widget JSP page:
MyOrdersWidget.jsp. Note the use of the OrdersAPI class and IOrderFactory
interface to retrieve carts of the appropriate type.

Widget JSP Page

Every widget JSP page has these important components:

- the include tag for **cmgtinclude.jspf**

- a scriptlet section that recovers the height, width, and other parameters
  from the request

- the HTML table that provides the content of the widget

See the **MyOrdersWidget.jsp** file for an example widget JSP page. It is in
debs_home/**Comergent/WEB-INF/web/en/US/orderMgmt/Orders/MyOrders-
Widget.jsp**.

CHAPTER 26   *Customizing Advanced Search*

The Comergent eBusiness System offers a sophisticated search capability. It is based on the Apache Lucene project. This chapter provides a guide to customizing the search capabilities to meet your implementation requirements.

Overview

The advanced search capabilities of the Comergent eBusiness System are based on the creation of search indexes. A search index is a set of files that index the occurrence of search terms in data objects. At any one time, there may be many search index sets stored in the Comergent eBusiness System.

Different search applications can access the same or different search index sets, but at any one time each search application can use only one *active* index set.

When an end-user performs a search, the search engine is used to search through the index for the requested terms and to return the search results. The search results are filtered prior to the results being displayed to the end-user to remove products that they should not be able to see: for example, by virtue of the price list assignment to their partner or effectivity dates on the products.
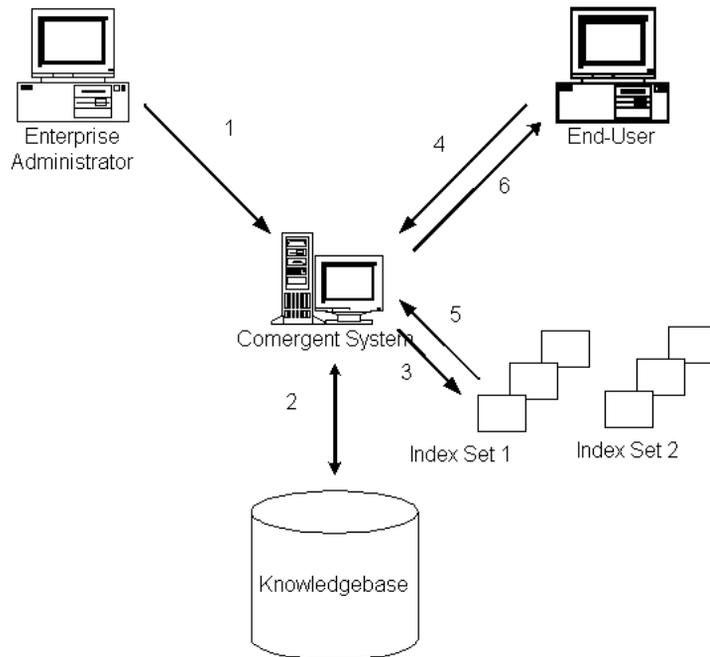
FIGURE 8. Advanced Search Concepts

Conceptually, the process comprises these steps:

1. An enterprise administrator initiates the creation of a search index. You can click the **Generate New** button in the Search Administration tab of the Product Manager application or create a cron job to generate search indexes at regular intervals.

2. The index set is created by an index set builder class and the index builder classes it invokes: these work through the product catalog by instantiating data beans for each product, feature, and so on.

3. The index builders create index documents and write out the documents to the file system on the Comergent eBusiness System machine. Each index set is created in a separate directory, for example as: *debs_home*/**Comergent/ WEB-INF/data/searchIndex/en_US/MasterIndex_101/**. In the directory, a number of index files are created: they contain a binary representation of the index set using a Lucene file format.

4. An end-user initiates a search either by performing a simple search using the **Find** button on the catalog pages or by navigating to the Advanced Search page and creating a search query using search terms.

5. The Comergent eBusiness System processes the request as follows:

    a. A list of search terms is created based on the terms entered by the user.

    b. A searcher is instantiated and initialized with the search terms: one for each row of the Advanced Search Terms form entered by the user.

    c. The *search()* method is invoked on the searcher. First, the query search terms are used to retrieve a raw list of search results, and then the filter search terms are used to refine the list by filtering out search results that should be excluded by virtue of factors such as the effectivity dates and price lists. The result is a list of hits.

    d. The hits are transformed into a list of objects representing the search results.

6. The search results are passed to the JSP page and displayed to the end-user.

Building Indexes

Search indexes are built using instances of the IndexBuilder classes and its sub-classes. All the builders implement the IIndexBuilder interface and should extend the abstract BasicBuilder class. The main IndexBuilder method used to create an index is *build()*. This method generates a document based on the list of entities passed to it.

Index builders can contain one or more child index builders, and so you can have a nested hierarchy of index builders. You can add and remove child index builders from parent index builders. In general, the hierarchy of index builders should reflect the relationships between the entities being indexed. For example, you should use a child index builder to index the data fields of child data objects.

Each index builder is associated with a prefix: the prefixes are used to build up a concatenated list of prefixes that reflect the hierarchy of index builders. When a child index builder creates index entries, its entries are prefixed by the prefix of its parent.

The hierarchy of index builders and associated prefixes are specified using the **SearchConfigurationProperties.xml** file. The index builders are declared in a nested set of Builder elements and each index builder is declared by specifying its prefix and the data field it indexes. The prefix is the key attribute of the Builder element and the index fields are declared as a list of IndexField elements.

```
<Builder key="prod" builderClass="package.GenericBuilder">
 <IndexFields>
  <IndexField id="ProductID" key="ID" />
  <IndexField id="HasAssignedSupersession" key="SUPER" />
 </IndexFields>
 <Builders>
  <Builder key="vftr" builderClass="VisibleProductFeatureBuilder">
   <IndexFields>
    <IndexField id="FeatureKey" key="ID" />
    ...
    <IndexField id="Description" key="DESC" />
   </IndexFields>
  </Builder>
  ...
  <Builder key="attach" builderClass="ProductAttachmentBuilder">
   <IndexFields>
    <IndexField id="DataSheet" key="DATASHEET" />
   </IndexFields>
  </Builder>
 </Builders>
</Builder>
```

```
                    ┌─────────────────────────┐
                    │      GenericBuilder      │
                    ├─────────────────────────┤
                    │       key is prod        │
                    └─────────────────────────┘
                         │               │
          ┌──────────────────────────┐  ┌──────────────────────────┐
          │ VisibleProductFeatureBuilder │  │  ProductAttachmentBuilder │
          ├──────────────────────────┤  ├──────────────────────────┤
          │     key is prod.vftr      │  │  key is prod.DATASHEET   │
          └──────────────────────────┘  └──────────────────────────┘
```
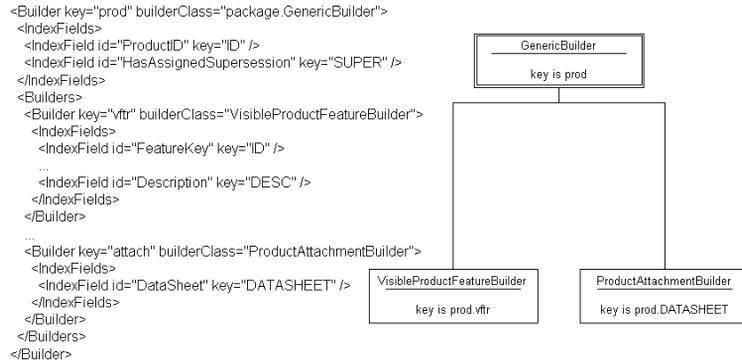
FIGURE 9. Index Builder Hierarchy

Note that the **SearchConfigurationProperties.xml** file is a shared file: access to it should use the *adjustFileName()* method and use the boolean values to access a shared, private, noloadable file.

Customizing Dictionary Mappings

If a user is searching for "color", then you might want them to find items that use the terms "colour" or "hue". You specify the equivalence of search terms and words using the **CatalogDictionary.mappings** file. Each line of this file takes the form:

```
term,word,word,word,...
```

The first string is referred to as the "term" and the following words are equivalent words. You can add terms or edit terms to specify the word equivalences you want. When a search is performed, occurrences of these words are regarded as equivalent to occurrences of the term. If a user searches for the term, they will see results that contain any of the other words. For example, suppose that you have in the mappings file:

```
color,colour,hue
```

Searches for "color" will also return references to "colour" and "hue"; however, searches for "hue" will not return references to "color" and "colour" unless you have another line that reads:

```
hue,color,colour
```

Note that the **CatalogDictionary.mappings** file is a shared file: access to it should use the *adjustFileName()* method and use the boolean values to access a shared, private, noloadable file.

Processing Search Requests

Each end-user search request is handled by a sequence of these controllers:

1. CatalogAdvancedSearchController:

   a. Receives the search request and sets the session context: the SessionContext class is instantiated with a string that identifies the index set to be used. The SessionContext class is used to hold information about the user's context such as their current locale, their price lists, and so on.

   b. It pre-processes the request by updating any filters to be used to filter the results.

   c. It creates a list of search terms as a List of search terms: each term is an instance of the SearchTerm class.

   d. The controller forwards the search request to the CatalogSearchResultController class.

2. CatalogSearchResultController: this class executes the search as follows:

   a. Initialize the SearchContext with information from the user's session.

   a. Create a Searcher using the List of SearchTerms created by the CatalogAdvancedSearchController.

   b. Invoking *search()* on the Searcher object. This returns a list of hits as a Hits object.

   c. The Hits object is transformed into a List of data beans: this is set in the request object and control is passed to the display controller and JSP page.

   d. Search results can be displayed either by rank or by category: the controller determines which display view has been requested, and forwards the request either using the message type "catSearchResultByCategory" or "catSearchResultByRank".

Lucene Classes

The advanced search functionality uses the APIs provided by the Apache Lucene project. You should not need to modify these classes in any way. In particular, the following important classes are used:

- Analyzer: the class used to parse, stem, and tokenize index entries.

- Document: the class used to hold index fields and to write them out to the index set files.

- Field: an index entry together with flags that indicate how it is to be used in searches.

- Filter: used to filter search results. It is extended by the CmgtFilter class.

- Hits: this class encapsulates the results of a search of the index set. The SearchResultBuilder class provides the *process()* method that transforms a Hits object into a List of results. The *score(int i)* method returns a score for the hit indexed by i: this reflects how well the hit matches the search criteria specified by the user.

- IndexReader: used to read index entries from the Lucene file format files.

- IndexWriter: used to write documents out in the Lucene file format.

- Query: the basic search object used to retrieve search results.

- Searcher: the basic class used to initiate searches over an index set. Its *search()* method typically takes a Query and a Filter and returns a Hits object.

IndexBuilder and IndexSetBuilder Classes

IndexBuilders

Index builders are Java classes that implement the IIndexBuilder interface. They are used to create index entries based on the values of fields in objects. The abstract BaseBuilder class implements the interface and all IndexBuilders should extend BaseBuilder. The principal methods that must be implemented are:

- public Document build() throws ComergentException: builds the index given the current document.

- public Document build(Document doc, IIndexBuilder parentBuilder) throws ComergentException: used when the index builder is a child of another index builder. The parent builder object is used to add the parent's

key prefix to the current index builder's prefix. The List of entities is initialized with the parent's List of entities. The BaseBuilder class defines this method so that it first calls *buildSelf()* and then recursively calls the *build()* method of the child index builders.

- protected Document buildSelf(Document doc) throws ComergentException: the method used to create the index entries for this index builder. Each entry is an instance of the Field class: the document is built up by creating each field and then adding each to the document. For example:

```
IndexFieldConfiguration iconf =
    this.conf.getIndexFieldConfiguration(id);
Field field = new Field(this.getPrefix()+iconf.getKey(),
    result, iconf.getIsStore(), iconf.getIsIndex(),
    iconf.getIsTokenize());
doc.add(field);
```

In creating a custom IndexBuilder, this method is overwritten with the specific index building method required to create the custom index entries.

- public void addBuilder(IIndexBuilder builder): add a child index builder to the current index builder.

- public List getEntities(): get the List of objects currently being indexed.

- public String getEntityType(): get the name of the class that is indexed by this index builder. For example, "com.comergent.bean.simple.ProductBean".

- public String getPrefix(): get the full prefix for entries created by this index builder. For example, "product.name".

- public void init(List entities) throws ComergentException: initializes the index builder with the list of entities to be indexed. Its main purpose is to take the list of entities passed in and build the list of entities that it must index. For example, for an index builder working on features, if the *init()* method is passed a list of products, it might be used to build a list of features that are assigned to the products along these (simplified) lines:

```
for (int i = 0; i < entities.size(); i++)
{
    IBizProduct prod = null;
    try
    {
        prod = (IBizProduct) entities.get(i);
        prod.restoreFeatures(true, context.getLocale());
        this.addEntities(getVisibleFeatures(prod.getFeatures()));
```

```
        }
        catch(ComergentException e)
        {
            throw new ComergentException(e.toString());
        }
    }
```

• public void removeBuilder(int i) throws ComergentException: remove a child index builder.

A GenericBuilder class can be used for simple index builders: it can perform the basic index building required for data fields of data beans. Each builder contains its configuration as a BuilderConfiguration class: this is passed in its constructor. The BuilderConfiguration class holds the key to be used while indexing: this can be accessed using *getKey()* and *setKey()* methods. It also identifies the Entity class (for example, ProductBean) and a Map of IndexFieldConfiguration classes: each IndexFieldConfiguration class is a mapping between a key and an ID. The ID identifies the field of the entity class whose values are to be indexed. The key is the string is to be used to uniquely identify this key from the other keys being generated by building the index.

For example, if the Name data field of the ProductBean data object is to be indexed by an index builder whose key is "prod" and it must index using the key "name", then you would use the following calls while creating the index builder:

```
BuilderConfiguration productBuilderConfiguration =
    new BuilderConfiguration();
productBuilderConfiguration.setKey("prod");
productBuilderConfiguration.setEntityClass(
    "com.comergent.bean.simple.ProductBean");
IndexFieldConfiguration productName = new IndexFieldConfiguration();
productName.setKey("name");
productName.setID("Name");
productBuilderConfiguration.addIndexFieldConfiguration(productName);
GenericBuilder productIndexBuilder =
    new GenericBuilder(productBuilderConfiguration);
```

Index entries that it creates are retrieved using the "prod.name" key: this is the concatenation of the index builder's key with the index field's key. If the index builder is itself a child index builder of a parent index builder whose key is "catalog", then the key to retrieve entries is "catalog.prod.name".

IndexSetBuilders

Each index set is created by invoking an IndexSetBuilder. The IndexSetBuilder class is an abstract class: you must create a sub-class that implements its *build()* method.

The *build()* method does the work of indexing the objects and writing out the results to the file system. For example, the CatalogIndexSetBuilder class implements the *build()* method by:

1. Instantiating the root index builder specified in the **SearchConfigurationProperties.xml** file. Typically, this is done using the *getRootIndexBuilder()* method of the IndexSetBuilder class. It invokes a factory method that creates the hierarchy of index builders.

2. Creating an index writer: this manages writing out the index document to the index files.

3. Retrieving the list of products.

4. For each product in the list of products:

    a. Pass the product to the builder as a parameter. It gets added to the list of entities in the index builder.

    b. Create the index document by invoking *build()* on the index builder.

    c. Writing out the document to the index writer.

5. Tidying up by logging the completion time.

Search Terms

Search terms are used to specify what index entries should be returned by the Searcher class. The SearchTerm class is used to specify a single search term and then a list of search terms is passed to the searcher by invoking the *addSearchTerm()* method: this takes a List of SearchTerms as its parameter.

Search terms can be aggregated: by adding search terms together you build up the complete search term used to perform the search.

The SearchTerm class has the following important methods:

*   *addTerm()*: adds a search term to the current search term.

*   *setCondition()*: specify whether this search term is looking for a field that must or must not be in the search results. You can also use:

    ```
    setCondition(SearchTerm.CONDITION_PREFER);
    ```

    in situations where the user has expressed this preference.

*   *setFieldName()*: set the name of the index field to be searched.

- *setOP()*: specifies the operator to be used to compare the search term value to the field values.

- *setType()*: specifies whether the search term is to be used as a query search term or a filter search term.

- *setValue()*: the value to search for.

- *setWeight()*: set the weight to place on this search term when a hit is found. This is used to evaluate the score for the hit.

For example, here is an example code fragment in which the search term is defined to retrieve those prod.name fields with the value of "MXWS-7890".

```
//Create a new advanced searcher
ICmgtSearcher temp_Searcher = new AdvancedSearcher();
//initialize with the a session context
SessionContext temp_Context = createSessionContext();
temp_Searcher.init(temp_Context);
//Create a search term
SearchTerm temp_SearchTerm = new SearchTerm();
temp_SearchTerm.setCondition(SearchTerm.CONDITION_MUST);
temp_SearchTerm.setFieldName("prod.name");
temp_SearchTerm.setOP(SearchTerm.OP_EQUAL);
temp_SearchTerm.setType(SearchTerm.TYPE_QUERY);
temp_SearchTerm.setValue("MXWS-7890");
temp_SearchTerm.setWeight(1.0);
//Add search term to list of search terms
List allSearchTerms = new ArrayList();
allSearchTerms.add(temp_SearchTerm);
temp_Searcher.addSearchTerm(allSearchTerms);
//Execute the search, retrieving results as a Hits object
Hits hits = temp_Searcher.search();
```

Search Term Types

There are different types of search terms:

- Query: these search terms are used to build the initial query which is used to search the index set.

- Filter: these search terms are used to filter the results returned from the initial search query.

- Filtering Query: this is a helper type of search term used when the search request has a "Must Have" condition.

Filters

Each search result is represented as an instance of the Document class. Filter search terms can act on each document to determine whether the result meets the requirements of the filter or to filter the result out. The following code example demonstrates the use of filter search terms: it creates a combination of search terms that exclude products whose effectivity dates mean that the product would not be available at the time of the search.

```
// Create parameters used in defining search terms
String now = DateField.dateToString(new Date());
SearchFieldConfiguration sconf =
    cntx.getIndexSet().getSearchFieldConfiguration("prodSDate");
SearchFieldConfiguration econf =
    cntx.getIndexSet().getSearchFieldConfiguration("prodEDate");
// Create top-level search term
SearchTerm allTerms = new SearchTerm();
// Create top-level filter term
SearchTerm top = new SearchTerm();
top.setFieldName("effectivityCheck");
top.setCondition(SearchTerm.CONDITION_MUST);
top.setNeedParsing(false);
top.setType(SearchTerm.TYPE_FILTERING_QUERY);
// term for checking effective start date
SearchTerm st1 = new SearchTerm();
st1.setFieldName(sconf.getKey());
st1.setCondition(SearchTerm.CONDITION_MUST);
st1.setNeedParsing(false);
st1.setOP(SearchTerm.OP_LET);
st1.setValue(now);
// term for checking for effective end date
SearchTerm st2 = new SearchTerm();
st2.setFieldName(econf.getKey());
st2.setCondition(SearchTerm.CONDITION_MUST);
st2.setNeedParsing(false);
st2.setOP(SearchTerm.OP_GET);
st2.setValue(now);
//Add filter terms to top-level filter term
top.addTerm(st1);
top.addTerm(st2);
//Add filter term to top-level search term
allTerms.add(top);
```

Note the use of the OP.LET and OP.GET operators: these are the "less than or equal" and "greater than or equal" operators used to compare dates represented as Strings.

Processing Results

Once the search has returned its list of results in the form of a Hits object, you must ready the results to pass them to the JSP page used to display them. You can use a SearchResultBuilder class for this purpose: this abstract class provides methods to process the results, and sub-classes of this class can provide custom processing.

The *getProdScoresAndTrim()* method is used to turn the Hits object into an ArrayList of Document objects, one for each hit. It also creates a HashMap of scores that is used to order the results by score.

The key method that sub-classes must implement is the *process()* method. Its purpose is to turn the Hits into a list of objects that can be rendered on the JSP page: typically, this is an ArrayList of data beans or presentation beans.

Here is a simplified example of what the process method does:

```
public List process(Hits hits, int pageNum, SessionContext cntx)
    throws ICCException, IOException
{
    this.indexSet = cntx.getIndexSet();
    Long stopCount = cntx.getStopCount().longValue()
    ArrayList docs = super.getProdScoresAndTrim(hits, stopCount);
    super.setTotalBeforeStopCount(hits.length());
    super.setTotalAfterStopCount(docs.size());
    super.setCurPageNum(pageNum);
    // Sort the result.
    this.doSort(docs, super.getSortCriteria(),
        super.isSortAscending());
    // gets price stored in the index.
    this.getProductPrices(docs);
    // get results for the page
    ArrayList pageDocs = getPage(docs, pageNum, super.getPageSize());
    // get product ids
    SearchFieldConfiguration sconf =
        this.indexSet.getSearchFieldConfiguration("prodID");
    ArrayList productIDs =
        this.getProductIDs(pageDocs, sconf.getKey());
    // create query
    DsQuery query = getProductQuery(productIDs);
    // restore products based on the query
    IBizProductList pl = restoreProducts(super.getDc(),
        query, super.isShowPromotions());
    // Need to re-order results returned from query
    List pList = pl.getProducts();
    pList = filterByEffectivity(pList);
    setPrices(pList);
```

```
    pList = reorderResults(productIDs, pList);
    return pList;
}
```

Typically, a controller constructs the appropriate results builder, uses it to processes the results, and then passes off the result to the JSP page:

```
SearchResultBuilder temp_ResultBuilder = new ProductListBuilder();
List resultList = temp_ResultBuilder.process(hits, pageNumber, cntx);
request.setAttribute("products", resultList);
```

Customizing IndexBuilders

Typically, you can customize index building in these ways:

* Modify the configuration using the configuration files and creating new index builders.

* Create complex search terms using the query, filter, and filtering query search term types.

* Apply filters that limit the objects to be indexed: building filters that perform complex comparisons of terms.

* Add or remove fields to be indexed: modifying the **SearchConfigurationProperties.xml** file to add new index fields and corresponding index builders for the new fields.

* Customize the stemming and parsing logic: either extend the existing CatalogSearchAnalyzer class or create a new Analyzer class that extends the Lucene Analyzer class.

CHAPTER 27 — *Web Services*

The Comergent eBusiness System provides a set of Web services that can be used to execute Comergent eBusiness System functionality from remote clients. Release 7.0 and higher provide an extensible framework in which SOAP messages can be handled, and some specific Web services that can be used out-of-the-box. This chapter covers:

- "Overview" on page 321
- "Web Services Provided by Comergent eBusiness System" on page 324
- "Creating a Web Service" on page 329

The Web services also provide the basic capability to provide portlets that display the data returned by the Web services. See the *Comergent eBusiness System Implementation Guide* for details.

Overview

Web services provide a framework that enables Web applications to expose functionality in a standard way to remote clients. They use a combination of XML and HTTP to communicate between client and server. Most Web services use the Simple Object Access Protocol (SOAP) to exchange the remote procedure calls, although other protocols can also be supported.

The basic rules in writing a new Web service and WSDL Interface definition are:

1. The input message should have a single part and is an element.

2. The Element should have the same name as the operation name.

3. The elements complex type should not have attributes.

4. The response element should be operationName appended with "Response".

WSDL Files

When a Web application provides a Web service, it declares them using a Web Services Description Language (WSDL) file: this is an XML file that describes the Web service: it specifies the location of the Web service and the methods that it exposes. For example, the following fragment of a WSDL file is used to obtain a stock quote:

```
<portType name="StockQuotePortType">
    <operation name="GetLastTradePrice">
        <input message="tns:GetLastTradePriceInput"/>
        <output message="tns:GetLastTradePriceOutput"/>
    </operation>
</portType>
```

In this example, the Web service, StockQuotePortType, supports a method called GetLastTradePrice that takes one parameter, an object of type GetLastTradePriceInput, and it returns an object of type GetLastTradePriceOutput. The precise structure of the GetLastTradePriceInput and GetLastTradePriceOutput types are also defined in the same WSDL document.

Web Service Clients

In order to access the Web service, you must write a client that generates SOAP messages and sends them to the Web application in such a way that the Web application can process the messages and respond. The client can be written in any programming language, provided that it can correctly issue and receive SOAP messages.

Writing the client entirely by hand could be a laborious task: fortunately, tools are available to generate the client code. The tools use the WSDL file to generate the client "stub" classes that can handle the SOAP communication between client and Web application, and so all you need to do is to write the application logic for the client.

Most modern integrated development environments (IDEs) such as Eclipse and Visual Studio support the generation of client classes, and an open source software project, AXIS, can also be used.

Typically, you use the automatically-generated "locator class" to instantiate a client class that can invoke the Web service, and then the client class can invoke one of the Web service operations in way which is entirely invisible to the client application.

Example

For example, the **OrderInterface.wsdl** WSDL file provided by the Comergent eBusiness System contains the following XML fragments:

```
<xsd:element name="OrderCreateRequest">
    <xsd:complexType>
    <xsd:sequence>
    <xsd:element name="MessageHeader" type="tns:MessageHeaderType" />
    <xsd:element name="RemoteUser" type="tns:RemoteUserType" />
    <xsd:element name="Order" type="tns:OrderCreateParams" />
    </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

If you run one of the Java-based tools against this WSDL, then you will see that (among others) the following classes are generated:

- OrderCreateServiceLocator.java

- OrderCreateParams.java

- OrderCreatePortType.java

- OrderCreateRequest.java

- OrderCreateResponse.java

By using these classes in your client application, you can invoke the Web service along these lines:

```
try
{
    OrderCreateServiceLocator temp_OrderCreateServiceLocator =
        new OrderCreateServiceLocator();
    OrderCreatePortType temp_OrderCreatePortType =
        temp_OrderCreateServiceLocator.getOrderCreatePort();
    MessageHeaderType temp_MessageHeaderType =
        new MessageHeaderType();
    MessageTypeType temp_MessageTypeType =
        MessageTypeType.fromString("dXML");
    temp_MessageHeaderType.setMessageType(temp_MessageTypeType);
    MessageVersionType temp_MessageVersionType =
        MessageVersionType.fromString("5.0");
    temp_MessageHeaderType.setMessageVersion(
```

```
        temp_MessageVersionType);
    temp_MessageHeaderType.setMessageID("Fred");
    RemoteUserType temp_RemoteUserType = new RemoteUserType();
    temp_RemoteUserType.setUserLogin("ERPAdmin");
    temp_RemoteUserType.setUserAuthenticator("ERPAdmin");
    OrderCreateParams temp_OrderCreateParams =
        new OrderCreateParams();
    temp_OrderCreateParams.setCurrencyCode("USD");
    ... set the values of other order fields
    DXMLOrderType temp_DXMLOrderType =
        temp_OrderCreatePortType.orderCreateRequest(
        temp_MessageHeaderType,
        temp_RemoteUserType, temp_OrderCreateParams);
    OrderResponseType temp_OrderResponseType =
        temp_DXMLOrderType.getOrder();
    OrderCreateResponse temp_OrderCreateResponse =
    temp_OrderCreatePortType.orderCreate(temp_OrderCreateRequest);
}
catch (Exception e)
{
    System.out.println("Throwing exception " + e.toString());
    e.printStackTrace(System.out);
}
...
```

The key lines are the ones that create the OrderCreateServiceLocator locator class, use the locator class to create the OrderCreatePortType, create the OrderCreateParams class, and then the line that invokes the service and receives back an instance of the DXMLOrderType class. Note the use of a try-catch block to capture any problems asscociated with invoking the Web service. The SOAPFault mechanism is used to capture any server-side errors and using the *printStackTrace()* method on the exception will help you to identify the cause of an error.

Web Services Provided by Comergent eBusiness System

This release of the Comergent eBusiness System provides the following Web services:

- "Catalog Management" on page 326

- "Invoice Management" on page 326

- "Lead Management" on page 326

- "OIL Management" on page 326

- • "Order Management" on page 327
- • "Partner Management" on page 327
- • "Promotion Management" on page 327
- • "Proposal Management" on page 327
- • "Quote Management" on page 328
- • "Return Management" on page 328
- • "Sales Contract Management" on page 328
- • "Task Management" on page 328
- • "User Management" on page 329

The Web services are declared in the Interface WSDL files to be found in the *debs_home*/**Comergent/dXML/5.0/** directory:

- • **CatalogInterface.wsdl**
- • **InvoiceInterface.wsdl**
- • **LeadInterface.wsdl**
- • **OILInterface.wsdl**
- • **OrderInterface.wsdl**
- • **PartnerInterface.wsdl**
- • **PromotionInterface.wsdl**
- • **ProposalInterface.wsdl**
- • **QuoteInterface.wsdl**
- • **ReturnInterface.wsdl**
- • **SalesContractInterface.wsdl**
- • **TaskInterface.wsdl**
- • **UserInterface.wsdl**

Before you notify your customers of the Web services, be sure to set the endpoints for the Web services through the Comergent eBusiness System System Services UI. For example, suppose that the main URL used to access the Comergent eBusiness System is:

```
http://www.matrix.com:8080/Comergent/en/US/enterpriseMgr/matrix
```

Then you must set the **System Services -> XML Messages -> Message URL** for SOAP Messages to:

`http://www.matrix.com:8080/Comergent/msg/matrix/soap`

Catalog Management

You can access the following Catalog-related services:

• ProductGet

• ProductSearch

You can access the Catalog Management WSDL at:

`http://<server:port>/Comergent/dXML/5.0/CatalogInterface.wsdl`

Invoice Management

You can access the following Invoice-related services:

• InvoiceChange

• InvoiceCreate

• InvoiceGet

• InvoiceListGet

You can access the Invoice Management WSDL at:

`http://<server:port>/Comergent/dXML/5.0/InvoiceInterface.wsdl`

Lead Management

You can access the following Lead-related services:

• LeadGet

• LeadSearch

You can access theLead Management WSDL at:

`http://<server:port>/Comergent/dXML/5.0/LeadInterface.wsdl`

OIL Management

You can access the following OIL-related services:

• OILCreate

• OILGet

• OILSearch

- RoutedOILSearch

You can access the OIL Management WSDL at:

`http://<`*`server`*`:`*`port`*`>/Comergent/dXML/5.0/OILInterface.wsdl`

Order Management

You can access the following Order-related services:

- OrderCreate
- OrderGet
- OrderSearch
- OrderStatus

You can access the Order Management WSDL at:

`http://<`*`server`*`:`*`port`*`>/Comergent/dXML/5.0/OrderInterface.wsdl`

Partner Management

You can access the following Partner-related services:

- PartnerCreate
- PartnerChange

You can access the Partner Management WSDL at:

`http://<`*`server`*`:`*`port`*`>/Comergent/dXML/5.0/PartnerInterface.wsdl`

Promotion Management

You can access the following Promotion-related services:

- PromotionGet
- PromotionSearch

You can access the Promotion Management WSDL at:

`http://<`*`server`*`:`*`port`*`>/Comergent/dXML/5.0/PromotionInterface.wsdl`

Proposal Management

You can access the following Proposal-related services:

- ProposalCreate
- ProposalSearch

You can access the Proposal Management WSDL at:

`http://<`*`server`*`:`*`port`*`>/Comergent/dXML/5.0/ProposalInterface.wsdl`

Quote Management

You can access the following Quote-related services:

• QuoteGet

• QuoteSearch

You can access the Quote Management WSDL at:

`http://<`*`server`*`:`*`port`*`>/Comergent/dXML/5.0/QuoteInterface.wsdl`

Return Management

You can access the following Return-related services:

• ReturnGet

• ReturnSearch

You can access the Return Management WSDL at:

`http://<`*`server`*`:`*`port`*`>/Comergent/dXML/5.0/ReturnInterface.wsdl`

Sales Contract Management

You can access the following Quote-related services:

• SalesContractGet

• SalesContractSearch

You can access the Sales Contract Management WSDL at:

`http://<`*`server`*`:`*`port`*`>/Comergent/dXML/5.0/SalesContractInterface.wsdl`

Task Management

You can access the following Task-related services:

• TaskGet

• TaskSearch

You can access theTask Management WSDL at:

`http://<`*`server`*`:`*`port`*`>/Comergent/dXML/5.0/TaskInterface.wsdl`

User Management

You can access the following User-related services:

• UserGet

• UserUpdate

You can access the Shopping Cart Management WSDL at:

`http://<server:port>/Comergent/dXML/5.0/UserInterface.wsdl`

Common Components

These files make use of common definitions defined in the dXML schema definition XSD files:

• **dXML-BasicComponents.xsd**

• **dXML-CatalogObjectDefinitions.xsd**

• **dXML-InvoiceObjectDefinitions.xsd**

• **dXML-OILObjectDefinitions.xsd**

• **dXML-OrderObjectDefinitions.xsd**

• **dXML-PartnerObjectDefinitions.xsd**

• **dXML-ProposalObjectDefinitions.xsd**

• **dXML-ReturnObjectDefinitions.xsd**

• **dXML-UserObjectDefinitions.xsd**

The **dXML-BasicComponents.xsd** XSD file defines elements that are common to all the Web services supported by the Comergent eBusiness System, and the other XSD files are specific to each of the Interface Web services.

Creating a Web Service

This section describes how to go about creating a Web service using the Comergent eBusiness System APIs. Throughout this section we use the example of creating a Web service to provide pricing information for a user.

To create a Web service for an existing API, you must provide the following components:

- WSDL: create a WSDL which declares the form of the request and response used to invoke the Web service. The WSDL file can include XSD files that define specific types used by the WSDL.

- Business API: either identify or create the business API used to perform the business logic and retrieve the required information.

- Bizlet Class: you must implement a Bizlet class to process the inbound service request.

- Message Conversion Files: these are entries in the corresponding converter definition files (**dxml_5_0_to_comergent_4_0.xml** and **native_to_dxml_5_0.xml** for the inbound and outbound messages resepctively), and the stylesheets used to convert the messages.

The following diagram provides a picture of how the components work together:
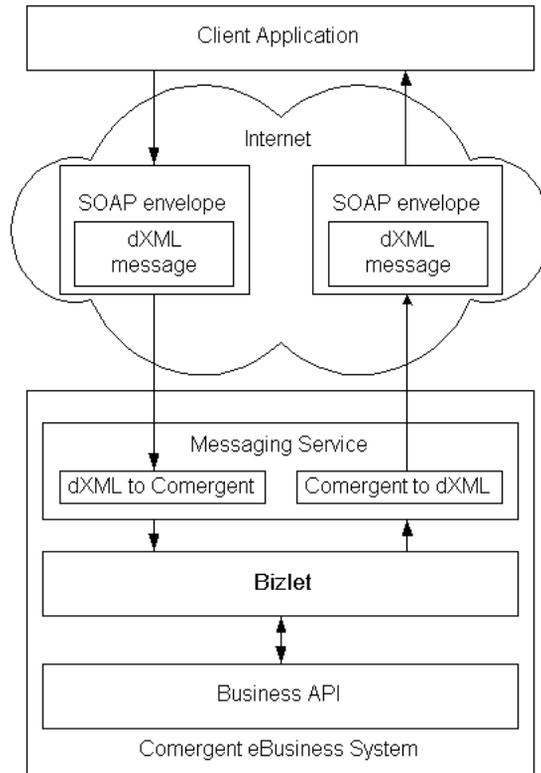
FIGURE 10. Comergent eBusiness SystemWeb Services

In developing your Web service, you will find it very helpful to have the following components set up:

• Logging set to debug in the business API that your Web service invokes.

• Logging set to debug in the messaging layer: this will enable you to view the form of the dXML 5.0 and Comergent 4.0 XML messages, and verify that the stylesheets are correctly converting from one to the other.

• A tool such as the Axis TCPMon tool: this enables you to view the form of the XML message being sent by your client application into the Comergent eBusiness System and the form of the XML message being returned from the Comergent eBusiness System to your client application.

WSDL

You must create WSDL file that defines the Web service. This must declare the form in which the request must be packaged and the form in which the response is packaged. Using a tool such as TCPMon, you can verify that the client application is sending and receiving XML mesages that match the specification declared in the WSDL file and its included XSD files.

In our example, we declare the form of the PricingGetRequest and PricingGetRequestResponse in a **PricingInterface.wsdl** file as follows:

```
<xsd:element name="PricingGetRequest">
    <xsd:complexType>
    <xsd:sequence>
    <xsd:element name="MessageHeader" type="cmgt:MessageHeaderType"/>
    <xsd:element name="RemoteUser" type="cmgt:RemoteUserType"/>
    <xsd:element name="PriceAvailabilityRequest"
        type="cmgt:PriceAvailabilityRequest"/>
    </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name="PricingGetRequestResponse">
    <xsd:complexType>
    <xsd:sequence>
    <xsd:element name="dXML" type="cmgt:dXMLPriceAvailabilityType"/>
    </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

The names of xsd:elements will determine the names of XML elements within the XML message passing over the wire. The types must be defined in the WSDL file (or files included using the xsd:include tag). For example, the cmgt:dXMLPriceAvailabilityType is declared as:

```
<xsd:complexType name="dXMLPriceAvailabilityType">
    <xsd:sequence>
    <xsd:element name="MessageHeader" type="cmgt:MessageHeaderType"/>
    <xsd:element name="ResponseHeader"
        type="cmgt:ResponseHeaderType"/>
    <xsd:element name="PriceAvailability"
        type="cmgt:PriceAvailabilityResponseType"/>
    </xsd:sequence>
</xsd:complexType>
```

In the same way, the cmgt:PriceAvailabilityResponseType is declared with its subsidiary types as:

```
<xsd:complexType name="PriceAvailabilityResponseType">
    <xsd:sequence>
```

```
    <xsd:element name="CurrencyCode" type="cmgt:StringType"
        minOccurs="0" maxOccurs="1"/>
    <xsd:element name="CustomerType" type="cmgt:StringType"
        minOccurs="0" maxOccurs="1"/>
    <xsd:element name="PaLineList" type="cmgt:PaLineArray"
        minOccurs="0" maxOccurs="1"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="PaLineArray">
    <xsd:sequence>
    <xsd:element name="PaLine" type="cmgt:PaLine"
        minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="PaLine">
<xsd:sequence>
    <xsd:element name="SKU" type="cmgt:SKUType"/>
    <xsd:element name="Quantity" type="cmgt:QuantityType"
        minOccurs="0" maxOccurs="1"/>
    <xsd:element name="ListPrice" type="cmgt:AmountType"
        minOccurs="0" maxOccurs="1"/>
</xsd:sequence>
```

Once the WSDL and included files are writtten, you should drop them into the *debs_home*/**Comergent/dXML/5.0/** directory, and verify that you can generate client stub classes by running a Web services tool such as the Axis WSDL2Java script against this WSDL.

If you write a client application using the generated client stub classes, then you should be able to run the client application and verify that an error message is reported by the Comergent eBusiness System messaging layer (because you have not yet defined the message type that you want to use to invoke the Web service).

Business API

Web services provide a mechanism to expose some of your business logic so that it can be executed by remote clients. You may already have created the business logic or you may have to create it as part of the process of creating the Web service.

In this section, we are using the example of the business API used to retrieve prices. Specifically, we want to invoke the *check(pricing Cart pc)* method of the com.comergent.api.apps.pricingMgr.PriceCheckAPI class. To do this, our Bizlet implementation class will need to assemble a PricingCart object from the inbound message: this PricingCart encapsulates all the information that the pricing engine requires: the partner key, the customer type, the currency, and the product IDs.

Bizlet Class

CHAPTER 5, "Bizlets" provides an introduction to the use of Bizlets in the Comergent eBusiness System, and so this section concentrates on the specific implementation we use in this example. Bizlets use beans to specify their interfaces, and so we must use either a pre-existing data object definition or create one, so that the bean class is generated. The data object definition can be used to generate the corresponding Comergent 4.0 DTD: it is this DTD that determines how the stylesheet will transform the inbound dXML message into the Comergent 4.0 XML form that is used to create the data bean used by the Bizlet class.

Together with the Bizlet class, you must declare the MessageType entry in one of the **MessageTypes.xml** configuration files: this provides the mapping between the message type and the Bizlet class. In our example, we simply create an entry in the PricingAdminAndAssignGroup message group along these lines:

```
<MessageType Name="PriceAvailabilityRequest">
    <ControllerMapping>
        com.comergent.dcm.bizlet.BizRouter
    </ControllerMapping>
    <BizletMapping>
    com.comergent.apps.priceCheck.bizlet.PriceCheckBizlet.checkPrices
    </BizletMapping>
</MessageType>
```

In our example, we will use the PriceAvailability data object: its XML data object definition file is part of the Release 7.0.2 WAR release, and its DTD is generated as part of an SDK installation. The Bizlet interface we create is called PriceCheckBizlet and its corresponding implementation class is PriceCheckBizletImpl. The interface supports one method:

```
public PriceAvailabilityBean checkPrices(PriceAvailabilityBean
    priceAvailabilityBean)
```

The implementation class provides the following implementation of the *checkPrices()* method:

```
public PriceAvailabilityBean checkPrices(PriceAvailabilityBean
    priceAvailabilityBean) throws ICCException
{
    PricingCart temp_InboundPricingCart =
        convertPABeanToPricingCart(priceAvailabilityBean);
    PricingCart temp_OutboundPricingCart =
        PriceCheckAPI.check(temp_InboundPricingCart);
    convertPricingCartToPABean(temp_OutboundPricingCart,
        priceAvailabilityBean);
    return priceAvailabilityBean;
}
```

Here, helper methods are used to convert between the bean used by the Bizlet API and the PricingCart used by the PriceCheck API.

At this point, you should be able to verify that the Bizlet class is being invoked correctly, by directly posting into the Comergent eBusiness System messages that match the Comergent 4.0 representation of the bean that your Bizlet uses.

Message Conversion Files

This section covers the conversion of the inbound XML message from the client applicaiton, and the outbound message back to the client. This involves creating stylesheets, and in addition to the stylesheets, you must update the converter configuration files to declare the new stylesheets (**dxml_5_0_to_comergent_4_0.xml** and **native_to_dxml_5_0.xml** for the inbound and outbound messages respectively).

Inbound Conversion

When the client application invokes the Web service by means of the stub classes, it sends a dXML message as an inbound post into the Comergent eBusiness System. The inbound message is transformed using a stylesheet that converts the dXML into Comergent 4.0 (the "native" format) XML, and then the messaging layer can convert this into the bean that the Bizlet class is expecting. Consequently, a stylesheet has to be created to convert the inbound dXML message into the Comergent 4.0 form.

For example, to mange the conversion of the inbound PricingGetRequest, the following entry is added to the **dxml_5_0_to_comergent_4_0.xml** file:

```
<MessageType Name="PricingGetRequest">
<ConvertedMessageType>PriceAvailabilityRequest</ConvertedMessageType>
<ConverterImpl>com.comergent.converter.ConverterImpl</ConverterImpl>
<Stylesheet>
    /WEB-INF/stylesheets/dXML50toCmgt40_PriceAvailability_Req.xsl
</Stylesheet>
</MessageType>
```

In the **dXML50toCmgt40_PriceAvailability_Req.xsl** stylesheet file, the following element is used to convert the MessageHeader element of the inbound dXML message into the corresponding MessageHeader element of the Comergent 4.0 message:

```
<xsl:template match="cmgtws:MessageHeader">
    <xsl:variable name="messageType"
        select="'PriceAvailabilityRequest'"/>
    <xsl:variable name="messageVersion" select="'4.0'"/>
    <MessageHeader>
```

```
        <MessageType>
            <xsl:value-of select="$messageType"/>
        </MessageType>
        <MessageVersion>
            <xsl:value-of select="$messageVersion"/>
        </MessageVersion>
        <MessageID>
            <xsl:value-of select="/cmgtws:PricingGetRequest/
cmgtws:MessageHeader/cmgtws:MessageID"/>
        </MessageID>
        <SessionID>
            <xsl:value-of select="/cmgtws:PricingGetRequest/
cmgtws:MessageHeader/cmgtws:SessionID"/>
        </SessionID>
    </MessageHeader>
</xsl:template>
```

Similarly, the following element converts the PriceAvailabilityRequest element:

```
<xsl:template match="cmgtws:PriceAvailabilityRequest">
    <PriceAvailability type="BusinessObject" state="INSERTED">
        <CurrencyCode>
        <xsl:value-of select="/cmgtws:PricingGetRequest/
cmgtws:PriceAvailabilityRequest/cmgtws:CurrencyCode"/>
        </CurrencyCode>
        <OrderType>
        <xsl:value-of select="/cmgtws:PricingGetRequest/
cmgtws:PriceAvailabilityRequest/cmgtws:CustomerType"/>
        </OrderType>
    <PaLineItemList state="INSERTED">
    <xsl:for-each select="/cmgtws:PricingGetRequest/cmgtws:PriceAv-
ailabilityRequest/cmgtws:PaLineList/cmgtws:PaLine">
    <PaLineItem>
        <SellerSKU>
            <xsl:value-of select="cmgtws:SKU"/>
        </SellerSKU>
        <BuyerSKU>
            <xsl:value-of select="cmgtws:SKU"/>
        </BuyerSKU>
        <Quantity>
            <xsl:value-of select="cmgtws:Quantity"/>
        </Quantity>
    </PaLineItem>
    </xsl:for-each>
    </PaLineItemList>
    </PriceAvailability>
</xsl:template>
```

Outbound Conversion

The conversion from Comergent 4.0 back to dXML 5.0 is declared in the **native_to_dxml_5_0.xml**. For example:

```
<MessageType Name="PriceAvailabilityReply">
    <ConvertedMessageType>PricingGetResponse</ConvertedMessageType>
    <ConverterImpl>com.comergent.converter.ConverterImpl</Converter-
Impl>
    <Stylesheet>
    WEB-INF/stylesheets/NativetodXML50_PricingGet_Rep.xsl
    </Stylesheet>
</MessageType>
```

The stylesheet uses the following elements to convert the Comergent 4.0 PriceAvailability element:

```
<xsl:template match="PriceAvailability">
    <PriceAvailability>
    <xsl:call-template name="PriceAvailability"/>
    <xsl:apply-templates select="PaLineItemList" />
    </PriceAvailability>
</xsl:template>
<xsl:template name="PriceAvailability">
<CurrencyCode>
<xsl:value-of select="/Comergent/PriceAvailability/CurrencyCode"/>
</CurrencyCode>
<CustomerType>
<xsl:value-of select="/Comergent/PriceAvailability/OrderType"/>
</CustomerType>
</xsl:template>
<xsl:template name="PaLineItemList" match="PaLineItemList">
    <PaLineList>
         <xsl:for-each select="PaLineItem">
         <PaLine>
         <SKU>
             <xsl:value-of select="SellerSKU"/>
         </SKU>
         <Quantity>
             <xsl:value-of select="Quantity"/>
         </Quantity>
         <ListPrice>
             <xsl:value-of select="ListPrice"/>
         </ListPrice>
         </PaLine>
         </xsl:for-each>
    </PaLineList>
</xsl:template>
```

At this point, you should be able to verify that the Bizlet class is being invoked correctly, by directly posting into the Comergent eBusiness System messages that match the dXML 5.0 form of the message that your Web service uses. If this succeeds, then you should be able to test the system end-to-end by running your client application class.

CHAPTER 28   *Maintaining History for Data Objects*

This chapter describes how to keep track of the history of a data object as it is created and modified. This information can be useful to users to understand what changes have been made to objects such as order, invoices, quotes, and so on. In addition, the history gives an audit trail of changes in cases where it becomes critical to understand which user made which changes to an object.

Framework

The recommended approach is to save the history of changes to a data object by using a history data object that takes a "snapshot" of the data object, and then persisting the history data object to the Knowledgebase. Before saving the snapshot you can add or remove information that you do not need to record from the history data object. Typically, the orginal data object has a field that acts as its unique unique key: this key can be used to retrieve all of the history data objects that relate to each original data object.

The approach taken is to define the history data object by extending the orginal data object. Using the *copyBean()* method to copy the data from the orginal bean to the history data bean is possible because the history data bean class extends the original bean class. Depending on what history is required to save, you can have additional fields in the history data object which should be populated before calling *persist()* on the history data bean.

Example

In this example, we use the OrderLine Item as the data object whose history we want to record. So assume that there is an **OrderLineItem.xml** file defining the structure of the data object, and that a corresponding simple bean and list bean have been generated: OrderLineItemBean.java and OrderLineItemListBean.java.

Here is a simplified form of the data object file:

```
<DataObject Extends="OrderInquiryListLineItem" Name="OrderLineItem"
    ObjectType="JDBC" Version="6.0">
    <DataFieldList>
        <DataField ExternalFieldName="LIST_PRICE" Mandatory="n"
            Name="OrderListPrice" Writable="y"/>
        ...
    </DataFieldList>
</DataObject>
```

1. Define a new data object to capture line item history. This should extend the OrderLineItem data object to keep it forwardly compatible with OrderLineItem across any changes. For example:

```
<DataObject Name="OrderLineItemHist" Extends="OrderLineItem"
    ExternalAlias="LI" ExternalName="CMGT_OIL_LI_H"
    ObjectType="JDBC" Version="6.0">
    <KeyFields>
        <KeyField Name="HistoryLineKey"
            ExternalName="HISTORY_LINE_KEY"
            KeyGenerator="HistoryLineKey"/>
    </KeyFields>
    <DataFieldList>
        <DataField Name="HistoryLineKey"
            ExternalFieldName="HISTORY_LINE_KEY"
            Mandatory="n" Writable="y"/>
    </DataFieldList>
</DataObject>
```

Note that in this example, the OrderLineItemHist data object uses a different table than the OrderLineItem. This is not strictly necessary. There are some additional data fields which may be relevant for the history capture. It defines a key field that uniquely identifies this version of the line history.

2. Make the relevant entries in the **DsDataElements.xml**, **DsBusinessObjects.xml**, and **DsRecipes.xml** files.

3. Run the bean generator to generate the data beans and their interfaces.

4. In the application that creates and modifies the OrderLineItem data objects, add the following to capture history. It uses the *copyBean()* method to transfer the data from the OrderLineItemBean class to the corresponding OrderLineItemHistBean class. Assume that you have an instance, olib, of the order line item bean for which you want to capture history:

```
OrderLineItemBean olib = instance of OrderLineItemBean
//Create an instance of the History bean
OrderLineItemHistBean newBean = (OrderLineItemHistBean)
    OMWrapper.getObject(OrderLineItemHistBean.class.getName());
//Call copyBean method to copy to new bean
olib.copyBean(newBean);
// Optional step to remove shipping information
int cnt1 = newBean.getOrderLineItemShipInfoCount();
Vector v1 = newBean.getOrderLineItemShipInfoVector();
for (int i = 0; i < cnt1; i++)
{
    OrderLineItemShipInfoBean foo1 =
        (OrderLineItemShipInfoBean) v1.elementAt(i);
    foo1.prune();
}
// Optional step to remove configuration information
int cnt2 = newBean.getCartConfigurationCount();
Vector v2 = newBean.getCartConfigurationVector();
for (int i = 0; i < cnt2; i++)
{
    CartConfigurationBean foo2 =
        (CartConfigurationBean) v2.elementAt(i);
    foo2.prune();
}
//Persist this new history bean
newBean.persist();
```

CHAPTER 29 *Coding Conventions*

The following conventions are Comergent Technologies coding conventions. By following them, you ensure that your code can be maintained easily.

Using Session and Cache Objects

When writing applications for the Comergent eBusiness System, bear in mind that you have the following available mechanisms for storing information:

• Using the Web Application: storing information on the server running the Comergent eBusiness System.

• Using the Client Application: storing information on the user's browser.

Using the Web Application

ComergentSession
Use the ComergentSession object when you need to persist information from one request of a user's session to another. If you are using a clustered installation of the Comergent eBusiness System, then you must use a ComergentSession and its *setAttribute()* method to ensure that data can be retrieved from one request to the next. See "Wrapper Classes" on page 18 for more information about ComergentSession. Objects stored in a ComergentSession must be serializable.

- Use *setAttribute(String s, Object o)* when the data must be persistent for the lifetime of the session. The object must be small and serializable. Retrieve the object with *getAttribute(String s)*.

- Use *cache(String s, Object o)* when the data is sufficiently recoverable if it is garbage collected. The object need not be serializable and will not be available to other members of a cluster. There is no constraint on its size. Retrieve the object with *retrieve(String s)*.

GlobalCache

Data stored in the GlobalCache is available to all applications running as part of the Comergent eBusiness System. It is also available to other machines running as part of the same cluster. See the *Comergent eBusiness System Implementation Guide* for implementation steps to support clustering. You should bear in mind the possibility of another application using the same String as a key.

Information stored in the GlobalCache is subject to garbage collection. If you store an object in the GlobalCache using *cache()*, then you must check for null when you retrieve the object, and you must be able to re-create the object if it is no longer in the cache. See "GlobalCache Interface" on page 28 for more information regarding the methods available to store and retrieve information in the GlobalCache.

- Use *cache(String s, Object o)* when the data is volatile and may be recovered if garbage collected. Objects need not be serializable and are not restricted in size. Retrieve the object with *get(String s)*.

- Use *set(String s, Object o)* when the data must be safe from garbage collection. This method is deprecated. Objects need not be serializable and are not restricted in size. Bear in mind that the object will not be available to other members of a cluster. Retrieve the object with *get(String s)*.

In general, you should try to use the ComergentSession for data that must persist from one request to another and which must be available irrespective of which machine in a clustered deployment of the Comergent eBusiness System is serving a request.

For performance reasons, you should otherwise limit your use of ComergentSession and use the GlobalCache mechanism for other data. However, bear in mind the cost of retrieving an object if it has been removed from the GlobalCache before you want to re-use it.

ComergentContext

You can access the ComergentContext to store server-wide data that needs to be accessed by different Comergent eBusiness System applications. Objects stored in

the ComergentContext are not shared across a cluster. There is no restriction on the size of objects stored in the ComergentContext. In general, you should its *setAttribute()* and *getAttribute()* methods to manage storing and retrieving data. Data stored in the ComergentContext is not garbage-collected, so bear in mind memory-consumption issues.

Using the Client Application

In addition to storing data as part of the Web application, you can also store data in a user's Web browser so that it can be retrieved as part of the execution of a user's request.

Form Data

When you generate a Web page as part of a Comergent eBusiness System application, you can store data that you would like to have returned as part of the request using a hidden input field in a form or as a URL parameter. For example:

```
<FORM ACTION="http://<server:port>/Comergent/partnerMkt/
matrix?cmd=CartDisplay>
<INPUT TYPE="HIDDEN" NAME="cartKey" VALUE="17">
</FORM>
```

or

```
http://<server:port>/Comergent/partnerMkt/matrix?cmd=CartDis-
play&cartKey=17
```

Note that users can edit the values of hidden input variables or URL parameters before posting a form request or URL, and so your application must verify that the variables are returned unchanged or check for access entitlements.

Cookies

You can store data as a cookie in the user's browser. Note that if a user has disabled the use of cookies in their browser, then any application that relies on the use of cookies may break. For this reason, we recommend against using cookies unless you can safely require that users must enable cookies in their browser. In addition, cookies may be hacked so you should not store user-sensitive or application-critical data in a cookie.

File Access

You should write your applications so that they are ready to support the deployment of the Comergent eBusiness System in a clustered environment. In particular, make sure that when your application accesses files that it uses methods that will ensure

that the location of the files is independent of which server in a cluster is making the access. Use the following classes and methods:

• ComergentContext provides the *getResourceAsStream()*: this method provides read-only access to a file as a stream. See "ComergentContext" on page 19 for more information.

• ComergentAppEnv provides the *adjustFileName()* method: this method must be used in its four-parameter form. See "ComergentAppEnv Class" on page 27 for more information.

Naming Conventions

These naming conventions are useful to improve the readability of the code.

TABLE 15. Naming Conventions

Type	Description	Examples
Packages	Base packages should start at the com.comergent level, followed by the product and then module group name. All components of package names should be lower-case.	com.comergent.dcm.protocol
Classes and Interfaces	Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Try to keep class names simple and descriptive. Interface names should begin with "I".	DataManager ProductBean IAccProduct IChannelCart
Constants (static final variables)	Constants should be all uppercase with words separated by underscores ("_").	REPLY_HEADER REPLY_STATUS_CODE
Methods	Method names should be verbs and be in mixed case with the first letter lowercase, and the first letter of each internal word capitalized. (This rule also applies to variables and parameters.)	listen() parseProps(DcmsEnv env)

TABLE 15. Naming Conventions (Continued)

Type	Description	Examples
Member Variables	Member variables must start with "m_" and should be short yet meaningful.	private int m_statusCode
Variables and Parameter	Variable names should be intuitive to the casual reader. One-character variable names should be avoided.	int idx long sessionID String tmp

Source File Organization

Each Java source file should contain a single class or interface. Avoid associating more than one class or interface in one file except inner classes. (Inner classes should be avoided in general.)

Package Organization

You should follow the overall structure of the Comergent eBusiness System source code in which a distinction is made between API classes, implementation classes, and reference classes. See CHAPTER 11, "Modularity and Generated Interfaces" for more information.

Source Files

Java source files should observe the following ordering:

- Standard file comment template

- Package statement

- Import Statements

- Class or Interface Declarations

Import Statements

Try to import only the packages and classes that are needed, packages imported should be ordered from the most fundamental package to more specific ones. For example:

```
import java.io.*;
import java.util.*;
import com.ibm.xml.parser.*;
import com.comergent.dcm.DataServices.DsElement;
```

For classes that import many packages, a blank line can be used to separate package groups.

Class or Interface Declarations

The general organization of classes should be as follows:

- Class/interface documentation comment that complies with the Javadoc syntax (that is, /**...*/).

- Class or interface statement.

- Class or interface implementation comment, if necessary. This comment should be included where there is class-specific information, such as implementation or algorithmic details that are not appropriate for formal documentation.

- Constants, class variables, and the keyword static, should start the statements (for example, static public final int OBJECT_NOT_FOUND_ERROR = 404;).

- Member variables, organized in a semantically meaningful way.

- Constructor(s) of the class.

- Methods, organized in a semantically meaningful order.

Style and Presentation

Comment style Javadoc comments should be provided for all classes and methods, including private and package private entities. Additional documentation, such as @see, @param, @return and other tags should be used where additional values can be provided.

For comments within methods, C++ comment style: //, should be used as opposed to Standard C comment style, that is /* ... */. The C comment style should be used only to comment out code segments (and for the file comment template).

Comments should be indented to the level of the code that follows. There should be a blank line before every comment and a blank line after every multi-line comment. Trailing comments should be avoided.

There should be two blank lines between methods, and no blank lines between method descriptions and bodies.

Example of method description and body:

```
/**
```

```
 * This method returns the width of the object.
 */

public int getWidth()
{
    return this.m_width;
}

/**
 * This method returns the y coordinate of the object.
 */

public int getY()
{
    return this.m_y;
}
```

Comergent Tag Library

This chapter describes the Comergent Technologies tag library that may be used with Web pages served by the Comergent eBusiness System.

Overview

A servlet-based product can use the technology of Java Server Pages (JSP) to serve content to users. A JSP page comprises a mixture of standard HTML content and special "tags" that are processed dynamically as the page content is generated. JSP tags may be used to generate HTML dynamically and to control the flow of the page based on attributes of the tag.

Comergent eBusiness System provides some custom tags in a tag library. The tag library is declared in a Tag Library Descriptor (TLD) file, **cmgt.tld**, found in the **cmgt-taglibs.jar** file of your installation. The following tags are used:

- encode Tag (deprecated)

- frame Tag (deprecated)

- getAttribute Tag (deprecated)

- getEscaped Tag (deprecated)

- getPrice Tag (deprecated)

- getProperty Tag (deprecated)

- getResource Tag (deprecated)

- if Tag (deprecated)

- ifResource Tag (deprecated)

- link Tag (deprecated)

- list Tag (deprecated)

- paramtext Tag

- text Tag

- url Tag (deprecated)

- widget Tag (deprecated)

We refer you to standard JSP books for more detail regarding the creation and use of JSP tags. This chapter is limited to providing a description of the custom tags provided by the Comergent eBusiness System.

General Usage

You must use the jsp:useBean tag as follows:

1. The controller class that is used to process the request creates beans that are set as attributes. Typically the bean is an attribute of the incoming request or is set in the current session. However, it is possible to set a bean as an attribute in the page or context.

2. The tag retrieves the bean through the name attribute of the tag.

3. The tag uses the bean to determine its actions; such as looping through a set of objects or evaluating the truth or falsity of a test condition.

Each JSP page used by the Comergent eBusiness System must declare all the beans passed into the JSP page with the standard <jsp:useBean> tag at the top of the page. These declarations must follow the page and taglib statements.

You must use the fully-qualified class name for each bean; do not rely on import statements to provide package information. Note that if you have not passed the JSP page a bean of the appropriate id, then the JSP page creates a bean of the specified type.

For example, to declare a product list bean whose id is "productList" and that has been set as an attribute of the request, enter the following at the start of the JSP page:

```
<jsp:useBean id="productList"
    class="com.comergent.apps.catalog.bean.entity.ProductList"
    scope="request" />
```

Tag Library

This section describes all the custom tags used by the Comergent eBusiness System. The tag attributes are described, and we include an example of their use.

encode Tag

Use this tag to encode the body content to be HTML and Javascript friendly.

Attributes

- type = "HTML | JavaScript" (required)

- convertSpace = "true | false": this optional attribute determines whether to convert a space to   (ASCII code 160). Note that if convertSpace is set to true, then the returned string may not be equal to the original string because space characters (ASCII code 32) are changed to . The default value is false (do not convert).

  If a string has one space between words, then the space is preserved. However, two or more spaces are converted to a space followed by the appropriate number of . Thus, "one space" is converted to "one space", whereas "two spaces" is converted to "two spaces".

Usage

HTML pages use a specific set of characters in them. For example, "<" to denote the start of a tag, ">" to end a tag, and so on. When you want to include HTML special characters in strings, you must encode the string in order to avoid breaking the page. Furthermore, Javascript string literals do not support '\n', '\r', ' (single quote), and " (double quote) characters. These characters need to be escaped whenever they occur in Javascript strings.

The following examples illustrate different ways to use the tag. We use SKU name in these examples because it is frequently system-generated and can contain any character.

1. Suppose that you need to put the SKU name, which is a property string, in a form hidden variable.

   ```
   <INPUT type="hidden" name="skuName" value='<cmgt:encode
   type="HTML"><jsp:getProperty name="currSku" property="SKUName"/>
   </cmgt:encode>'/>
   ```

Note that convertSpace is not set here.

2. Suppose that you need to put an attribute string in a hidden variable.

```
<cmgt:encode type="HTML"><%= request.getAttribute("SkuName")%>
</cmgt:encode>
```

3. Suppose that you need to put SKU name in a HTML table cell to display.

```
<TD>SKU Name: <cmgt:encode type="HTML" convertSpace="true">
<jsp:getProperty name="currSku" property="SKUName"/>
</cmgt:encode>
</TD>
```

Note that convertSpace is set to true here.

The general rule is that if you want to display a string, then set convertSpace to "true". If the value needs to be sent to the Comergent eBusiness System, then set it to "false".

4. Suppose that you need to assign a SKU name string to a Javascript variable.

```
var skuName = '<cmgt:encode type="JavaScript"><jsp:getProperty
name="idSkus" property="SKUName"/></cmgt:encode>';
```

5. Suppose that you need to call a JavaScript function when a user clicks a SKU name hyperlink on your HTML page.

```
<A id='aItem<%=i%>' href="javascript:ProcessSku-
Click('<cmgt:encode type="HTML">
<cmgt:encode type="JavaScript">
<jsp:getProperty name="idSkus" property="SKUName"/>
</cmgt:encode>
</cmgt:encode>')">
<cmgt:encode type="HTML" convertSpace="true">
<jsp:getProperty name="idSkus" property="SKUName"/>
</cmgt:encode></A>
```

Note that in this example the string is converted first to Javascript and then to an HTML escape string.

frame Tag

This tag is used as a workaround if the servlet contiiner appends session information to a URL in the wrong place. It is principally used when a URL is built up in a Javascript function from several fragments.

Attributes

• name="frame name" (required)

• src="URL to frame content" (required)

Usage

Instead of using:

```
<frame name="Header" src="URL to header content">
```

you can use:

```
<cmgt:frame name="Header" src="URL to header content" />
```

getAttribute Tag

This tag is used to get and display an attribute from the page, request, session, or application scope. The scopes are searched in that order.

Attributes

• name="attribute name" (required)

• alt="alternate text"

Usage

To display the attribute, using its implicit *toString* method, use:

```
<cmgt:getAttribute name="thisProduct" alt="Not available" />
```

getEscaped Tag

This tag is used to retrieve a String property and to replace escape sequences to make the String Javascript friendly.

Note:	This tag is deprecated. You should use the encode tag instead.

Attributes

• name="name of bean" (required)

• property="name of the property in the bean" (required)

Usage

To retrieve a String property of a bean as an escaped String, use the tag like this:

```
<cmgt:getEscaped name="thisProduct" property="name" />
```

In this example, if the Name variable of this particular product as a Java string is '15" Monitor', then this tag returns '15\" Monitor'.

getPrice Tag

This tag is used to display the price of a named bean which may be a Product. It uses the getPrice() method of the Bean. The price is formatted for locale and currency. It also rounds numbers appropriately if they have been calculated.

Attributes

- name="name of bean" (required)

- alt="alternate text to display if price is null" (optional)

Usage

To display the price of a product that has been passed to the JSP page as a bean, use the tag like this:

```
<cmgt:getPrice name="thisProduct" alt="Price unavailable"/>
```

If the product bean has a null value for the price, then the alternate text will be displayed.

getProperty Tag

This tag is used to extend the functionality of the standard JSP getProperty tag. It enables you to specify a default value if the property value is null.

Attributes

- name="name of bean" (required)

- property="name of the property in the bean" (required)

- default="default value in case the property value is null" (optional)

Usage

Use the tag to enable the JSP page to display a default value like this:

```
<cmgt:getProperty name="thisProduct" property="description"
    default="Description not available" />
```

getResource Tag

This tag is used to get and display a resource from a named bean given the resource type. The field attribute is used to specify whether the resource value, label, or description should be displayed.

Attributes

- name="resource name" (required)

- type="type of resource" (required)

- field = "ResourceValue | ResourceLabel | ResourceDescription" (required)

Usage

To display a resource associated to a bean, use the tag like this:

```
<cmgt:getResource name="thisProduct" type="URL"
    field="ResourceValue" />
```

if Tag

A body tag which is used to include the body of the tag conditionally on the evaluation of the test condition.

Attributes

- name="bean name | attribute name" (required)

- property="bean property" (optional)

- test="defined | undefined | true | false | eq | lt | gt" (required)

- value="integer" (optional)

Usage

Extending the example of list tag, suppose that you wish to control the appearance of an HTML page by indicating whether or not a particular product is configurable. Suppose that the Product class has a boolean variable, boolean_Config with an accessor method *getConfigurable()*. Then, you can modify the JSP page along the following lines:

```
<TABLE>
<TR><TH>Name</TH><TH>Click to configure</TH></TR>
<cmgt:list id="thisProduct" name="productList" property="products">
    <!-- Body of list loop. For example: -->
    <TR>
    <TD>
    <jsp:getProperty name="thisProduct" property="name"/>
    </TD>
    <cmgt:if name="thisProduct" property="configurable" test="true">
        <TD>
        <IMAGE SRC="images/configurable.gif">
        </TD>
    </cmgt:if>
    <cmgt:if name="thisProduct" property="configurable" test="false">
        <TD>
        <IMAGE SRC="images/notconfigurable.gif">
        </TD>
```

```
    </cmgt:if>
    </TR>
</cmgt:list>
</TABLE>
```

ifResource Tag

This tag is used to test the existence or value of a resource for a bean. You use the type attribute to specify the resource type. The field attribute distinguishes between a resource's value, label, or description.

Attributes

- name="bean name | attribute name" (required)

- type="resource type" (required)

- field = "ResourceValue | ResourceLabel | ResourceDescription" (required)

- test="defined | undefined | true | false | eq | ne | lt | gt" (required)

- value="integer" (optional)

link Tag

You use this tag to generate an appropriately encoded url. If the user's browser has disabled cookies, then the session information is encoded as part of the generated URL. URL parameters are encoded to escape characters such as spaces. The URL path information is specifed as attributes of the tag whereas the body of the tag is used to pass parameters.

Note that the URL contains an identifier that is used to determine which instance of the Comergent eBusiness System should service the request. In a standard installation of the Comergent eBusiness System, this is defined as the default partner in the **Comergent.xml** configuration file. In a hosted partner implementation of the Comergent eBusiness System, the identifier distinguishes between the hosted partners.

Attributes

- app="name of the application examples: catalog, debs, and so on" (required). Note that "debs" is used to point to a special class of message types, and the syntax of the generated URLs is consequently different.

- cmd="name of the message type" (optional)

- forwardParam="a flag that is false by default, you can set it to true to forward the current request parameters to another url" (optional)

Usage

Here are some example usages of the link tag:

JSP tag	URL on the generated page
<cmgt:link app="debs" cmd="Display">UserKey=3</cmgt:link>	debs/matrix/Display?UserKey=3
<cmgt:link app="catalog" cmd="Display" forwardParam="true">UserName=Brent Wells</cmgt:link>	catalog/ matrix?cmd=Display&Username=Brent% 20Wells
<cmgt:link app="pricing" cmd="find" />	pricing/matrix&cmd=find

list Tag

A body tag used to iterate through a list of member beans, repeating the body content for each element. The counter attribute can be used to refer to an index for each member bean.

Attributes

- counter = "index" (optional)

- id = "user element name" (required): the id attribute is used within the body of the list to refer to the member bean in the list

- name = "bean name | attribute name" (required): the name attribute refers to the list bean. Typically, the list bean has been passed to the JSP page through the request or session object

- property = "bean property" (optional): the property attribute is the member variable of the list bean that accesses the list of member beans.

- type = "fully qualified name of class" (required): this is the class of the member beans. If the list bean is a generated bean, then typically this is an inner class of the list bean.

Usage

In this section, imagine that you are creating a list of products that you wish to display as a list on an HTML page. You want to generate an HTML table that lists the products one for each row of the table.

1. The controller creates the list object. This may be either:

   - an Enumeration, Hashtable, Vector, or an array

   or

> • an object that holds the list as a member variable accessible through an accessor method.

In our example, we use a class called ProductList that contains a Vector member variable, m_ProductsVector. This Vector is intended to hold one or more products. Each product is an instance of a Product class. The m_ProductsVector can be accessed through an accessor method *getProducts()*. The Product class has member variables Name and Sku accessible through standard accessor methods *getName()* and *getSku()* respectively.

Suppose that the controller creates a ProductList object called temp_ProductList. The controller sets the object as a bean in the request or session. For example:

```
getRequest().setAttribute("productList", temp_ProductList);
```

2. The list object is retrieved in the JSP page use of the list tag. In our example,

```
<TABLE>
<TR><TH>Name</TH><TH>SKU</TH></TR>
<cmgt:list id="thisProduct" name="productList" property="products"
    type="com.comergent.apps.catalog.bean.entity.Product"
    counter="count">
    <!-- Body of list loop. For example: -->
    <TR>
    <TD>
    <cmgt:getAttribute name="count"/>
    </TD>
    <TD>
    <jsp:getProperty name="thisProduct" property="name"/>
    </TD>
    <TD>
    <jsp:getProperty name="thisProduct" property="sku"/>
    </TD>
    </TR>
</cmgt:list>
</TABLE>
```

If the list object is itself an Enumeration, Hashtable, Vector, or array, then you do not have to use the property attribute in the list tag. For example, if you had set:

```
getRequest().setAttribute ("productList", m_ProductsVector);
```

then you can retrieve the list object through the tag:

```
<cmgt:list id="thisProduct" name="productList">
```

paramtext Tag

A body tag used to prepare strings used in a text tag. Typically, the values of the tag are derived from scriptlets. It must be used within the body of a text tag. Each

occurrence of the paramtext tag is stored in an array and is retrieved using the notation {0}, {1}, and so on.

Usage

The following example uses paramtext twice to manage the display of two strings in a sentence:

```
<%
    Weather tWeather = (Weather) session.getAttribute("weather");
    Date tDate = new Date();
%>
...
<cmgt:text id="*">
    <cmgt:paramtext>
        <%= DateFormat.getDateInstance().format(tDate) %>
    </cmgt:paramtext>
    <cmgt:paramtext>
        <%= tWeather.getState() %>
    </cmgt:paramtext>
    Today is {0}: it is {1} out.
</cmgt:text>
```

Here, the value of the second string is assumed to have been localized prior to having been set in the Weather object whereas the date is localized using the DateFormat class.

text Tag

A body tag used to manage the display of locale-specific text. Use it to prepare your JSP pages for localization for new locales. It is used in conjunction with resource bundles that are created and maintained for each JSP page and JSP fragment. You can maintain resource bundles manually or use the tool provided as part of the Software Development Kit.

Its behavior is governed by configuration parameters set in the **Internationalization.xml** configuration file.

Attention:	Do not use this tag in scriptlets. When you need to localize text in scriptlets, use the *cmgtText()* method. See "Included JSP Pages" on page 231 for more information.

Attributes

• bundle: specifies the resource bundle in which the display text is specified. This is primarily for use in JSP fragments which are included in multiple JSP pages.

- id: identifies a tag uniquely within a JSP page or JSP fragment.

Usage

To prepare text for locale-substitution, wrap each displayed string in the text tag:

```
<cmgt:text id="1234">Display text</cmgt:text>
```

In general, formatting tags and font tags should be outside the text tag. That is:

```
<b><cmgt:text id="1234">Display text</cmgt:text></B>
```

For each id value, make sure that you have a corresponding name-value pair in each of the supported locales properties files. For example:

```
1234 = Welcome to Matrix
```

in the **<*Name of JSP page*>_en_US.properties** file and

```
1234 = Bienvenue á Matrix
```

in the **<*Name of JSP page*>_fr_FR.properties** file.

If you plan to use the SDK tool to generate the ids automatically, then set the id to "*". If you set the id to "!", then the tag is ignored: this is useful for text that is constant in every locale: for example the "*" character that denotes required fields. You can ensure that the same text is displayed in two different places on the same JSP page by using the same value for the id attribute.

You can control the behavior of the tag using the following elements of the **Internationalization.xml** configuration file:

- debugJSPResouceBundle: a value of "true" will display error messages if the text id is not found in the resource bundle, or if the id is "*" as follows:

  ```
  ## 'Username' error: missing text for id: 'user'##
  ```

- enableJSPResouceBundle: a value of "false" will render the cmgt:text tag as a no-op: that is, the body of the tag is passed through unprocessed.

- enableJSPResouceBundleCaching: a value of "false" will make the page load the resourcebundle every time.

Use the HTML sequences for "{" and "}": these are { and } respectively.

url Tag

A body tag used to perform URL rewriting if required. If the browser has cookies disabled, then this tag re-writes the body URL to contain session information. Note that the way in which session information is encoded is servlet container-specific.

If the browser is cookie-enabled, then the URL is not encoded with the session information.

> **Note:** This tag is deprecated. You should use the link tag.

Attributes

None

Usage

To encode a URL, wrap it in the url tag:

```
<A HREF="<cmgt:url>ProductSelect</cmgt:url>">Select a product</A>
```

widget Tag

This tag is used to deploy UI widgets in Comergent eBusiness System application pages. See the CHAPTER 25, "Widgets" for more information on the use of widgets.

Attributes

- height: the height of the widget element in pixels
- name: the name of the widget is the message type executed by the Comergent eBusiness System
- width: the width of the widget element in pixels

CHAPTER 31　　*Comergent Internet Commerce Tag Library*

This chapter describes the Comergent Internet Commerce (CIC) tag library. This tag library is designed to be used as a set of UI components that can be used in JSP pages of the Comergent eBusiness System. It covers the following tags:

- "cic:banner Tag" on page 375

- "cic:checkbox Tag" on page 376

- "cic:column Tag" on page 376

- "cic:columnHeader Tag" on page 377

- "cic:command_link Tag" on page 377

- "cic:concat Tag" on page 378

- "cic:date Tag" on page 378

- "cic:div Tag" on page 378

- "cic:el Tag" on page 379

- "cic:img Tag" on page 379

- "cic:input Tag" on page 379

- "cic:inputDate Tag" on page 380

- "cic:javascriptLink Tag" on page 381

In addition to this chapter, the Javadoc provided with the SDK index documentation documents the classes that implement the tags.

Overview

We refer you to standard JSP books for more detail regarding the creation and use of JSP tags. This chapter is limited to providing a description of the custom CIC tags provided by the Comergent eBusiness System. These tags provide a set of re-usable UI components intended to support customization of Comergent eBusiness System JSP pages. These tags are declared in the **cic.tld** tag library descriptor file to be found in the **cmgt-taglibs.jar** file. A set of legacy tags to support backward-compatability with Release 6.7 is declared in the **cic67.tld** tag libray descriptor file. They can be used together with or independently from the tags declared in the **cmgt.tld** tag library descriptor file and described in CHAPTER 30, "Comergent Tag Library".

CIC tags give you the ability to create pages that look and behave similarly using the same components. For example:

• Column sorting behavior is built into the cic:column tag

• Inquiry list-like objects can be displayed using the cic:table tag

CIC tags provide common ways to pass properties and parameters to UI components which make the UI easier to maintain and customize.

CIC tags are also intended to be modelled on the JavaServer Faces (JSF) framework and consequently will make it easier both to use JSF components and to migrate pages to JSF-style presentation logic in the future.

Tag Specification

Each CIC tag is declared in the com.comergent.taglib.cic package or a sub-package. Typically, you should use tags already created in the com.comergent.taglib.cic.commerce package or create your own custom package for your UI tags.

Each CIC tag is designed to be used as a UI component whose basic structure is defined by a JSP page. The CIC tags that extend the cicComponent class have a member variable called JSPMessageType. Setting this variable to the name of the message type is what determines which JSP page is used to render the content for the tag.

Tags come in two flavors: atomic tags and component tags.

Atomic Tags

Atomic tags are used for rendering HTML that requires no customization. For example, the comand_link tag is used to generate the opening and closing tags for the anchor element: <a> and . Each atomic tag builds a formattable object and this object renders the HTML.

Component Tags

Component tags render UI components such as workspace tabs. Component tags reflect the MVC architecture in that the tag implementation creates a component model object in memory and uses a JSP page to render the view of the model as HTML. The models follow the Java Swing model pattern whereby the model holds all the data required to render the component. If the look-and-feel of a component is to be changed, then the JSP page can be modified without changing the underlying model.

An example of a component tag is the cicCommerceProductList tag: this tag specifies the "cicCommerceProductList" as the value of the JSPMessageType variable. In the **cicMessageTypes.xml** file, there is a message type element as follows:

```
<MessageType Name="cicCommerceProductList">
```

```
    <JSPMapping>../cic/cicCommerceProductList.jsp</JSPMapping>
</MessageType>
```

Thus the **cicCommerceProductList.jsp** JSP page is used to render content
whenever the cicCommerceProductList tag is used.

Table-based tags such as the cicWorkspace tag makes use of the cic:column tag.
Column tags should be used in the order that the columns are displayed in the table
and they can include atomic tags within them to display links and other content.

Nesting CIC Tags

The CIC component tags are designed to be nested. For example, the table tags
typically use column tags within them to control the display of table columns.

Customizing Tags

You can create and use custom tags in the following ways:

- Changing the Look-and-Feel

- Extending a Tag

Changing the Look-and-Feel

By modifying the JSP page of a component tag, you change the way in which the
tag renders the underlying model.

Extending a Tag

You can create a new tag by extending an existing one. Make sure that you decalre
the new tag in the **cic.tld** file. In the case of a new component tag, make sure to
create a message type and corresponding JSP page to render the HTML. The
message type to JSP page mapping should be added to the appropriate
MessageTypes.xml file.

JSP Expression Language

This release of the Comergent eBusiness System also supports the use of the JSP
2.0 expression language as specified in the JSP specification 1.2. Note that this
support is only for J2EE 1.3 or above servlet containers. See "JSP Expression
Language" on page 390 for more information.

rendered Attribute

The rendered attribute can be used to mark an element to be skipped if a condition
is not met. For example:

```
<cic:outputLink rendered="${line.promoKey != null}"
```

```
    href="javascript:displayPromo('${line.promotionCount}')">
    <cic:img src="../images/ico_sale.gif"/>
</cic:outputLink>
```

In this cic:outputLink, the link will be displayed if the promoKey is not null; that is, the link will not be displayed if the promoKey field of the line item is null.

To evaluate booleans in the rendered attribute, you must set the booleans in one of the page, request, or session contexts. For example, to use an expression such as:

```
<cic:column rendered="${isDealer || isInternal || isExternal}"
    css="left">
```

You must include code along the following lines before this tag:

```
pageContext.setAttribute("isDealer", new Boolean(isDealer));
pageContext.setAttribute("isInternal", new Boolean(isInternal));
pageContext.setAttribute("isExternal", new Boolean(isExternal));
```

General Usage

You set the CIC tags in JSP pages simply by declaring them as part of the JSP page. For example:

```
<cic:commerceProductList beanName="cartPresentation"
    formAction='<%=link("*","ShoppingCartDataDisplay")%>'>
...
</cic:commerceProductList>
```

Tag attributes serve to pass in parameters to the JSP page. For example, the beanName attribute is used to pass in the bean used to provide the data for the content of the page. Properties in the bean can be retrieved using the cic:property tag. For example:

```
<cic:commerceProductListColumn width="25%"  align="left">
    <cic:title><cmgt:text id="*">Name</cmgt:text></cic:title>
    <cic:property name="Name" />
</cic:commerceProductListColumn>
```

Example

Suppose that you would like to display a list of leads in a table that looks like this:

ID	Name	Priority	Lead Status	Created By
600501	Western Sales Conference 1	Low	Unassigned	1
600500	Table Lead	Low	Unassigned	1

FIGURE 11. CIC Tag Table Example

In HTML, the table looks like this:

```
<table cellpadding="0" cellspacing="0" border="0" width="100%">
    <tr>
        <td class="standardBackgroundColor" colspan="2" rowspan="2"
            width="-1%">
            <img src="../images/grph_crn_tplft.gif" width="2"
            height="2">
        </td>
        <td class="columnHeaderBorderColor" colspan="9" width="100%">
            <img src="../images/spacer.gif" width="1" height="1"
            border="0">
        </td>
        <td class="standardBackgroundColor" colspan="2" rowspan="2"
        width="-1%"><img src="../images/grph_crn_tprt.gif" width="2"
            height="2">
        </td>
        <td width="-1%">
        </td>
    </tr>
    <tr>
        <td class="standardBackgroundColor" colspan="9" width="-1%">
            <img src="../images/spacer.gif" width="1" height="1"
            border="0">
        </td>
    </tr>
    <tr class="standardBackgroundColor">
        <!-- a width of -1% is needed for Netscape //-->
        <td class="columnHeaderBorderColor" width="-1%">
            <img src="../images/spacer.gif" width="1" height="1"
            border="0">
        </td>
        <td class="standardBackgroundColor" width="-1%">
            <img src="../images/spacer.gif" width="1" height="1"
            border="0">
        </td>
        <td class="dataTableHeader" height="18" width="20%"
            nowrap="true" align='center'>
            <a href="javascript:sort('LeadKey');">ID</a>
        </td>
        <td class="columnHeaderBorderColor" width="-1%" >
            <img src="../images/spacer.gif" width="1" height="1"
            border="0">
        </td>
        <td class="dataTableHeader" height="18" width="20%"
            nowrap="true" align='center'>
            <a href="javascript:sort('Name');">Name</a>
        </td>
        <td class="columnHeaderBorderColor" width="-1%" >
```

```
                        <img src="../images/spacer.gif" width="1" height="1"
                        border="0">
                    </td>
                    <td class="dataTableHeader" height="18" width="20%"
                        nowrap="true" align='center'>
                        <a href="javascript:sort('LeadPriorityString');">
                        Priority</a>
                    </td>
                    <td class="columnHeaderBorderColor" width="-1%" >
                        <img src="../images/spacer.gif" width="1" height="1"
                        border="0">
                    </td>
                    <td class="dataTableHeader" height="18" width="20%"
                        nowrap="true" align='center'>
                        <a href="javascript:sort('LeadStatusString');">
                        Lead Status</a>
                    </td>
                    <td class="columnHeaderBorderColor" width="-1%" >
                        <img src="../images/spacer.gif" width="1" height="1"
                        border="0">
                    </td>
                    <td class="dataTableHeader" height="18" width="20%"
                        nowrap="true" align='center'>
                        <a href="javascript:sort('CreatedBy');">Created By</a>
                    </td>
                    <td class="standardBackgroundColor" width="-1%">
                        <img src="../images/spacer.gif" width="1" height="1"
                        border="0">
                    </td>
                    <td class="columnHeaderBorderColor" width="-1%">
                        <img src="../images/spacer.gif" width="1" height="1"
                        border="0">
                    </td>
                </tr>
                <tr>
                    <td class="standardBackgroundColor" rowspan="2" colspan="2"
                        width="-1%">
                        <img src="../images/grph_crn_btlft.gif" width="2"
                        height="2">
                    </td>
                    <td class="standardBackgroundColor" colspan="9">
                        <img src="../images/spacer.gif" width="1" height="1"
                        border="0"></td>
                    <td class="standardBackgroundColor" rowspan="2" colspan="2"
                        width="-1%">
                        <img src="../images/grph_crn_btrt.gif" width="2"
                        height="2">
                    </td>
                    <td width="-1%">
```

```
            </td>
        </tr>
        <tr>
            <td class="columnHeaderBorderColor" colspan="9">
                <img src="../images/spacer.gif" width="1" height="1"
                border="0">
            </td>
        </tr>
        <tr>
            <td width="-1%">
                <img src="../images/spacer.gif" width="1" height="1"
                border="0">
            </td>
            <td width="-1%">
                <img src="../images/spacer.gif" width="1"
                height="1" border="0">
            </td>
            <td class="dataTable" height="18" align='left'>600501 
            </td>
            <td width="-1%">
                <img src="../images/spacer.gif" width="1"
                height="1" border="0">
            </td>
            <td class="dataTable" height="18" align='left'>
                Western Sales Conference 1 
            </td>
            <td width="-1%">
                <img src="../images/spacer.gif" width="1"
                height="1" border="0">
            </td>
            <td class="dataTable" height="18" align='left'>Low 
            </td>
            <td width="-1%"><img src="../images/spacer.gif" width="1"
                height="1" border="0"></td>
            <td class="dataTable" height="18" align='left'>
                Unassigned 
            </td>
            <td width="-1%"><img src="../images/spacer.gif" width="1"
                height="1" border="0">
            </td>
            <td class="dataTable" height="18" align='left'>1 
            </td>
            <td width="-1%"><img src="../images/spacer.gif" width="1"
                height="1" border="0">
            </td>
            <td width="-1%"><img src="../images/spacer.gif" width="1"
                height="1" border="0">
            </td>
        </tr>
```

```
<tr>
    <td class="standardBackgroundColor" colspan="11">
        <img src="../images/spacer.gif" width="1" height="1"
        border="0">
    </td>
</tr>
<tr>
    <td width="-1%"><img src="../images/spacer.gif" width="1"
        height="1" border="0">
    </td>
    <td width="-1%"><img src="../images/spacer.gif" width="1"
        height="1" border="0">
    </td>
    <td class="dataTable" height="18" align='left'>
        600500 
    </td>
    <td width="-1%"><img src="../images/spacer.gif" width="1"
            height="1" border="0">
    </td>
    <td class="dataTable" height="18" align='left'>
            Table Lead 
    </td>
    <td width="-1%"><img src="../images/spacer.gif" width="1"
            height="1" border="0">
    </td>
    <td class="dataTable" height="18" align='left'>Low 
    </td>
    <td width="-1%">
        <img src="../images/spacer.gif" width="1" height="1"
        border="0">
    </td>
    <td class="dataTable" height="18" align='left'>
        Unassigned 
    </td>
    <td width="-1%"><img src="../images/spacer.gif" width="1"
        height="1" border="0">
    </td>
    <td class="dataTable" height="18" align='left'>1 
    </td>
    <td width="-1%"><img src="../images/spacer.gif" width="1"
        height="1" border="0">
    </td>
    <td width="-1%"><img src="../images/spacer.gif" width="1"
        height="1" border="0">
    </td>
</tr>
<tr>
    <td class="standardBackgroundColor" colspan="11">
        <img src="../images/spacer.gif" width="1" height="1"
```

```
            border="0">
        </td>
    </tr>
</table>
```

Using the cic:table tag, we can generate this table from the LeadListBean as follows:

```
<%
    ListIterator iterator = leadList.getLeadLightWeightIterator();
    request.setAttribute("iterator", iterator);
%>
<cic:table datasourceRef="iterator" var="lead" showSelect="false"
    sortAscending="<%= sortOrderAscending %>">
    <cic:column width="20%" sortProperty="LeadKey" align="left">
        <cic:columnHeader><cic:span value="ID"/></cic:title>
        <cic:span value="${lead.leadKey}"/>
    </cic:column>
    <cic:column width="20%" sortProperty="Name" align="left">
        <cic:columnHeader><cic:span value="Name"/></cic:columnHeader>
        <cic:span value="${lead.name}"/>
    </cic:column>
    <cic:column width="20%" sortProperty="LeadPriorityString"
        align="left">
        <cic:columnHeader><cic:span value="Priority"/>
        </cic:columnHeader>
        <cic:span value="${lead.leadPriortyString}"/>
    </cic:column>
    <cic:column width="20%" sortProperty="LeadStatusString"
        align="left">
        <cic:columnHeader><cic:span value="Lead Status"/>
        </cic:columnHeader>
        <cic:span value="${lead.leadStatusString}"/>
    </cic:column>
    <cic:column width="20%" sortProperty="CreatedBy" align="left">
        <cic:columnHeader><cic:span value="Created By"/>
        </cic:columnHeader>
        <cic:span value="${lead.createdBy}"/>
    </cic:column>
</cic:table>
```

The cic:table tag uses the datasourceRef attribute to specify which Iterator object is to be used to display rows in the table. You must set the Iterator object in the request so that the tag can retrieve it.

Each cic:column tag declares a column of the table: they specify the column heading (using the cic:title tag), and the data object property whose data should populate the column. Each cic:property name attribute must be the same as a data field name of the bean objects in the iterator. In this example, the lead data object

has fields called LeadKey, Name, and so on. The cic:whitespace tag is used to provide a little separator space around each string.

Tag Library

This section describes all the main CIC tags used by the Comergent eBusiness System. The tag attributes are described, and we include an example of their use.

cic:banner Tag

Use this tag to generate the banner at the top of HTML pages. Its attributes, renderHelp, renderHome, and so on, take the values "true" or "false" and are used to specify whether buttons are displayed for these general navigation functions.

Attributes

- helpTopic="*topic*": determines which help page is displayed. The mapping from topic to topic ID is maintained in the **HelpTopicsMap.xml** configuration file: see "Configuration Files" on page 249 for more information.

- navigationTarget="*target*": specifies the value of the target attribute of the links rendered in the banner.

- renderHeader="*true*": determines if the header is displayed.

- renderHelp="*true*": determines if **Help** button is displayed.

- renderHome="*true*": determines if **Home** button is displayed.

- renderLogout="*true*": determines if **Logout** button is displayed.

- renderWorkspace="*true*": determines if a Workspace page is to be displayed.

- workspaceTab="*Tab Header*": determines which tab is displayed.

Usage
```
<cic:banner renderHeader="true" renderHelp="true"
   helpTopic="<%=ph(helpTopic)%>">
   <cic:span css="banner"
   value="${localized['UserListHeaderText']}"/>
   <cic:el>  </cic:el>
   <cic:span rendered="${partnerInScope}" css="banner-small"
       value="${partnerScopeName}"/>
</cic:banner>
```

cic:checkbox Tag

The use of this tag is deprecated. Use this tag to display check boxes.

Attributes

- property="*Property*": specifies the value of the checkbox. The corresponding data bean must have a method called *get*Property*()*.

Usage

```
<cic:checkbox property="Property"/>
```

Example

```
<cic:checkbox property="InvoiceKey"/>
```

cic:column Tag

Use this tag to specify the column of a table.

Attributes

- align="*alignment*": takes the standard HTML alignment values of "center", "left", and "right".

- width="*width*%": the width of this column as a percentage of the width of the entire table.

Usage

```
<cic:column width="x%" align="align">
...
</cic:column>
```

Example

```
<cic:column width="17%" align="left">
    <cic:columnHeader><cic:span value="List Name"/>
    </cic:columnHeader>
    <cic:command_link target="_top">
        <cic:link app="partnerMkt" cmd="OILDisplay">
            <cic:param name="ShoppingCartKey">
                <cic:property name="ShoppingCartKey" />
            </cic:param>
        </cic:link>
        <cic:property name="Name" />
    </cic:command_link>
</cic:column>
```

cic:columnHeader Tag

Use this tag to specify what text should be displayed in the header for a table column.

Attributes

• sortProperty="*sortProperty*": the property on which to sort the table if the header of this column is clicked.

Usage

```
<cic:columnHeader sortProperty="${localized['ProductID']}">
    <cic:span value ="${localized['ProductID']}"/>
</cic:columnHeader>
```

cic:command_link Tag

The use of this tag is deprecated. Use this tag to generate a clickable link.

Attributes

• None

Usage

```
<cic:command_link>
...
</cic:command_link>
```

The value of the tag is displayed as the clickable link. Within the cic:command tag, you must specify the type of link to be used using one of:

• cic:javascriptLink

• cic:link

Example

```
<cic:command_link>
    <cic:javascriptLink methodName="processDetail" >
        <cic:param name="SKU">
            <cic:property name="SKU" />
        </cic:param>
    </cic:javascriptLink>
    <cic:property name="SKU" />
</cic:command_link>
```

This tag will be replaced by the string href="javascript:processDetail(*SKU*)", where *SKU* is determined by the use of the cic:property tag.

cic:concat Tag

Use this tag to concatenate strings. You can nest the usage of this tag to concatenate more than two strings.

Attributes

- None

Usage

```
<cic:span value="${cic:concat(cic:concat(invoice.dataBean.lastName,
', '), invoice.dataBean.firstName)}"/>
```

cic:date Tag

The use of this tag is deprecated. Use this tag to display a date. Typically, the date is retrieved using the cic:property tag from a data bean.

Attributes

- None

Usage

```
<cic:date>
...
</cic:date>
```

Example

```
<cic:date>
    <cic:property name="CreateDate" />
</cic:date>
```

cic:div Tag

Use this tag to generate an HTML DIV tag.

Attributes

- css="*class*": specifes the CSS class used to format the link.

- id="*id*": used to set an ID for this DIV element.

Usage

```
<cic:div css="box no-border">
<cic:span css="instruction"
value="${localized['InstructionalText']}"/>
</cic:div>
```

cic:el Tag

Use this tag to include HTML tags such as <br/> or . You can use this tag to add hidden field in forms:

```
<cic:el>
<input type="hidden" name="parameterName" value="parameterValue">
</cic:el>
```

You can also use the rendered attribute of the cic:el tag to mark a particular section of HTML as being displayed if a condition is met. For example:

```
<cic:el rendered="${line.isMajorLine && line.isEditable}">
Renderable content
</cic:el>
```

You can read this to say: only display this content if the line is a major line item and it is editable. If you want to use boolean values to control this tag, then the booleans must be added to the request object. For example:

```
<%
    boolean rule = true;
    request.setAttribute("rule", new Boolean(rule));
%>
<cic:el rendered="${rule}"><hr></cic:el>
```

cic:img Tag

Use this tag to display images when you want to test for a condition to determin eif the image should be displayed.

Attributes

• rendered="*true*": determines whether the image is displayed.

• src="*URL*": specifies the URL for the image source.

You can also specify the other standard image attributes used by the IMG HTML tag.

Usage

```
<cic:img src="../images/btn_add.gif"
    width="37" height="19" border="0" />
```

cic:input Tag

Use this tag to specify the input field use to enter values. the id attribute distinguishes between input fields. The value attribute is used to specify the name-value combination of the request parameter that is returned to the server.

The type attribute of a cic:input tag can be used to specify different forms of input field, such as "checkbox", "hidden", "input", "radio", and so on.

You can specify the following HTML attributes as being true or false:

- disabled

- readonly

- ismap

- multiple

- checked

- nowrap

- selected

Example

```
<cic:input id="QuickSearchCartSKU" selected="true"
    value="${cic:equals(param['CommerceQuery_SearchField'],
    'CommerceQuery_SKU')?
    param['CommerceQuery_SearchFieldValue']:''}"/>
```

cic:inputDate Tag

Use this tag to create an input field with a calendar widget associated with it. To use this tag, you must include the **Calendar.jsp** JSP page and reference the required Javascript files as follows:

```
<jsp:include page="../common/Calendar.jsp"/>
<script type="text/javascript" src="../js/commerce.js"></script>
<script type="text/javascript" src="../js/com_DateUtils.js"></script>
```

Attributes

- id="*field*": identifes the name of the parameter associated with the input field.

- value="*value*": the displayed value if it exists.

Usage

```
<cic:inputDate id="CommerceQuery_ContractStartDateStart"
    value="${cic:formatDate(query.dateFieldsMap['ContractStart-
Date'].startDate, null)}" />
```

cic:javascriptLink Tag

The use of this tag is deprecated. Use this tag to generate a clickable link that invokes a Javascript method when clicked.

Attributes

• methodName="*MethodName*": name of Javascript method (required).

Usage

```
<cic:javascriptLink methodName="MethodName" >
...
</cic:javascriptLink>
```

You must provide any parameters to the tag using the cic:param tag.

Example

```
<cic:javascriptLink methodName="processDetail" >
    <cic:param name="SKU">
        <cic:property name="SKU" />
    </cic:param>
</cic:javascriptLink>
```

This tag will be replaced by the string href="javascript:processDetail(*SKU*)", where *SKU* is determined by the use of the cic:property tag.

cic:link Tag

The use of this tag is deprecated. Use this tag to generate a clickable link that points to a standard hypertext link.

Attributes

• app="*ApplicationName*": name of application (required).

• cmd="*CommandName*": name of message type to be specifed as the cmd parameter in the URL.

Usage

```
<cic:link app="ApplicationName" cmd="MessageType">
...
</cic:link>
```

You must provide any parameters to the tag using the cic:param tag.

Example

```
<cic:command_link target="_top">
    <cic:link app="*" cmd="PartnerProfileDisplay">
```

```
        <cic:param name="PartnerKey">
            <cic:property name="ContactPartnerKey" />
        </cic:param>
    </cic:link>
    <cic:property name="PartnerName" />
</cic:command_link>
```

This tag will be replaced by the string:

```
<a href="http://server:port/Comergent/en/US/catalog/matrix?
    cmd="PartnerProfileDisplay&PartnerKey=21">Anderel</a>
```

cic:options Tag

Use this tag to specify the values that may be selected from a drop-down list. It can be used to retrieve all values of the specified lookup type.

Attributes

- labelRef="*reference*" is the value of the description field of the lookup code.

- selectedValue="*value*": the value displayed when the page is first rendered.

- valueRef="*reference*" is usually the valueof the lookup code.

Users see the labelRef values, and their choice of search value is passed back to the server as the valueRef value.

Example

```
<cic:options var="statusCode" valueRef="${statusCode.code}"
    labelRef="${statusCode.string}" datasourceRef="${states}"
    selectedValue="${cic:equals(param['CommerceQuery_SearchField'],
    'CommerceQuery_InvoiceStatusCode')?
    param['CommerceQuery_SearchFieldValue']:''}"/>
```

In this example, the lookup type is the InvoiceStatus, so that the valueRef and labelRef fields will take pairs of values such as 10 and "New" as determined by the CMGT_LOOKUPS table values.

cic:outputLink Tag

Use this tag to generate links. The href attribute can use the *link()* method to generate the hyperlink and the cic:span tag is used to generate the displayed text for the link.

Attributes

- css="*class*": specifes the class used to format the link.

- href="*URL*": the hyperlink behind the visible link.

- rendered="*boolean*": used to specify if the link should be displayed.

- target="*target*": used to specify the target window or frame to populate when the link is clicked.

Example

```
<cic:outputLink rendered="${isLineCount}"
    css="focus-button focus-button-small right"
    href="javascript:createMemo();">
    <cic:span value="${localized['Create']}"/>
</cic:outputLink>
```

cic:param Tag

The use of this tag is deprecated. Use this tag to generate parameters in the form *name*=*value*.

Attributes

- name="*ParameterName*": name of parameter (required).

Usage

```
<cic:parameter name="ParameterName">Value
</cic:parameter>
```

You must provide the value of the parameter as the value of the cic:parameter tag.

Example

```
<cic:param name="SKU">
    <cic:property name="SKU" />
</cic:param>
```

This tag will be replaced by the string SKU=*SKU*, where *SKU* is determined by the use of the cic:property tag.

cic:property Tag

The use of this tag is deprecated. Use this tag to retrieve values from a bean. The bean is usually specified in the component tag that includes the cic:property tag as a child element.

Attributes

•   name = "*PropertyName*": name of property (required). There must be a
    corresponding *get*PropertyName*()* method defined on the bean. If no such
    method exists, then the tag will attempt to retrieve a data bean from the
    bean by first calling *getDataBean()* on the bean, and then calling the
    *get*PropertyName*()* method if it is defined on the retrieved data bean.

Usage

```
<cic:property name="PropertyName" />
```

Example

Suppose that you have a Cart data object and a corresponding CartBean data bean,
and suppose that for presentation purposes, you have created a
CartPresentationBean as a wrapper to the CartBean class. Suppose that you use the
cic:property tag like this:

```
<cic:commerceProductList beanName="cartPresentation"
    formAction='<%=link("*","ShoppingCartDataDisplay")%>'>
...
    <cic:property name="SKU">
...
</cic:commerceProductList>
```

This tag will be replaced by the value returned by *getSKU()* invoked on the
cartPresentation bean. If the cartPresentation bean has no *getSKU()* method, then
the tag will effectively try to call:

```
CartBean cartBean = cartPresentationBean.getDataBean();
cartBean.getSKU();
```

This will display the result of calling *getSKU()* on the cartBean data bean.

cic:quickSearch Tag

Use this tag to display a Search widget on a page. Use the cic:quickSearchParam
tag to specify which fields should be displayed in the drop-down list of search
criteria. Use the cic:outputLink tag to display other links within the Search widget
(such as More and Advanced Search links).

FIGURE 12. Example Quick Search Widget

The left-hand field is where users enter their search values, and the right-hand field is where they select on which field they want to search.

Example

```
<cic:quickSearch css="box medium padded-bottom"
    title="${localized['QuickSearchTitle']}"
    idToSubmit="CommerceQuery_SearchField"
    idValueToSubmit="CommerceQuery_SearchFieldValue"
    action='<%=link("*", "InvoiceWorkspaceDisplay")%>'>
    <cic:quickSearchParam name="CommerceQuery_OrderNumber"
        label="${localized['OrderNumber']}">
        <cic:input id="QuickSearchOrderNumber"
        value="${cic:equals(param['CommerceQuery_SearchField'],
'CommerceQuery_OrderNumber')?param['CommerceQuery_SearchFieldValue']:
''}"/>
    </cic:quickSearchParam>
...
    Possibly other cic:quickSearchParam tags
...
    <cic:outputLink css="edit" href='<%=link("*",
        "InvoiceWorkspaceDisplay", "ShowAll=true")%>'>
        <cic:span value="${localized['ShowAll']}"/>
    </cic:outputLink>
    <cic:outputLink css="edit" href='<%=link( "*",
        "InvoicesAdvancedSearch")%>'>
        <cic:span value="${localized['AdvancedSearch']}"/>
    </cic:outputLink>
</cic:quickSearch>
```

cic:quickSearchParam Tag

Use this tag within a cic:quickSearch tag. Each quickSearchParam tag specifies a different selectable search field. The cic:input tag specifies the name of the parameter that will be passed back to the server and how the search value is retrieved.

Example

In this form of the cic:quickSearchParam tag, the name of the search field is specified through the use of the cic:equals function, and the value field is left blank for free-form entry of the search value.

```
<cic:quickSearchParam name="CommerceQuery_OrderNumber"
    label="${localized['OrderNumber']}">
    <cic:input id="QuickSearchOrderNumber"
    value="${cic:equals(param['CommerceQuery_SearchField'],
'CommerceQuery_OrderNumber')?param['CommerceQuery_SearchFieldValue']:
''}"/>
</cic:quickSearchParam>
```

In the following form of the cic:quickSearchParam tag, the search values field is pre-populated by the valid values of the field:

```
<cic:quickSearchParam name="CommerceQuery_InvoiceStatusCode"
label="${localized['InvoiceStatus']}">
    <cic:select id="QuickSearchStatusCode">
        <cic:options var="statusCode" valueRef="${statusCode.code}"
        labelRef="${statusCode.string}" datasourceRef="${states}"
        selectedValue="${cic:equals(
    param['CommerceQuery_SearchField'],
    'CommerceQuery_InvoiceStatusCode')?
    param['CommerceQuery_SearchFieldValue']:''}"/>
    </cic:select>
</cic:quickSearchParam>
```

The valid values are determined by the cic:options tag. This uses the valueRef and datasourceRef attributes to retrieve the valid values of this lookup type from the Knowledgebase.

cic:select Tag

Use this tag to generate a drop-down list of values. It uses a cic:options tag to generate the list of valid option elements with their labels and values. The id attribute is used to distinguish between cic:select tags.

Attributes

- displayAsText="*true*": determines whether the component is read-only.

- id="*id*": identifies the input field.

- onchange="*function*": associates a Javascript function if the selected value in the drop-down list changes.

- rendered="*true*": determines whether the drop-down list is displayed.

Example

```
<cic:select id="QuickSearchStatusCode">
   <cic:options var="statusCode" valueRef="${statusCode.code}"
   labelRef="${statusCode.string}" datasourceRef="${states}"
   selectedValue="${cic:equals(param['CommerceQuery_SearchField'],
   'CommerceQuery_InvoiceStatusCode')?
   param['CommerceQuery_SearchFieldValue']:''}"/>
</cic:select>
```

cic:span Tag

Use this tag to generate display text. You can manage the display of localised text by using the ${localized['String']} function: this will retrieve the locale-specific form of the String value from the resource bundle for the JSP page.

Example

```
<cic:span value="${localized['AdvancedSearch']}"/>
```

cic:table Tag

Use this tag to display tables that take their data from ListBean objects. Within this tag, each column is declared using the cic:column tag.

Attributes

- css="data-table": this attribute specifies the style used to render the table.

- datasourceRef="${majorLines}": specifies an object to be used to iterate through the rows of the table. The object must be either of type Iterator, Collection, or Array. You must add the object to the request object and then retrieve it from the request. For example:

```
<%
ListIterator iter = commentListBean.getCommentIterator();
    request.setAttribute("commentList", iter);
%>
<cic:table datasourceRef="${commentList}" var="comment"
    showSelect="false" sortAscending="true" labelrowcss="label"
    rowcss="normal,alternate" >
```

- jsp="../cic/cicCommerceProductList.jsp":specifies a JSP page to be used to render the table. By default, **cic/cicTable.jsp** is used.

- labelrowcss="label": specifies the style used for the table labels.

- listBeanParam="List Iterator": this attribute is a deprecated attribute, you should use the datasourceRef attribute. It specifies the iterator used to iterate through the list of beans. Typically, you get this object by calling

the *get*Object*Iterator()* method on the list bean whose items you want to display.

- messageType: specifies the message type that is used to identify which JSP page is used to render the tag.

- rowcss="normal,alternate": specifies the styles used to display rows.

- showHeaderRow="true": specifies whether the table has a header row. If this attribute is set to "true", then you should specify a cic:columnHeader in each cic:column tag.

- showPagination="true": specifes whether to show the Previous and Next links to navigate back and forth through a paginated data set.

- showSelect: not used in this release.

- sortAscending: takes "true" or "false" to specify whether the table should be sorted ascending or descending.

cic:title Tag

The use of this tag is deprecated. Use this tag to set the title of a column in a table.

Attributes

- None

Usage

```
<cic:title>Title</cic:title>
```

Example

```
<cic:title><cmgt:text id="*">List Name</cmgt:text></cic:title>
```

This column in which this cic:title tag is used will have the title "List Name" (or whatever this text tag maps to in the generated JSP properties file.

cic:whitespace Tag

The use of this tag is deprecated. Use this tag to add space around text in a table.

Attributes

- None

Usage

```
<cic:whitespace/>
```

cic:workspace Tag

The use of this tag is deprecated. Use this tag to display workspace tabs in the Comergent eBusiness System UI.

Attributes

- cartListParam="*id of List Bean to be used to display rows*"

- sortAscending="*Sort order*"

- sortFieldName="*Field on which to sort*"

- formName="*FormName*"

- formAction="*FormAction*"

Usage

```
<cic:workspace cartListParam="workspaceLists"
    sortAscending="sortOrder" sortFieldName="sortField"
    formName="FormName" formAction="FormAction" >
...
</cic:workspace>
```

Example

```
<cic:workspace cartListParam="workspaceLists"
    sortAscending="<%=sortOrder%>"
    sortFieldName="<%=sortField%>" formName="domMyLists"
    "formAction='<%=link("*","WorkspaceDisplay")%>'>
    <cic:workspace_command operation="optionCommand" action="email"
        description="<cmgt:text id='cmgt_invoicing/
WorkspaceInvoiceData_11' bundle='invoicing.WorkspaceInvoiceDataRe-
sources'>Email Selected Invoices</cmgt:text>"
        type="option"/>
    <cic:workspace_command operation="optionCommand"
        action="download" description="<cmgt:text id='cmgt_invoicing/
WorkspaceInvoiceData_12' bundle='invoicing.WorkspaceInvoiceDataRe-
sources'>Download Selected Invoices</cmgt:text>"
        type="option"/>
    <cic:column width="3%">
        <cic:checkbox property="InvoiceKey"/>
    </cic:column>
    <cic:column width="12%"  sortProperty="InvoiceNumber"
        align="left">
        <cic:title>
            <cmgt:text
            id='cmgt_invoicing/WorkspaceInvoiceData_13'
            bundle='invoicing.WorkspaceInvoiceDataResources'>
                Invoice Number
```

```
            </cmgt:text>
        </cic:title>
        <cic:command_link target="_top">
            <cic:link app="*" cmd="InvoiceDisplay">
                <cic:param name="invoiceKey">
                    <cic:property name="InvoiceKey" />
                </cic:param>
            </cic:link>
            <cic:property name="InvoiceNumber" />
        </cic:command_link>
    </cic:column>
    ... some columns omitted
</cic:workspace>
```

cic:workspace_command Tag

Use this tag to add commands to a workspace tab defined using the cic:workspace tag.

Attributes

- operation="*OptionCommand*"

- action="*Action*"

- description="*Description of command*"

- type="*Type of command*"/>

Usage

```
<cic:workspace_command operation="OptionCommand" action="Action"
    description="Description of command" type="Type of command"/>
```

Example

```
<cic:workspace_command operation="optionCommand" action="email"
    description="<cmgt:text id='cmgt_invoicing/
WorkspaceInvoiceData_11' bundle='invoicing.WorkspaceInvoiceDataRe-
sources'>Email Selected Invoices</cmgt:text>"
    type="option"/>
```

JSP Expression Language

Overview

To address the current restrictions on column cell customization, the Comergent eBusiness System now supports use of the expression language defined in the JSP

2.0 specification. This adoption allows for greater customizability for the column cells while making compatible with the future specifications.

The expression language introduces a new "el" tag and modifies the "if" and "icon" tags to use expression languages for defining conditions. For a more detailed explanation of the full capability of the expression language please refer to JSP 2.0 specifications and the J2EE 1.4 tutorial mentioned above.

Tag Changes

el tag

The "el" tag allows the JSP page designer to write expression language scripts that will be evaluated on each iteration to render the tables. Those scripts can access the request, session parameters, and JSP page variables. The examples below use the current lineItem bean for the row being rendered using the identifier "lineItem".

You can access three methods to protect from cross-site scripting attacks as follows. The methods are "cic:pu", "cic"ph" and "cic:pj": these all take a single string and return a string.

An example of the usage of the "el" tag follows:

```
<cic:el>
    <table cellpadding="0" cellspacing="3" border="0">
    <tr>
        <td>
            ${cic:pu(lineItem.formattedComputedPrice)}
        </td>
    </tr>
    <tr>
        <td>
            ${cic:pu(lineItem.formattedDiscountMarkupComputedPrice)}
        </td>
    </tr>
    </table>
</cic:el>
```

In this example, you can see that the "el" tag can incorporate normal text and an expression enclosed with the "${}" syntax. The "cic:pu" call gives access to the "pu" function that protects against cross-site scripting, while the "lineItem" reference gives access to the current row bean.

The following example shows how to access the properties in the data bean associated with the presentation bean; this is accomplished using the dot notation.

```
<cic:el>
<input type = "text"
```

```
    name = "Discount${cic:pu(lineItem.dataBean.lineKey)}"
    value = "${cic:pu(lineItem.discount)}" />
</cic:el>
```

Note that the property name must have its first character in lower case.

if and icon Tags

Note that as of Release 7.0.2 the use of the cic:if tag is deprecated. Use the rendered attribute to manage tests for conditions that control the display of elements. The rest of this section is included to support legacy code.

For the if and icon tags, Release 6.4 has added an alternate usage to make use of the expression language. Instead of using the "property" and "testValue" attribute to specify a condition, you can use the new "test" attribute to specify an expression that will evaluate to true or false. Using expression language allows the access to the data bean's properties and to the request and session attributes as well as page variables.

The following examples demonstrate how to use the new attribute instead of the old ones. Here is an example of the usage of the if tag from earlier releases:

```
<cic:if property "isEditable" testValue="true">
    <cic:then>
        ...
    </cic:then>
    <cic:else>
        ...
    </cic:else>
</cic:if>
```

This is an example of the new usage:

```
<cic:if test="${lineItem.isEditable}">
    <cic:then>
        ...
    </cic:then>
    <cic:else>
        ...
    </cic:else>
</cic:if>
```

This example does not show much difference in format, but it makes it easier to identify the property to the line item bean.

In earlier releases, you have to nest if tags to test for two conditions:

```
<cic:if property "isEditable" testValue="true">
    <cic:then>
        <cic:if property "IsInvalidProduct" testValue="true">
        <cic:then>
```

```
        ...
        </cic:then>
    </cic:then>
</cic:if>
```

You can now implement this as:

```
<cic:if test="${lineItem.isEditable && lineItem.isInvalidProduct}">
    <cic:then>
        ...
    </cic:then>
    <cic:else>
        ...
    </cic:else>
</cic:if>
```

In this example we see that we can use a single if tag to specify an "and" condition instead of having to nest if tags.

The next example shows how to access a data bean property. This was not possible in Release 6.3.1 and earlier releases.

```
<cic:if test="${lineItem.dataBean.lineKey='44'}">
    <cic:then>
        ...
    </cic:then>
    <cic:else>
        ...
    </cic:else>
</cic:if>
```

The next example shows how to access the request parameter called "productId". This was not possible in Release 6.3.1 and earlier releases.

```
<cic:if test="${param['productId']=='6'} ">
    <cic:then>
        ...
    </cic:then>
    <cic:else>
        ...
    </cic:else>
</cic:if>
```

CHAPTER 32 *Internationalization*

This chapter describes the internationalization (i18n) and localization (l10n) issues that you must bear in mind as you work on Comergent eBusiness System applications.

Overview

The Comergent eBusiness System is internationalized: that is, it has built-in support for:

- multiple currencies
- multiple languages
- number and date formats
- character sets

In addition, you can manage other aspects of localization for specific markets such as:

- local laws and regulations
- currency processing
- shipping and export information
- taxes

Support for internationalization is managed using locales. Each locale identifies a language and country. By identifying which locale is to be used when displaying information to a user, you ensure that the user sees information that is both specific to their locale and presented as they would expect to see it. See the *Comergent eBusiness System Implementation Guide* for more information regarding locales.

When users log in to the Comergent eBusiness System, a locale is assigned to the session: this is the preferred locale specified in the user's profile. Users can change their preferred locale in their user profile, but the change will only take effect when they next log in. User administrators can change a user's preferred locale just as they can change other aspects of a user's profile. See *Comergent eBusiness System Administration Guide* for more information about user administration.

In addition, the system default locale is specified in the **Internationalization.xml** configuration file using the defaultSystemLocale element. In addition, you can specify a default locale for each language: see "Failover Behavior" on page 400 for more information.

The Comergent eBusiness System offers full Unicode support for data entry and display.

A significant amount of localization can be performed using Java ResourceBundles: see "Resource Bundles and Formats" on page 408 for more details.

Supporting Locales

If you plan to implement the Comergent eBusiness System to provide support for more than the en_US locale, then you must produce pages to reflect local language and other locale-specific information (such as office locations).

Presentation and Session Locales

When a user logs in to the Comergent eBusiness System, the authentication process retrieves their preferred locale: this is defined in their user profile. The system makes use of two logically distinct locales:

- session locale: this determines what data is retrieved for data objects from the Knowledgebase.

- presentation locale: this determines what JSP pages and resource bundles are used to render HTML pages to the user.

In general, the set of locales that you support as presentation locales must be a subset of the possible session locales. For example, you choose to maintain fr_CA,

fr_CH, and fr_FR as session locales, but only support fr_FR and fr_CA as presentation locales.

When a user first logs in, the system calculates a presentation locale for the user session as follows:

1. If the user's preferred locale is declared in the Comergent eBusiness System **web.xml** file, then set this to be the presentation locale.

2. If not, then consult the **Internationalization.xml** file: if the useCountryDefaulting element is set to "true", then identify the default country locale for the language of the user's preferred locale. Check to see if the default country locale is declared in the **web.xml** file. If it is, then set the presentation locale to this.

3. If either the useCountryDefaulting element is set to "false" or the default country locale is not present in the **web.xml** file, and if the useGeneralDefaulting element is set to "true", then set the user's presentation locale to the default system locale specified by the defaultSystemLocale element.

4. If the Defaulting elements are set to false or if no locale is identified that is declared in the **web.xml** file, then the presentation locale is set to the session locale.

This presentation locale is used to determine the user's experience as they navigate through the Comergent eBusiness System by controlling which JSP pages and properties files are used to render the Web pages that they see. At the same time, the preferred locale is also set as their *session locale*: this session locale is used to determine what data is retrieved from the database when localized data objects are displayed to the user.

Attention:	You must make sure that every locale you create in the database either has a corresponding set of entries in the **web.xml** file or that its default country locale has entries in the **web.xml** file and you enable country defaulting. If you do not do this, then some users may not be able to access the system.

JSP Pages and Properties Files

1. For each JSP page, there must be at least one JSP page located in the appropriate module sub-directory under the system default locale directory. When you first install the Comergent eBusiness System, the default system locale is set to en_US. Consequently a full set of JSP pages is provided under *debs_home*/**Comergent/WEB-INF/web/en/US/**. If you change the default

system locale, then take care to fully populate the corresonding directories for the new locale.

2. All visible text on each page is declared using the Comergent tag library text tag or the corresponding *cmgtText()* method. For example:

```
<cmgt:text id='cmgt_channelMgmt/channelCartDisplay/
ChannelCartDisplayData_7' bundle='channelMgmt.channelCartDis-
play.ChannelCartDisplayDataResources'>Build Product List
</cmgt:text>
```

or

```
String title =
    cmgtText("cmgt_commerce/search/AdvancedSearchBody_2",
    "Inquiry Lists Search");
```

The bundle attribute must correspond to a file in the com.comergent.reference.jsp package of the class tree. For the example above, there must be a file called **ChannelCartDisplayDataResource.properties** in the ***debs_home*/ Comergent/WEB-INF/classes/com/comergent/reference/jsp/ channelMgmt/channelCartDisplay/** directory. The id attribute must be unique within the properties file. For the example above, there should be a line of the form:

```
cmgt_channelMgmt/channelCartDisplay/
ChannelCartDisplayData_7=Build Product List
```

3. For each additional supported locale (say, *la_CO*), you must copy the following directories from **debs_home/Comergent/WEB-INF/web/en/US/** to ***debs_home*/Comergent/WEB-INF/web/*la*/*CO*/**:

   • **cic/**

   • **common/**

   • **home/**

4. For each additional supported locale (say, *la_CO*) and for each JSP page, you must:

   a. Either create a new JSP page for the locale and put it in the corresponding directory location in the Web application: a directory under ***debs_home*/ Comergent/WEB-INF/web/*la*/*CO*/**. If the same page can be used for more than one locale in the same language (for example, fr_FR and fr_CA), then make sure that you put it in the default locale for the language. See "Failover Behavior" on page 400 for more information about default locales for languages.

b. Or prepare a properties file that contains the appropriate text for each id. These properties files are organized so that there is one for each JSP page and JSP fragment.

> **Note:** HTML and Javascript characters such as "<", ">", "'", and so on must not be included in the property values. These characters must be escaped using the HTML or Javascript mechanisms to escape characters. For example: use "<" for "<" in HTML and "\'" for "'" in Javascript.

The properties files must conform to the Java standard for properties files used by resource bundles. Specifically, they should follow this naming convention: ***<Name of JSP page>*Resources_*la_CO*.properties**. They must be text files in which each line should take this form:

```
cmgt_module/package/JSPname_n=Display text for this locale
```

For example:

```
cmgt_channelMgmt/channelCartDisplay/
ChannelCartDisplayData_7=Build Product List
```

The properties files are all located in the ***debs_home*/Comergent/ WEB-INF/classes/com/comergent/reference/jsp/** directory and are organized by module within this directory in the same way that the module JSP pages are organized within a module. Note that if you want to change the location of these resource bundles, then you must customize the text tag to retrieve the resource bundles from their new location.

If you add text to a JSP page, then take care to update the corresponding locale JSP pages or properties files, either with amended text for an existing tag id or by adding a new id.

Notes

Note the following:

- The length of the translated text can be significantly different: this can affect the layout of a Web page.

- Drop-down lists and Javascript functions can have text that if translated will affect the logic of the Comergent eBusiness System. See "Javascript" on page 405 and "JSP Pages" on page 405.

- Local regulations can effect the display of information (such as the display of prices in both Euros and a local currency).

- Take particular care if the logical flow of pages must change to reflect local practice (such as the display of an export notice or tax information).

Debugging

You can use the debugJSPResouceBundle element of the **Internationalization.xml** configuration file to help you identify missing strings. Set this element to "true" and if a string is missing from the referenced resource bundle, then an error message is displayed on the browser page. You should set this value to "false" in your production systems.

Failover Behavior

This section describes what happens when resources (JSP pages or properties) are not defined for the user's current presentation locale. Note that the failover behaviors are slightly different for JSP pages and resource bundles:

- JSP pages can fail over from a specific locale to the default country for the language locale and then to the system default locale. For example: fr_CA to fr_FR to en_US.

- Resource bundles fail over according to the Java specification: ***_fr_CA.properties** to ***_fr.properties** to ***.properties**.

Two properties in the **Internationalization.xml** configuration file are used to manage failover behavior for JSP pages:

- useCountryDefaulting: if this is set to true, then default to the country specifed in the appropriate language element if no resource is present for the presentation locale.

- useGeneralDefaulting: if this is set to true, then default to the system locale if no resource is available for the presentation locale.

Resource Bundles

You do not need to translate all text strings into each locale. If a text string is not present for a given id in a resource bundle properties file, then the standard Java failover process is followed. For example, if the **ChannelCartDisplayDataResource_fr_CA.properties** does not define the cmgt_channelMgmt/channelCartDisplay/ChannelCartDisplayData_7 string, then, if it exists the **ChannelCartDisplayDataResource_fr.properties** file is consulted. If this file does not exist or does not have an entry for this id, then the **ChannelCartDisplayDataResource.properties file** is consulted.

JSP Pages

Not all the JSP pages need be available for all supported locales. For example, you may choose to use en_US pages for all but a small number of pages viewed by en_CA users. This section describes what happens when a message type is processed:

The request is forwarded to the JSP page specified by the JSPMapping element of the message type in the appropriate **MessageTypes.xml**.

1. If the JSP page does exist for the current locale, then this page is used to generate the Web page.

2. If the JSP page does not exist for the current locale, then the failover mechanism identifies the default locale for the language of the current locale. This is declared as the defaultCountry element for the language in the **Internationalization.xml** configuration file.

3. If a JSP page exists in the language-default locale, then this page is used to generate the Web page. For example, the following element in **Internationalization.xml** specifies that US is the default country for the en language locales, and so if a JSP page is not present for the en_CA locale, then the corresponding en_US JSP page is used.

    ```
    <en  visible="false">
        <defaultCountry ...>US</defaultCountry>
    </en>
    ```

4. If there does not exist a JSP page for the default country, then the failover mechanism identifies the default system locale. This is declared as the value of the defaultSystemLocale element of the **Internationalization.xml** file. If a JSP page exists in the system default locale, then this page is used to generate the Web page.

5. Finally, if no JSP page exists in the default system locale, then an exception is thrown and an error page is displayed.

Methods to Retrieve Locales

Most of the time you should be able to make use of the Comergent eBusiness System's built-in support to display appropriate content to users for their locales. If you do need to manually access locales, then the ComergentI18N class can be used. It provides the following methods:

*   *getDefaultLocale()*: returns the system default locale.

*   *getComergentLocale(boolean b)*: if b is true, then returns the user's presentation locale; otherwise returns the user's session locale.

• *findPresentationLocale(Locale sessionLocale)*: used to calculate what presentation locale should be used for a given session locale.

Using Properties Files in Code

You can make use of properties files in your Java code too. For example, to retrieve the locale-specific String that corresponds to the String keyString defined in the **com.comergent.reference.jsp.AdvisorBodyResources.properties** file, use:

```
String temp_NamedPopertiesFile =
    "com.comergent.reference.jsp.AdvisorBodyResources.properties";
ResourceBundle temp_ResourceBundle =
    com.comergent.dcm.util.ComergentI18N.-
        getBundle(temp_NamedPopertiesFile);
String temp_LocalisedString =
    temp_ResourceBundle.getString("keyString");
```

This uses the current locale of the user as stored in the user's session. If you want to force the use of a different locale, then use:

```
Locale specific_Locale = new Locale("fr", "CA");
String temp_NamedPopertiesFile =
    "com.comergent.reference.jsp.AdvisorBodyResources.properties";
ResourceBundle temp_ResourceBundle =
    com.comergent.dcm.util.ComergentI18N.-
        getBundle(temp_NamedPopertiesFile, specific_Locale);
String temp_LocalisedString =
    temp_ResourceBundle.getString("keyString");
```

Data for Internationalization

If you expect enterprise users and end-users to be entering data in multi-byte characters, then you need to consider the length of data fields and their corresponding database table columns. In our experience, data entered into the Comergent eBusiness System that uses multi-byte characters can be up to three times as long in the database as the strings used for the en_US locale. Consequently, you should review the length of fields in which you expect data to be entered that will take multi-byte characters: notably name and description fields.

If you want to change the length of fields, then bear in mind that you have to both change them in the **DsDataElements.xml** configuration file and make the corresponding change to the SQL script that is used to generate the Knowledgebase schema.

For example, to make the Description field of the Product data object suitably long for multi-byte characters, you must do the following:

1. Identify the data field that is used to hold product descriptions. Because the Product data object is a localizable data object (Localized="y"), this is the Description field of the ProductLocale data object. Its corresponding database table and column is CMGT_PRODUCT_LOCALE.DESCRIPTION.

```
<DataField Name="Description" ExternalFieldName="DESCRIPTION"
    Mandatory="n" Writable="y"/>
```

2. Suppose that you want to allow for descriptions that are up to 240 characters long:

```
<DataElement Name="Description" DataType="STRING"
    Description="Description" MaxLength="240" />
```

3. Change the corresponding SQL statement that creates the CMGT_PRODUCT_LOCALE table so that the DESCRIPTION column is set to VARCHAR2(720):

```
DESCRIPTION VARCHAR2(720) DEFAULT 'Not available',
```

4. Run the appropriate SDK targets (merge and createDB) to make the changes to your implementation of the Comergent eBusiness System.

Note that in this example, the Description data field is widely used by many different data objects and so changing its definition in the **DsDataElements.xml** configuraton file can have unanticipated side-effects elsewhere. An alternative approach is to create a new data field called ProductDescription and to use this in the ProductLocale data object. Thus, you could put in the **ProductLocale.xml** file:

```
<DataField Name="ProductDescription"
    ExternalFieldName="DESCRIPTION" Mandatory="n" Writable="y"/>
```

Then put in the **DsDataElements.xml** configuration file:

```
<DataElement Name="ProductDescription" DataType="STRING"
    Description="This is the product description field"
    MaxLength="240" />
```

Note also that if you provide a Javascript methods to validate that users have entered valid data in fields, then when you check for length of fields, check for the length specified in the corresponding DataElement.

Email Templates

If your system supports languages other than English and your installation of the Comergent eBusiness System uses email templates to generate messages that are sent to users, then bear in mind that these need to translated.

Release 6.4 has introduced the ability to use JSP pages to generate email messages: see CHAPTER 10, "Sending Email from the Comergent eBusiness System" for more information. This provides support for internationalizing email messages by using the existing framework for internationaizing JSP pages.

For legacy applications, you can use the default templates provided by the Comergent eBusiness System: these are located in *debs_home*/**Comergent/WEB-INF/templates/**.

HTML Pages

Static HTML pages must be translated where appropriate. If you want to provide support for multiple languages simultaneously, then you should take care to produce pages for each language. Provided that you maintain the location of these pages consistently across your locale directory structure, then the relative references to these pages will always resolve correctly to the correct HTML page.

For example, the following JSP fragment will dynamically generated URLs to point to a locale-specific **Example.html** page:

```
<A HREF="<cmgt:link app="catalog">
/static/Example.html
</cmgt:link>">
resourceBundle.getString("ExamplePage")
</A>
```

In this example, a resource bundle is used to determine the displayed text for the link.

Images

In general, use images that do not have embedded text. Doing so, ensures that you can use the same images in more than one locale: thereby reducing the cost of localization and maintenance.

However, where necessary you should provide localized versions of images. Just as for static HTML pages, you can use relative URLs to ensure that locale-specific images are retrieved from the correct location relative to the JSP page.

In particular, remember that all of the buttons in externally facing pages are image buttons with text. Where necessary, you should create localized versions of each button. The image source URLs can then be generated as follows:

```
<IMG ALT="Locale-specific alternate text goes here"
    SRC="../images/button.gif"></A>
```

Javascript

Take care to localize displayed text used in your Javascript. For example, alert dialog boxes should reflect the user's locale in the displayed text.

- Some Javascript files are included in the Web pages along these lines:

```
<script language='JavaScript' src='../js/genericUtil.js'>
</script>
```

    You must maintain these Javascript files for each locale so that the browser can correctly include these in the generated Web pages.

- When Javascript is defined within a JSP page or an included JSP fragment, then display text must be wrapped in the text tag. For example:

```
alert("<cmgt:text id="*">Product ID is missing.</cmgt:text>");
```

    When these tags are processed as part of the SDK tool, then the id attribute is changed into a unique ID, and the ID and body of the tag are added to the resource bundle for the JSP page or fragment.

JSP Pages

In general, all localization for labels, explanatory text, populated lists, and locale-specific formatting for dates and currencies should be reflected in the JSP pages created for a locale.

A useful organizing principle is to create a HashMap of all localized strings on page, and then to refer to this throughout the rest of the page. For example:

```
HashMap localized = new HashMap();
localized.put("TaskListHeader",
    cmgtText("cmgt_taskMgr/TaskWorkspaceData_3","Task List:"));
localized.put("QuickSearchTitle",
    cmgtText("cmgt_taskMgr/TaskWorkspaceData_4","Search for Tasks"));
localized.put("TaskID",
    cmgtText("cmgt_taskMgr/TaskWorkspaceData_5","ID"));
localized.put("TaskName",
    cmgtText("cmgt_taskMgr/TaskWorkspaceData_6","Name"));
localized.put("Status",
    cmgtText("cmgt_taskMgr/TaskWorkspaceData_7","Status"));
localized.put("Priority",
    cmgtText("cmgt_taskMgr/TaskWorkspaceData_8","Priority"));
localized.put("CreateDate",
    cmgtText("cmgt_taskMgr/TaskWorkspaceData_9","Create Date"));
request.setAttribute("localized", localized);
```

You can reference these strings using the scripting capabilities along these lines:

```
<cic:span css="banner" value="${localized['TaskListHeader']}"/>
```

This technique has the advantages that JSP pages are more readable, that you can re-use localized strings easily, and it is closer to the JSF model.

See "Calendar Widget" on page 406 for information about localizing this UI component. For example, populate a drop-down list of days of the week for a French-language locale as follows:

```
<SELECT Name="DayOfWeek">
<OPTION VALUE=0>dimanche</OPTION>
<OPTION VALUE=1>lundi</OPTION>
<OPTION VALUE=2>mardi</OPTION>
<OPTION VALUE=3>mercredi</OPTION>
<OPTION VALUE=4>jeudi</OPTION>
<OPTION VALUE=5>juin</OPTION>
<OPTION VALUE=6>vendredi</OPTION>
<OPTION VALUE=7>samedi</OPTION>
</SELECT>
```

You can also use resource bundles to manage locale-specific display information. For example, this would be an alternate method for populating a drop-down list of days of the week in the Gregorian calendar:

```
<SELECT Name="DayOfWeek">
<OPTION VALUE=0><%= resourceBundle.getString("Sunday") %></OPTION>
<OPTION VALUE=1><%= resourceBundle.getString("Monday") %></OPTION>
<OPTION VALUE=2><%= resourceBundle.getString("Tuesday") %></OPTION>
<OPTION VALUE=3><%= resourceBundle.getString("Wednesday") %></OPTION>
<OPTION VALUE=4><%= resourceBundle.getString("Thursday") %></OPTION>
<OPTION VALUE=5><%= resourceBundle.getString("Friday") %></OPTION>
<OPTION VALUE=6><%= resourceBundle.getString("Saturday") %></OPTION>
</SELECT>
```

Calendar Widget

When you use the calendar widget (see "Using the Calendar Widget" on page 217) in a JSP page, then it must be localized. You do this by customizing the **I18N.js** Javascript file to be found in the locale directory *debs_home*/**Comergent/**/*la*/*CO*/**js/**. For example, to support the de_DE locale, create a file called *debs_home*/**Comergent/de/DE/js/I18N.js** that reads:

```
// DEFAULT LOCALE (English)
var MONTH_NAMES = new Array('Januar', 'Februar', 'Maerz', 'April',
'Mai', 'Juni', 'Juli', 'August', 'September', 'Oktober', 'November',
'Dezember', 'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug',
'Sep', 'Okt', 'Nov', 'Dez');
var DAYOFWEEK_HEADER_NAMES = new
```

```
Array("So","Mo","Di","Mi","Do","Fr","Sa");
var WEEK_START_DAY = 0;
// Create CalendarPopup object
var popupCal = new CalendarPopup();
```

Reports

Bear in mind that all the *C3* Analyzer reports have labels and text. Localized text for each report is maintained in the Knowledgebase CMGT_ANALYZER_TEXT table. You must maintain this data for each supported locale to ensure that users see the appropriate locale-specific text in each report.

If you add a new locale, then you must add text for each of the text codes and report codes for the new locale. See the *Comergent eBusiness System Reference Guide* for more information.

Style Sheets

The Comergent eBusiness System uses cascading style sheets to set the formatting of HTML elements. If you use fonts for a specific locale, then make sure that you create a style sheet that specifies these fonts. For each locale save this locale-specific style sheet in the same relative location.

In JSP pages, you can include a locale-specific cascading style sheet, say **customer.css**, with the following:

```
<LINK rel="stylesheet" href="../css/customer.css" type="text/css">
```

System Properties

In general, the configuration files only present data to administrators. To localize these files, you should not need to change the names or values of elements, but you should consider changing the Help text for elements. Note that there is only one set of configuration files for each Comergent eBusiness System, and so you should use the language of the default system locale for these files.

Resource Bundles and Formats

PropertyResourceBundles and Properties Files

The Comergent eBusiness System makes extensive use of properties files to manage locale-specific data. These have replaced the use of ResourceBundle Java classes. See "Supporting Locales" on page 396 for more details.

ResourceBundles

A useful mechanism to manage localization is the use of Java ResourceBundles.

> **Note:** The use of resource bundles classes in the Comergent eBusiness System is deprecated. You should use properties files as described in "Supporting Locales" on page 396.

These are classes that manage locale-specific information. ResourceBundle classes used in the Comergent eBusiness System all extend the ListResourceBundle. These define the mapping between name Strings and the value Strings returned when the *getString (String nameString)* method is invoked.

By following the naming convention for ResourceBundles, you can create locale-specific ResourceBundles for all of the locales you need to support. For example, you can create the following ResourceBundles to be used in a new application called Inventory:

- InventoryResourceBundle

- InventoryResourceBundle_fr

- InventoryResourceBundle_fr_FR

- InventoryResourceBundle_fr_CA

The following scriptlet can retrieve the appropriate resource bundle for use in a JSP page:

```
<%
    String baseName = "AdvisorResourceBundle";
    ResourceBundle resourceBundle =
        AdvisorResourceBundle.getBundle (baseName,
            session.getLocale());
%>
```

NumberFormats and DateFormats

You can use the NumberFormat class to help you display numbers in locale-specific ways. You create an instance of a NumberFormat by passing in the locale to the constructor.

For example, the following scriptlet displays the total number of shopping carts in a format appropriate to the locale:

```
<%
    NumberFormat numberFormat =
    NumberFormat.getInstance(session.getLocale());
    int number = request.getParameter("ShoppingCartsTotal");
%>
<P>The number of active shopping carts in use is:
<%= numberFormat.format(number) %>
</P>
```

Similarly, use the DateFormat class to help you display date in locale-specific ways. You create an instance of a DateFormat by passing in the locale to the constructor.

For example, the following scriptlet displays the current date in a format appropriate to the locale:

```
<%
    DateFormat dateFormat =
    DateFormat.getInstance(session.getLocale());
    Date todaysDate = new Date();
%>
<P>It is now:
<%= dateFormat.format(todaysDate) %>
</P>
```

CHAPTER 33 *Exceptions*

This chapter describes the framework for exception handling in the Comergent eBusiness System. You should follow this to ensure consistency across your implementation of the system, and to help other people working on the implementation.

Comergent Exception Hierarchy

Exception Root

ComergentException

All compile time exception classes declared in the production software should inherit ultimately from com.comergent.dcm.util.ComergentException class. This class extends java.lang.Exception to provide chaining and an independent user message.

ICCException

ICCException provides a convenience subclass of ComergentException. Rather than create a set of exception classes for a subsystem, you can use the ICCException class uniformly across a subsystem.

ComergentRuntimeException

All runtime exception classes should inherit from
com.comergent.dcm.util.ComergentRuntimeException, which extends
java.lang.RuntimeException to provide identical functionality.

Subsystem Grouping

A subsystem of the Comergent eBusiness System is defined to be either a distinct
and separable application, or an application level or a system level service. A
subsystem is a logical organization. It may span multiple packages in the Java
package hierarchy or comprise part of a package.

Each logical subsystem is expected to declare its own exception root class. This
root inherits from ComergentException and is the parent class of all compile time
exceptions within the subsystem. The subsystem is defined to be either a distinct
and separable application, or an application level or a system level service. A
subsystem is a logical organization. It may span multiple packages in the Java
package hierarchy or comprise part of a package, although you should organize
your package structure in conformance with the logical subsystem organization.

For example, suppose there is a subsystem named Foo. There should be a class
FooException:

```
public class FooException extends ComergentException
{
    public FooException(String msg)
    {
        super(msg);
    }

    public FooException(String msg, Exception ex)
    {
        super(msg, ex);
    }
}
```

Suppose Foo responds to a bad initialization state by throwing
BadInitializationException for all subsequent requests. This exception would
inherit from FooException:

```
public class BadInitializationException extends FooException
{
    ...
}
```

Subsystem by Subsystem Exception Policy

Each subsystem should implement a consistent policy for differentiating exceptions. Either it should subclass the subsystem exception class for each distinct exception type (this is the standard Java style policy) or the subsystem's root exception should inherit from ICCException, and should set the status parameter to differentiate exceptions (this is the ICCException policy).

For example, if subsystem Foo chooses a Java style exception policy, then FooException should extend ComergentException. If subsystem Bar chooses an ICCException policy, then FooException should extend ICCException (which in turn extends ComergentException).

```
public class BarException extends ICCException
{
    ...
}
```

Exception Chaining

Each subsystem is expected to throw only exceptions from its own subsystem to its caller. If an underlying service throws an exception that a given subsystem cannot handle, then it is expected to catch that exception and rethrow an exception that is meaningful in its own context. The new exception should use a chaining constructor to include the original exception, so that when the exception is finally handled and logged, the original exception is not lost.

For example, suppose subsystem Foo attempts to open a property file and could incur an IO exception. If it implements a Java style exception policy, then it may declare a new exception class, FooPropertyFileException, which extends FooException. The IO Exception catch statement would throw a new FooPropertyFileException with a constructor that passes a message and the original I/O exception.

```
try
{
    ...
    Properties props = new Properties();
    props.load(input);
    ...
}
catch (IOException ex)
{
    // chain the io exception
    throw new FooPropertyFileException("Loading file" + filename, ex);
}
```

Throwing, Catching, and Logging Exceptions

When to Throw Exceptions

Exceptions should be thrown when the contract between a method and its caller cannot be fulfilled. This is the usage identified in the Java Language Specification. Unfortunately, this provides only a little guidance since the contract can be defined so broadly that exceptions are unnecessary, or defined so narrowly that exceptions occur frequently. As a general rule of thumb, exception usage should balance the following two opposing goals:

Exceptions should not be the norm.

- They involve the creation of an additional object, so, if only from a performance standpoint, it is problematic if exceptions can occur frequently.

- Mixing data and control should be avoided. The alternative to throwing an exception is often returning a null value from a method. This means that the return value encapsulates two meanings (success or failure and whatever the data means when present). It is good programming practice to avoid this usage where possible.

  If null is a reasonable value for the stated purpose of a method, or if a method is expected to fail often in the normal course of operation, then it is reasonable to return null to indicate failure; otherwise it is better to throw an exception.

Throwing Runtime or Compile Time Exceptions

According to the Java Language Specification, runtime exceptions should be thrown when the caller has provided erroneous input (in essence, breached the method contract) and it would be burdensome to declare a compile time exception. For example, if a caller invokes a method passing a negative value for a parameter that is an array index, it is reasonable to throw a runtime exception. Otherwise throw compile time exceptions.

Catch Clauses and Throws Declarations

Catch clauses and throws declarations should avoid being overly general. If the called method throws, for example, FileNotFoundException, then the caller should catch FileNotFoundException, not Exception or Throwable. The reason for this is that if the underlying code changes to throw a new exception, or ceases throwing

this exception, then it is desirable that the change produces a compilation error to signal to the programmer to consider the new situation.

There are exceptions to this rule where practicality should prevail. If the variety of exceptions that can be thrown is large and our response is the same in all cases, then there is no reason to catch each individually.

Logging Exceptions

If a method catches an exception and handles it (that is, does not rethrow it) then it should log it. Presumably this method knows the significance of the exception, and knows whether to log it with an error severity or some other lower level severity. Empty catch statements should be regarded with great suspicion.

Never do this:

```
catch (SomeException ex)
{

}
```

Do this:

```
catch (SomeException ex)
{
    Global.logVerbose(ex);
}
```

Or this:

```
catch (SomeException ex)
{
    ex.printStackTrace(Global.debugStream);
}
```

When exceptions from underlying subsystems or third party packages are caught and chained to a new exception, there is no need to log the exception. Some process further up the hierarchy will eventually catch and handle it, and the process will know how to log it.

Displaying Exceptions

In general, users of the Comergent eBusiness System should not see exceptions: the appropriate subsystem must handle the exception gracefully by responding appropriately to the error condition.

The Comergent eBusiness System error pages place the exception stack trace between HTML comments. By viewing the source of the displayed Web page, you can read the stack trace.

If an exception stack trace is passed to the JSP page, then bear in mind that the buffer limits of the JSP page may prevent a full exception message from being passed to the Web page. If a long exception stack trace is passed to a JSP page, then you can display it by modifying the buffer of the JSP page. Use the buffer tag as follows:

```
<%@ page buffer=1024kb %>
```

Once the error condition has been diagnosed and fixed, then you should remove this tag because it impacts performance.

CHAPTER 34 *Customizing Existing Reports*

This chapter provides an overview of what you need if you plan to customize the Comergent eBusiness System reports provided in *C3* Analyzer.

- "Small Changes" on page 417 describes easy changes that can be made to the reports; that is, changes that can be made without opening the report up in Actuate e.Report Designer, plus some very simple changes you can make within e.Report Designer.

- "Structure of a Report" on page 421 discusses the structure of reports. You will need to study this section if you plan to make more sweeping changes than those described in Small Changes.

Note that the instructions provided here presuppose some knowledge of Actuate products and their functions. In addition, you should familiarize yourself with the location of the Actuate Report Server: you must identify the location of ***actuate_home*** and ***actuate_server_home*** directories as described in the *Comergent eBusiness System Implementation Guide*.

Small Changes

Changing Database and Server Information

To change the database connection information, open up ***actuate_server_home*/ Comergent/ComergentConfig.xml** in a text editor. Add or modify Connection

elements as appropriate. See the *Comergent eBusiness System Implementation Guide* for information on this file.

If you change an existing Connection element and that is the one specified in the **Comergent.xml** configuration file, as the MarketplaceAnalyzer.AnalyzerDBConnection element, then the change that you make takes effect immediately. If you change the value of this element in the **Comergent.xml**, then you must restart the servlet container for this change to take effect.

Rebranding: Changing Colors and Logos

Comergent eBusiness System Release 7.0.2 introduces a simple mechanism that allows some rebranding exercises to be performed without modifying or rebuilding report files. The reports that ship with the product have certain design features in common: text sizes, fonts and colors, thickness and color of lines in tables, and header and footer formats. A Knowledgebase database table, CMGT_ANALYZER_PROPS, stores the values of properties as records. See the *Comergent eBusiness System Reference Guide* for more information about this table.

You can change the values stored here and hence the look-and-feel of the reports either by modifying the data before it is loaded as part of the minimal data or by directly updating the database table. Note that if you change the database table, then the changes will not be captured for re-use as part of the SDK project.

You cannot change the font face and size by modifying them in this table. If the font face or size need to be changed, then you must open the report in Actuate e.Report Designer and make the changes to the report. The other features listed here can be changed.

Note that colors are expressed in two ways:

• In RGB values, separated by periods rather than by commas

• Using Actuate color constants

The following Actuate color constants (which correspond to common Windows colors) are available for use in this table:

TABLE 16. Constant Color Values

Constant	RGB Value
Black	00
Blue	FF0000

TABLE 16. Constant Color Values

Constant	RGB Value
Cyan	FFFF00
DarkGray	404040
Forest	7F00
Gray	808080
Green	FF00
LightGray	C0C0C0
Magenta	FF00FF
Maroon	7F
Navy	7F0000
Olive	7F7F
Purple	7F007F
Red	FF
Teal	7F7F00
White	FFFFFF
Yellow	FFFF

Key for File Values

There are constant strings used in the report files whose display values are maintained in the CMGT_ANALYZER_PROPS table. **CmgtPropConstants.bas** is used to map strings in the *C3* Analyzer report files to codes that can be used to look up the corresponding codes in the CMGT_ANALYZER_PROPS table. For example, in **CmgtPropConstants.bas**, the line:

```
Global Const C_SECTION_HDR_TEXT_COLOR    = 5 ' Black
```

indicates that where C_SECTION_HDR_TEXT_COLOR specifies the color of text in Actuate report pages, the color is retrieved from the VALUE column of the record of the CMGT_ANALYZER_PROPS table whose PROPERTY_CODE column is 5. The ' character is used denote comments: the comments indicate the default values of these properties.

Note that changes to these values are propagated to all reports in the system that include them; changes designed to differentiate individual reports from one another can only be made by changing the report file itself.

SQL Changes

It is strongly recommended that you make changes to database queries only if you are very comfortable with writing SQL.

The SQL for database queries is generated by Actuate Basic functions. These functions are stored in separate, text files, named as **<*reportname*>SQL.bas**. Simple changes can sometimes be made to the queries in these files, provided that the number and type of the columns in the SELECT clause remain the same.

To make such changes, first determine which database server is used for the Knowledgebase.

- If DB2 Universal Server, then copy the **<*reportname*>.rod** file from *actuate_home***\My Reports\Comergent\oracle** to *actuate_home***\My Reports\Comergent\db2** and then open it with e.Report Designer Pro.

- If Oracle, then open up the **<*reportname*>.rod** file in the *actuate_home***\My Reports\Comergent\oracle** directory using e.Report Designer Pro.

- If SQL Server, then copy the **<*reportname*>.rod** file from *actuate_home***\My Reports\Comergent\oracle** to *actuate_home***\My Reports\Comergent\mssql** and then open it with e.Report Designer Pro.

Next, note that there is a **<*reportname*>SQL.bas** file in the **db2/**, **mssql/**, and **oracle/** directories. Use a text editor to open up the one appropriate for the database you are using (make a copy of it first). It will contain functions resembling the following:

```
Function GetOrderStatusesQuery( locale As String ) As String
    Dim mySql    As String
    mySql =         "SELECT "
    mySql = mySql &     "lookup_code, description "
    mySql = mySql & "FROM "
    mySql = mySql &     "vw_cmgt_lookups "
    mySql = mySql & "WHERE "
    mySql = mySql &     "lookup_type = 'erp_order_status' "
    mySql = mySql &     "AND locale = '" & locale & "' "
    GetOrderStatusesQuery = mySql
End Function
```

Note that this is a very simple example; the majority of the queries in these files are very much more complex. Note also that the job of the function is to create a string containing the query text, which is then passed back to the caller of the function.

Make changes to the query as you wish, but note the following:

- do not add, remove or change function parameters

- ensure that string quotes are preserved

- strings are appended in Actuate Basic using the & character, for example:

```
mySql = mySql & "AND locale = '" & locale & "' "
```

concatenates these strings, one of which is the locale function parameter.

When the changes are made, save the file and then build and run the report in e.Report Designer. If the report runs successfully, and you have no further changes to make, then deploy the report executable (*<reportname>*.**rox**) file to the server and test it again in the server environment.

Making Major Changes to Reports

If you want to make more sweeping changes than those described above, then you will need to read the next section to understand the basics of how our reports are constructed. You should also consult your Actuate documentation: at least the *Getting Started* and *Building Your First Report* guides to be found in the Actuate Online Help.

Structure of a Report

Basic Report Files

A report has at least the following files (some have more):

- a *<reportname>*.**rod** file

- a *<reportname>*.**rox** file for each supported database

- a *<reportname>***SQL.bas** file for each supported database

Replace *<reportname>* in the foregoing filenames with the name of the report; for instance, the files for the Direct Product Report are: **DirectProduct.rod**, **DirectProduct.rox**, and **DirectProductSQL.bas**.) The ROD file is the Actuate report design file, used by the Actuate IDE. The ROX file is the executable that is deployed to the server, and the BAS file contains Actuate Basic code to generate the SQL queries for the report.

If you have installed the files in the *actuate_home*/**My Reports/Comergent/** directory, as described in the *Comergent eBusiness System Implementation Guide* for Comergent eBusiness System Release 7.0.2, then your directory structure will

be as described in "Directory Structure" on page 425. You may want to check that your directory structure is the same as the one shown there before you go further.

Note first that the Comergent directory contains the subdirectories named after databases: **db2/**, **oracle/**, and **mssql/**. For a single report, the files are distributed in these directories as follows:

TABLE 17. File Locations

db2/	oracle/	mssql/
	<reportname>.rod	
<reportname>.rox	*<reportname>*.rox	*<reportname>*.rox
*<reportname>*SQL.bas	*<reportname>*SQL.bas	*<reportname>*SQL.bas

Note that the **<reportname>.rod** file occurs only in the **oracle/** directory. This is because the ROD file is the same for all databases; it contains data about the appearance of the report, but (as used in the Comergent eBusiness System) contains no database-specific information.

The ROX and BAS files, on the other hand, are database-specific, and hence copies are found in each database directory. If you are going to develop or customize in a SQL Server environment, then you need to copy the report's ROD file into the **mssql/** directory before beginning.

Each directory for a database server also contains a file called **CmgtDBConnection.rol**. This file contains the appropriate database connection information for database type.

Other Report Files

As you develop, you will notice that other report files will appear. **<reportname>.rov** is an intermediate development file that you do not really need to worry about; **<reportname>.roi** is a report instance file, the result of running the report.

Files Shared by All Reports

Files in the **include/** and **lib/** directories are shared by most or all of the reports. The table below gives a short summary of what each file contains. Note that files with the extension **.bas** are text files containing Actuate Basic functions, while the extension .rol indicates a binary library file. You can browse the BAS files using any text editor, and make changes as described in "SQL Changes" on page 420. The ROL files can only be opened with Actuate's e.Report Designer Professional.

The contents of these directories are created by Comergent, and distributed as part of the product. Their purpose is to provide reusable components, and a consistent look and feel.

The controls in **lib/CmgtControls.rol** also provide mechanisms to support internationalization and rebranding of reports.

All report controls should be created by dragging one of these controls into the report, and renaming it. Failure to do this will limit internationalization and customization capabilities of the resulting report.

TABLE 18. Support Files

Directory	File	Description
include/	**CmgtCodeLookup.bas**	Used in retrieval of lookup values.
	CmgtConstExternal.bas	Contains constants used by reports; these reflect the values of lookup codes defined in the Knowledgebase.
	CmgtConstInternal.bas	Contains constants used by reports.
	CmgtDateTimeUtils.bas	Contains functions used to display times and dates.
	CmgtGlobals.bas	Contains common global variables and some functions used to set them.
	CmgtPalette.apr	Contains a saved palette of controls. However, the preferred way to create controls is to drag them from **CmgtControls.rol**.
	CmgtPropConstants.bas	Maps constants defined in the reports to property codes that are used to retrieve values stored in the CMGT_ANALYZER_PROPS table.
	CmgtPropUtils.bas	Contains utility functions to retrieve property values.
	CmgtTextConstants.bas	Contains mappings from text in reports to text codes used to retrieve values from the CMGT_ANALYZER_TEXT table.
	CmgtUtilities.bas	Contains various utility functions.
	CmgtZeroValues.bas	Contains helper class for filling in missing values in charts.

TABLE 18. Support Files (Continued)

Directory	File	Description
lib/	**CmgtComponents.rol**	Contains various report components that are at a higher level than simple text or numeric fields.
	CmgtControls.rol	Contains text and numeric controls. All report controls should be created by dragging one of these controls into the report, and renaming it.

Directory Structure

The report executables are deployed on the server using the Actuate Management Console or e.Report Designer Pro.

ComergentConfig.xml File

The **ComergentConfig.xml** is deployed manually. It is deployed in the following location:

*actuate_server_home***/Comergent/ComergentConfig.xml**

Source Directory Structure

All of the report source files are managed under the following directory: *actuate_home***/MyReports/Comergent/**.

• Under this directory there are **db2/**, **mssql/**, and **oracle/** sub-directories. In these are located the BAS, ROD, and ROX files for each report. For example, the *actuate_home***/MyReports/Comergent/oracle/** directory contains the BAS, ROD, and ROX files for Oracle database servers.

• The *actuate_home***/MyReports/Comergent/lib/** subdirectory contains the ROL library files.

• The *actuate_home***/MyReports/Comergent/include/** subdirectory contains the utility files.

Integrating Reports Into the Comergent eBusiness System

In Release 6.4 and later releases, integration has changed a little from earlier releases. Here are some notes on how to integrate a customer's custom reports in a typical way. (This is written in such a way that it is easy to split custom reporting work between an Actuate developer and a Comergent application developer.)

Running and Viewing Reports from the Comergent eBusiness System

SOAP messaging is used to run reports, and they are then viewed using the view frameset in ActivePortal. The steps are as follows:

1. The user selects a report to run (and optionally also specifies parameters). The user clicks a link or button and the browser sends the resulting request to the Comergent application.

2. The Comergent application receives the request, and adds some additional parameters (database information, locale etc. – more on this later). It then generates a SOAP request to run the report which is sent to the Actuate server.

3. When the first page of the report has been generated successfully, the Actuate server sends back a response.

4. If the response indicates the report has run successfully, and includes an identifier for the report document, the Comergent application uses the information to build a URL for viewing the document. It then sends a message back to the browser redirecting it to that URL.

5. The browser is redirected to the URL, and downloads the DHTML that comprises the report document and its immediate frameset (the Actuate View frameset, which includes report search and navigation capabilities).
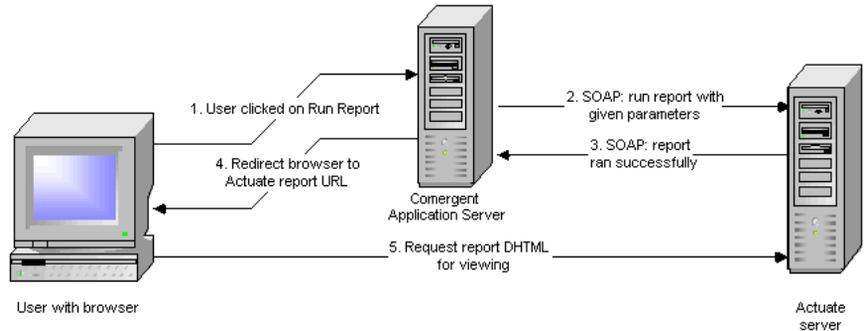
FIGURE 13. Exchanges performed when running and viewing a report

The illustration in Figure 14 on page 427 shows the Actuate View frameset embedded in a typical Comergent eBusiness System frameset. Note the Actuate navigation bar immediately above the report contents (includes "TOC", "Search", "Next", "Previous", and so on).
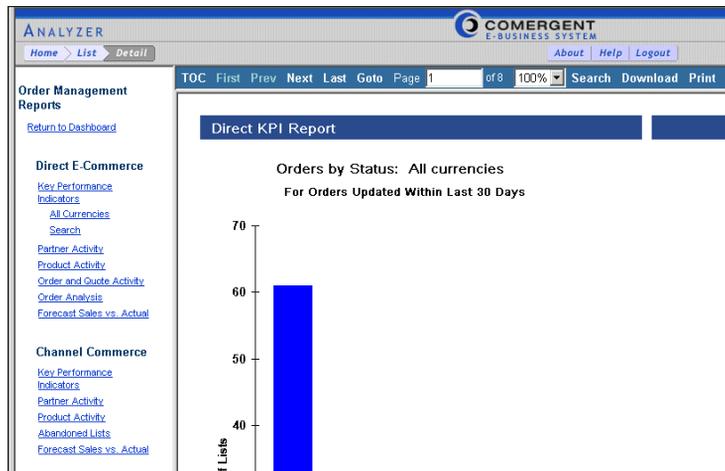
FIGURE 14. Typical Actuate View in Comergent eBusiness System Frameset

Much of the activity involved in the run-and-view process is done under the covers by Comergent eBusiness System controllers. New reports and their associated search pages can be added to the application without modification to this code.

Linking to Reports from the Comergent eBusiness System

Creating links to the reports from the Comergent eBusiness System need not be done by the report developer. There is a good deal of separation between the reports and the application, so the only thing that the Comergent eBusiness System developer needs to know about the reports is the name of the report executable and the names and expected values of the report parameters.

The report developer, in turn, needs only to follow certain conventions in parameter naming, and to include the parameters that all reports must support.

Parameter Naming Conventions

All report parameters follow the convention of prefixing parameter names with the string "P_". In addition, if a particular report parameter already exists in another report, then the same name is used in any new report. The major reason for this is that it allows the reuse of report search pages (pages for users to enter report parameters).

Here is a list of some often-used report parameter names and their uses:

- P_PartnerName: A string with wildcards used to search for a partner name in cmgt_partners.

- P_PartnerKey: An integer used to match PARTNER_KEY in several tables.

- P_StartDay, P_StartMonth, P_StartYear: Dates are passed in as three separate parameters rather than a string. Using these will save a lot of work on the Comergent side. Report developers should use *GetStartDate()* (an Actuate Basic function in **CmgtDateUtilities.bas**)—or a modified version of this function—to put them together into a date and default to a standard start date if no date is passed in.

- P_EndDay, P_EndMonth, P_EndYear: As for the start dates, *GetEndDate()* is the corresponding function.

- P_CommerceCode: Integer matching the COMMERCE_CATEGORY column in the CMGT_PARTNERS table.

- P_PartnerTerritories: String of partner territory codes with commas as delimiters (e.g. 10, 20, 30). Plugs easily into SQL. To obtain a list of a partner's territories, join the CMGT_PARTNER table to the CMGT_PARTNER_TERRITORIES table by the PARTNER_KEY column (there may be multiple entries), then join that table to the CMGT_TERRITORIES table by the TERRITORY_KEY column.

- P_PartnerLevels: String of partner_level codes (e.g. 10, 20, 30). Compare with the PARTNER_LEVEL_CODE column of the CMGT_PARTNERS table.

- P_PartnerTypes: String of partner type codes (e.g. 10, 20, 30). Compare with PARTNER_TYPE_CODE column of the CMGT_PARTNERS table.

- P_ReportingPricelistKey: If for any reason you find yourself needing to get prices from a pricelist (reporting on Open carts/lists is the main reason for this), then include this parameter and the application will automatically pass in the key of the pricelist you should use.

- P_ProductID: The product ID or SKU

The following parameters need to be supported by all reports:

- P_DBConnection: The value of ConfigKey in **ComergentConfig.xml** (a standard Actuate configuration file). The value that will be passed to the report is stored in Comergent.xml, and is passed automatically when a report is run as described here.

- P_Locale: Must be used when the Comergent application is to be used in the international arena (and report labels must therefore be translated). Once again, the value will be passed automatically by existing Comergent controllers.

Creating Links in the Application

The most common way of adding custom reports to the application is by creating a simple page with a list of the reports, and creating a link to this page from an existing page. If there are both administration and outward-facing reports (for example, reports are made available to Partner Administrators as well as System Administrators) more than one listing page may be necessary.

A typical Administration report listing page might look like this (but would normally be nested inside the customer's standard Administration frameset):
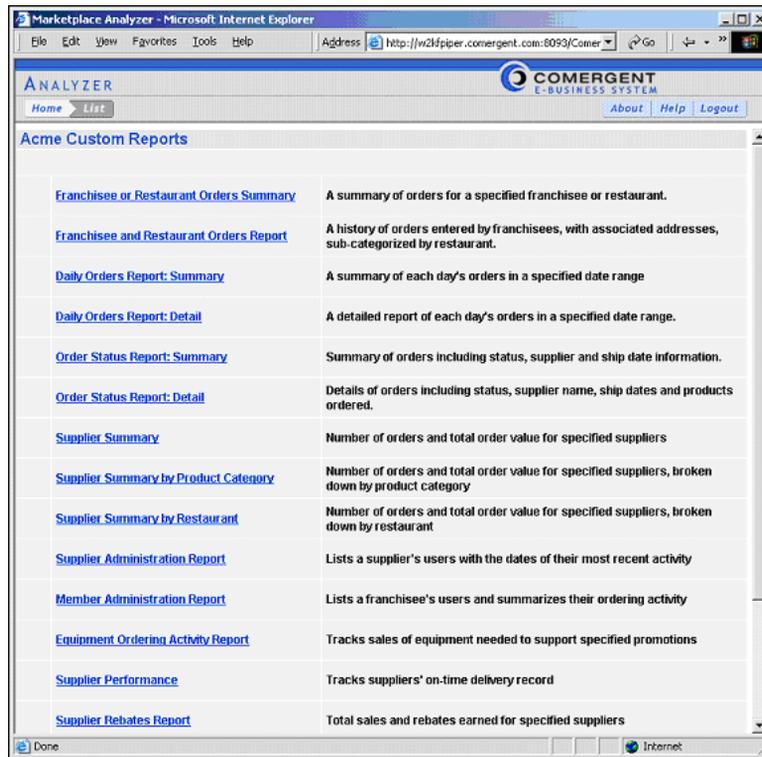
FIGURE 15. Typical Report Listing Page

Each link does one of two things:

• links to a search page for that report (where the user may enter parameters to filter results)

• links directly to a non-displaying JSP page that handles the running and displaying the report. (The search pages also link to this page after parameters are entered.)

The non-displaying JSP page can be used equally by out-of-the-box reports and custom reports. The page invokes the controller that handles the various exchanges mentioned at the beginning of this document. Under most circumstances, no changes are needed to the code for this page when adding a new report.

Search Pages

A typical report search page looks like the illustration in Figure 16 on page 431:
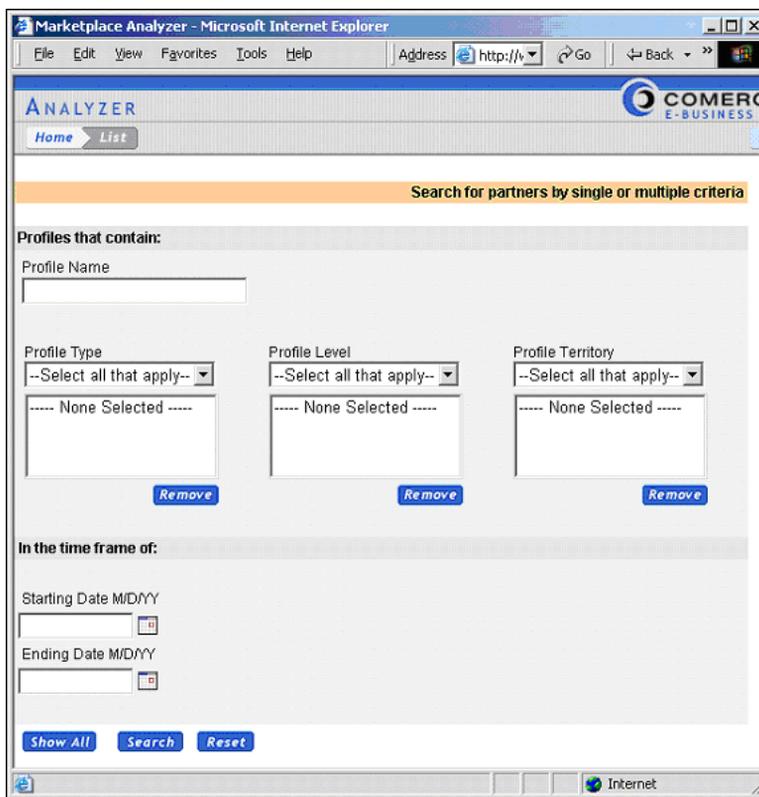
FIGURE 16. Typical Search Page (AnalyzerGenericSearch1.jsp)

This page combines a partner profile search with a date range search. A few important things to note:

• The page makes use of drop-down lists, but only data easily accessible from the CMGT_LOOKUPS table is included in them. There is no drop-down list for partner names (because there may be a large number of partners); instead, a wildcard search is offered.

• The date range search is used in almost all reports, except for those offering an analysis of current state.

As far as possible, it is best to keep the pages simple, and try not to create designs that require pulling lists of data from the database.

Development will be easier and faster if you can reuse or modify an existing page.

Using Drop-down Lists With Large Quantities of Data

As noted in the foregoing section, it is not usually recommended to use drop-down lists to select from large numbers of items. However, if a customer requires this, an example of how to manage this is included here. The example uses the selection of partner suppliers from drop-down where there are many choices (see the illustration in Figure 17 on page 432).
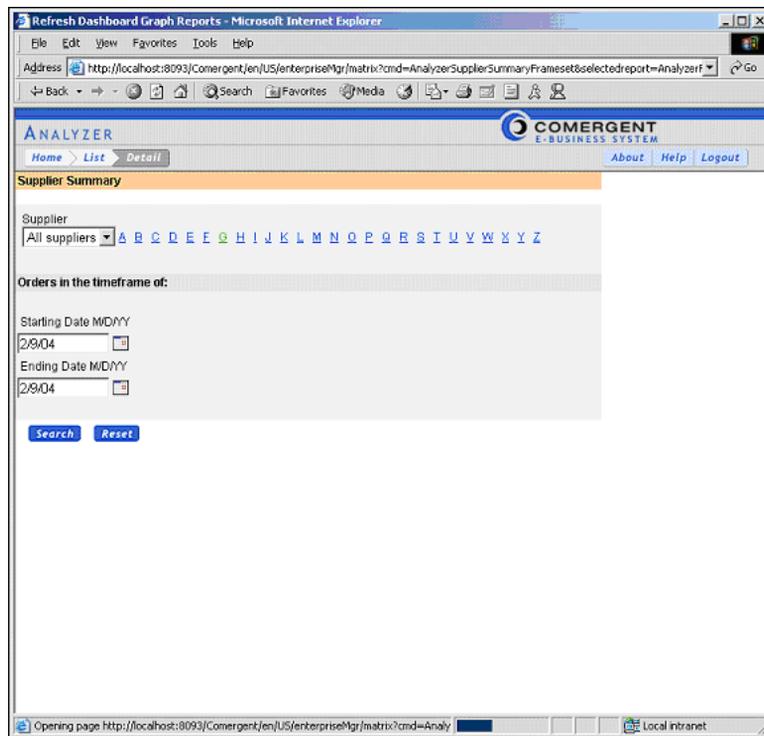
FIGURE 17. Selecting From a Large Number of Items

Note that the drop-down list always offers "All suppliers" as the first selection, and the other elements in the drop-down list are all the suppliers whose name begins with the selected letter. You select a letter by clicking on a link in the page. When

the page is first displayed, all the suppliers whose names begin with "A" are included in the drop-down list.

(The partner lists are retrieved by the controller using LightWeightPartnerList; the example controller source file (AnalyzerPartnerSearchController.java) is available from Comergent on request.)

Adding a Report to the Application: Checklist

Reports without search pages

If you do not want users to enter parameter values, then it is very simple to add an additional report to the application:

1. Create a link in your report list page with the name of the report. At this point you need to know the name of the report file and the parameters you will be passing, if any. For the purposes of this document, we'll assume that the name of the report executable is MyReport.rox.

2. If you have no parameters, and therefore no search page, the link should look like this:

```
<a href="<cmgt:link app="enterpriseMgr"
    cmd="AnalyzerDashReport">&selectedreport=TestKPI</cmgt:link>"
    target="data">
    My Report
</a>
```

(This link causes the report to be run and displayed in the target frame. The value of the selectedreport parameter must be the name of the report executable.)

Reports that reuse existing search pages

Reusing an existing search page is also fairly simple. Find the search page you like, and look in **AnalyzerMessageTypes.xml** for the corresponding message. For instance, if you plan to reuse **AnalyzerGenericSearch1.jsp**, searching in **AnalyzerMessages.xml** yields the following message:

```
<MessageType Name="AnalyzerAbandonedLists">
    <ControllerMapping>
        com.comergent.dcm.caf.controller.ForwardController
    </ControllerMapping>
    <JSPMapping>
        ../marketPlace/AnalyzerGenericSearch1.jsp
    </JSPMapping>
</MessageType>
```

For historical reasons, some pages (this one, for example) have several associated messages, but you can just pick one and ignore the rest. Plug the message name into your link as follows:

```
<a href="<cmgt:link app="enterpriseMgr"
    cmd="AnalyzerAbandonedLists ">
        &selectedreport=TestKPI
    </cmgt:link>" target="data">
    My Report
</a>
```

You will now be able to select parameters from the page, and they will be passed to the report. (This is one of the reasons why report parameter names are standardized.)

Creating a new search page

If none of the existing search pages fit your purposes, then you will need to create a new one. It is highly recommended that you find the page nearest your requirements and modify that rather than starting from scratch. In particular, if you need a date range, an existing page will contain a lot of HTML code and inclusions to support this element.

If you examine a typical existing page, then you will see that there is usually a *search()* function, which collects the various parameter values into a string suitable for a URL. The function *startDashboardReport(paramString, selectedReport)* is then called. This generates the message to the Comergent eBusiness System that starts the process of generating and viewing the report. If the correct values find their way into this function, everything will work correctly.

Once you have created or modified your page, you will need to create a new message type in **AnalyzerMessageTypes.xml**. Suppose your new page is called **MyNewReportSearch.jsp**, and you plan to name the new message "AnalyzerMyNewReport". You need to make two additions to **AnalyzerMessageTypes.xml**. The first goes in the section near the beginning that starts:

```
<MessageGroup Name="MarketplaceAnalyzerGroup">
```

All Analyzer messages are listed in this group. Add your message like this:

```
<MessageTypeRef Name="AnalyzerMyNewReport"/>
```

Now you need to add an entry to link this message to the new page. This goes further down the file, after the line that reads:

```
</MessageGroup>
```

Create the entry as follows:

```
<MessageType Name=" AnalyzerMyNewReport">
    <ControllerMapping>
        com.comergent.dcm.caf.controller.ForwardController
    </ControllerMapping>
    <JSPMapping>
        ../marketPlace/MyNewReportSearch.jsp
    </JSPMapping>
</MessageType>
```

If you have put the JSP page in a different location, then you will need to make sure the path is correct. The ForwardController class will always be used unless your page needs to bring back data from the database that is not already available in one of the pages. When you have modified **AnalyzerMessageTypes.xml**, you need to restart the Comergent server.

If everything is done correctly, and the Comergent eBusiness System is restarted, the new report and its search page are now linked in and will function correctly.

Making Reports Available to Users Other than System Administrators

The out-of-box reports are only available to the Comergent System Administrators, and any other Enterprise user with permission to view this part of the application.

The usual difficulty with making reports available to partners and other users is that reports need to be parameterized with their partner or user keys, to prevent them seeing other people's data. Clearly this parameterization must be hidden from the user; it needs to be done in a controller, and not in the page, where users could see the parameters and their values, and perhaps modify them. The standard name for the partner key parameter is P_PartnerKey.

In Release 6.4 and later, running reports in real time is done using the AnalyzerRunReportNowController class. To run a report for a single partner only, a modified controller is needed.

Customize the **AnalyzerRunReportNowController.java** file by using the customize target to copy the source file to your project. Make the following changes:

First, find this piece of code:

```
Properties reportParams = new Properties();
addCommonReportParams(reportParams);
addReportParams(reportParams);
IAnalyzerReportRunImmedTask immedTask =
    iAnalyzerFactory.getAnalyzerReportRunImmedTask(reportName,
```

```
                    jobName,
             this.actuateAuthId, this.currentLocale, this.actuateReportDir,
             this.actuateVolumeName, userName, this.actuateWaitTime,
             reportParams);
```

Add these lines of code so it looks like this:

```
Properties reportParams = new Properties();
addCommonReportParams(reportParams);
addReportParams(reportParams);

Long partnerKey =
    ComergentAppEnv.getCurrentSession().getUser().getPartnerKey();
reportParams.setProperty("P_PartnerKey", partnerKey.toString());
IAnalyzerReportRunImmedTask immedTask =
    iAnalyzerFactory.getAnalyzerReportRunImmedTask(reportName,
        jobName,
        this.actuateAuthId, this.currentLocale, this.actuateReportDir,
        this.actuateVolumeName, userName, this.actuateWaitTime,
        reportParams);
```

The report is now parameterized with the partner key. This can be used when exposing existing reports to partners, or when creating new partner reports.

Similar changes can be made to expose reports to individual users (not usually a good idea, unless the users are a very limited subset of the total), or to place any similar restrictions on the data that can be viewed by a specific user or group.

Implementing Cron Jobs

This chapter describes the creation of cron jobs that run as part of the Comergent eBusiness System.

Overview

Certain tasks within an implementation of the Comergent eBusiness System are not initiated in response to user input. For example, the hourly synchronization of order data with an external system or the weekly import of catalog data from a third party is best done without user intervention. These jobs can be scheduled to run at suitable intervals using the Job Scheduler functionality provided by the Comergent eBusiness System.

Cron jobs can be defined either as system cron jobs or as application cron jobs.

* A system cron job is run by the Comergent eBusiness System and is not associated with any user. A system cron job calls Comergent eBusiness System classes directly. A system cron job must be run by a class that extends the SystemCron abstract class. Typically, system cron jobs perform tasks such as cleaning the cache.

* Each application cron job is run as a user: the username and password of the user are provided when the cron job is created using the Job Scheduler user interface. Application cron jobs work by posting XML messages to the Comergent eBusiness System which are then processed by the system.

An application cron job must be run by a class that extends the
ApplicationCron abstract class. Typically, you use application cron jobs to
perform necessary administrative tasks that touch user or product data
such as order synchronization.

Attention:	Note that a system cron job should not attempt *restore()* and *persist()* operations itself. There is no user associated with the cron job class and so the access checking built in to the data access methods will throw an exception.

CronManager and CronScheduler

The definition and creation of cron jobs is managed by the CronManager class.
Cron job configuration information is represented in memory by the
CronConfigBean data bean. The definition of cron jobs are maintained in the
Knowledgebase.

The scheduling and running of cron jobs is managed by the CronScheduler class.
This singleton class is instantiated at server startup time.

CronJob Interface

Each cron job is a Java class that implements the CronJob interface:

```
public interface CronJob extends java.lang.Runnable
{
    /**
     * Specify the Cron Configuration bean object.
     *
     * @param config Cron configuration bean object.
     */
    public void setCronConfiguration(CronConfigBean config);

    /**
     * Return the Cron Configuration bean object.
     *
     * @return CronConfigBean object.
     */
    public CronConfigBean getCronConfiguration();

     /**
     * Initialization function. This function is called
     * immediately after the object is created.
     *
     * @return true if initialization success, false otherwise.
     */
    public boolean init();
```

```
/**
 * Return the current scheduled time.
 *
 * @return Current schedule time in Calendar object.
 */
public Calendar getSchedule();

/**
 * Reschedule the cron to reflect the changes made to the
 * cronfiguration parameter. This function is called by the
 * Cron Manager whenever cron configuration changes.
 */
public void reschedule();

/**
 * Whether the job needs to be run again. This function is
 * useful if there is some problem in the current run and you
 * want to retry at specified time.
 *
 * @return true if the job is allowed to retry if the job
 * did not run successfully
 * on the last time of execution
 */
public boolean retry();

/**
 * Determines whether to stop this cron job from running.
 *
 * @return true if the job has been slated to not run again
 */
public boolean stopRun();

/**
 * Compute next cron run time: this is usually based on the cron
 * run interval.
 */
public void computeNextSchedule();

/**
 * Check to determine if the cron job is
 * in a good state to run before triggering the thread to run.
 *
 * @return true or false. True means ready to run.
 */
public boolean isOKtoRun();

/**
 * Is called when the thread starts.
```

```
 *
 * @return false if the job needs to be stopped. Return true to
 * continue running.
 */
public boolean service();

/**
 * Checks whether the next run time is later than the end run date.
 *
 * @return true if next run time greater than end run time
 */
public boolean isExpired();

}
```

To create a new cron job, follow these steps:

1. Write a CronJob class: you must extend either the SystemCron or ApplicationCron classes. Both these classes are abstract and they both extend the abstract class AbstractCronJob.

   The only method that you need to implement is *service()*. This is the method that processes the inbound post initiated by the CronScheduler.

   • If the job is passed parameters that are defined using the Job Scheduler user interface, then you can retrieve the parameters using the *getParameter(String s)* and *getParameters()* methods of the AbstractCronJob class. These methods behave identically to the corresponding methods of the HttpServletRequest class.

   • If you want the result of the job to be saved to the database, then the *service()* method must call the *setExecutionOutcome(String s)* method.

   • You can specify that the cron job should be re-executed at a later time by calling the *setRetry(Calendar c)* method of the AbstractCronJob class. Use the Calendar parameter to specify when the job should be re-executed.

2. Using the Job Scheduler user interface provided as part of the system administration application, define the cron job by specifying the cron job class, the schedule to determine when it is run, and any parameters to be passed to the cron job at runtime. If the cron job is to run as an application cron job, then you must also provide the username and password of the user. See the *Administration Guide* for further information.

   Parameters are passed in to the cron job using the same syntax as for HTTP request parameters. For example: Name1=Value1&Name2=Value2.

CHAPTER 36　*Customizing Catalog Exports*

This chapter describes how you can customize the process of exporting products using the export functionality in *C3* Product Manager.

Overview

The Comergent eBusiness System provides the ability to export some or all of the product catalog as a dXML file. This is useful when you want to synchronize your catalog with another catalog installation, or to import your catalog in another catalog application.

When a user initiates a request to export products, the ProdMgrRunDataSyndController class invokes the ExportManager class to manage the export of the catalog data. The ExportManager class invokes the CatalogItemsExportHandler class to export the fields of the data object. This class uses the **DataSyndicationConfig.xml** configuration file to determine how to export each of the fields of data objects. By adding elements to this file, you can provide additional instructions on how fields are to be exported.

DataSyndicationConfig.xml Configuration File

The purpose of the **DataSyndicationConfig.xml** configuration file is to specify exactly how data objects should be exported and imported. Its basic structure is to

specify how entities should be exported and imported, and it does this by specifying what Java class should be invoked to process the entity. For example, this is an sample element:

```
<Entity
    Class="com.comergent.api.appservices.productService.IBizProduct">
    <EntityHandler Type="export"
Class="com.comergent.apps.productMgr.dataSynd.ProductExportHandler">
        <Element Name="ProductUpdate" Action="FullUpdateOrInsert"/>
    </EntityHandler>
</Entity>
```

It specifies that when instances of the class IBizProduct are exported, the ProductExportHandler class should be used to export instances of the IBizProduct.

- The Name attribute determines the element name of the output from the handler. In this case, the output will begin with "<ProductUpdate>" and end with "</ProductUpdate>".

- The Type attribute specifies whether the handler should be used for export, import, or both (Type="all").

The ProductExportHandler exports all of the fields in the IBizProduct data object, and so you need only add a custom field handler if you have added a new field to the IBizProduct that you want to export.

Field elements can be added to the Entity element to specify an additional handler for fields that are not exported by the main entity handler class. For example, suppose that Weight is declared as an extrinsic field of the BizProduct data object. Then the following element is used to specify that the ExtrinsicFieldHandler is used to handle the Weight field of the BizProduct data object.

```
<Field Mandatory="no" Localized="no"
    ElementName="Weight" BeanProperty="Weight">
    <FieldHandler Type="all"
        Class="com.comergent.apps.productMgr.dataSynd.-
            ExtrinsicFieldHandler"/>
</Field>
```

The ElementName attribute is used to specify the element name of the output. The BeanProperty attribute specifes the name of the data field which is to be exported. Thus for a particular IBizProduct object, the output for this field will look like this:

```
<Extrnisic Name="Weight">n</Extrinsic>
```

Here the value *n* is the value returned from a call to *getWeight()* on the IBizProduct object.

Handlers

ExtrinsicFieldHandler Class

The dXML DTD for catalog export supports the definition of Extrinsic elements. Typcially, these are used to manage data fields that are added to data objects as part of the customization process. Release 6.3 provides a generic field handler, the ExtrinsicFieldHandler class, that can be used to handle fields that are added to a data object by exporting them as Extrinsic elements. This class provides a lightweight means to export custom data fields without creating your own field handler.

To use the ExtrinsicFieldHandler class, simply add the appropriate Field element to the Entity element. For example:

```
<Field Mandatory="no" Localized="no" ElementName="Weight"
    BeanProperty="Weight">
    <FieldHandler Type="all"
        Class="com.comergent.apps.productMgr.dataSynd.-
            ExtrinsicFieldHandler"
</Field>
```

ElementName is the element name when the field is exported, and its value is determined by the call to the *getWeight()* method of the data bean. Make sure that the data object does have a data field called Weight (and hence a corresponding method *getWeight()*).

Writing a Custom Handler

Custom handlers for fields must implement the FieldExportHandler interface and its *getField()* method. For example, the following is an example implementation of *getField()* in class called PartNumberHandler:

```
public String getField(EntityField field, Object bean)
{
    IBizProductBean bizProductBean = (IbizProductBean) bean;
    String productID = bizProductBean.getProductID();
    PartNumberListBean pnlBean = new PartNumberListBean();
    pnlBean.restore(productID);
    StringBuffer temp_StringBuffer = new StringBuffer();
    temp_StringBuffer.append("<PartNumbers>");
    for {int i = 0; i < pnlBean.getCount(); i++)
    {
        PartNumberBean pnBean = pnlBean.getPartNumberBean(i);
        temp_StringBuffer.append("<PartNumber>" +
        pnBean.getPartNumber() + "</PartNumber>");
    }
```

```
    temp_StringBuffer.append("</PartNumbers>");
    return temp_StringBuffer.toString();
}
```

Having written and compiled the custom handler, you must add it to the **DataSyndicationConfig.xml** file:

```
<Field Manadatory="no" Localized="no" ElementName="PartNumbers"
    BeanProperty="none">
    <FieldHandler Type="export"
    Class="com.comergent.apps.productMgr.PartNumberHandler" />
</Field>
```

The ElementName attribute determines the name of the element used to enclose the output from the handler. When the product catalog is exported, then the XML file will include text along these lines:

```
<PartNumbers>
<PartNumber>PN-10056</PartNumber>
<PartNumber>PN-10058</PartNumber>
</PartNumbers>
```

CHAPTER 37   *Customizing **C3** Configurator*

This chapter describes how you can customize the *C3* Configurator. It covers:

Custom Controls

When option classes and option items are displayed to end-users you can control how they are displayed by specifying which control should be used to display their content. Out of the box, the Comergent eBusiness System supports the following choice of controls:

- Radio button
- Checkbox
- Drop down list
- Listbox
- Multiple Selection Listbox
- Display All Children

- User Entered Values

- Tabular Display

When modelers are creating the model for configurations, they determine which control is used for an option class by selecting it from the Control drop-down list on the Display tab of the option class detail page.

Each control corresponds to a JSP page and this correspondence is defined in the **control.properties** configuration file in *debs_home***/Comergent/WEB-INF/ properties/**. For example:

```
RADIO.name=Radio Button
RADIO.jsp=controls/radio.jsp
RADIO.behavior=single
```

This specifies that for the radio button control, the **radio.jsp** JSP page should be used to render the option class to end-users. The behavior property determines how the *C3* Configurator will handle picks in this control:

- entry: used for user-entered controls.

- expand: expand all the children of this control if the control itself is picked.

- multiple:allow one or more option items to be picked from this control.

- single: if an option item is picked, then remove any previous picks from this option class.

Customizing an Existing Control

You can customize an existing control by modifying the corresponding JSP page or by creating a new JSP page and modifying the **control.properties** file to point to the new JSP page.

Creating a New Control

You can define a new control by adding the name of the control to the list of controls declared. For example, to add a MATRIX_CUSTOM control:

```
controls=MATRIX_CUSTOM,RADIO,CHECKBOX,COMBOBOX,LISTBOX,
    MULTISELLISTBOX,ALLPICKED,UEV,DISPLAY
```

Then declare the properties of the new control as follows:

```
MATRIX_CUSTOM.name=Matrix Custom Control
MATRIX_CUSTOM.jsp=controls/MatrixCustom.jsp
MATRIX_CUSTOM.behavior=single
```

Customizing and modifying controls does not require a server restart because this file is read each time a Visual Modeler or *C3* Configurator session is launched.

Control Handlers

Control handlers are a mechanism for invoking Java code to handle special actions that may be difficult to handle in a JSP page alone. For example, the DynamicInstantiationControlHandler class dynamically adds child option items to a model when it is retrieved from the cache and removes them (the dynamic items) when the model is returned to the model cache.

The control handlers must implement the IControlHandler interface (typically they extend the StandardControlHandler class). They implement (or override the implementation of a base class) the methods:

- public void initializeControl(IModelBean model, IOptionClassBean optionClass)

- public void resetControl(IModelBean model, IOptionClassBean optionClass)

- public void handleComergentRequest(IModelBean model,ComergentRequest request,Map picks)

The *initializeControl()* method is called just after the model is fetched from the cache. The *resetControl()* method is called just before the model is returned to the cache. The *handleComergentRequest()* method is called to construct the picks map used to apply picks.

Function Handlers

This section describes how to implement function handler classes in the Comergent eBusiness System. These Java classes are used to define custom functions that can be invoked by the *C3* Configurator rule engine.

Overview

The Comergent eBusiness System provides a rule engine that is used to evaluate rules defined for each implementation. The rule engine can invoke custom functions to handle situations where existing functions are incapable of solving a configuration specification.

The function handlers are declared in the **functionHandlers.properties** configuration file. This file declares a name for each function handler and the

directory in which the function handler class is. For example, here is a sample fragment from the file:

```
WEB-INF/classes/com/comergent/apps/configurator/functionHan-
dlers=CheckLookupFunctionHandler,ChildSum,CountFunctionHandler,IsSele
ctedHandler,LengthFunctionHandler,ListFunctionHandler,LookupFunctionH
andler,MaxFunctionHandler,MinFunctionHandler,ParentFunctionHandler,Pr
opValHandler,SumFunctionHandler,ValueFunctionHandler,WebServiceLookup
CheckLookupFunctionHandler=com.comergent.apps.configurator.function-
Handlers.CheckLookupFunctionHandler
```

Writing a Custom Function Handler

Follow these steps to create a function handler class:

1. Create a new Java class with the com.comergent.apps.configurator.functionHandlers package declaration. The class declaration must declare that the class extends the AbstractRuleFunctionHandler class.

2. Implement the following methods:

    • public String getFuncName(): return the function name, such as "sum" or "max". This is case-sensitive: you can use different function handlers to manage "sum" and "SUM".

    • public int getType(): return the type of value returned by the function. This should be a constant defined in the com.comergent.api.appsservices.rulesEngine.Value class. The AbstractRuleFunctionHandler class method returns Value.STRING, and so you must override this method if the function returns any other type.

    • public Value handle(State state, String prop): return the Value calculated for the function.

    • public boolean isPublicHandler(): return true if the function handler may be used by any client application; otherwise return false. The AbstractRuleFunctionHandler class method returns true, and so you must only overwrite this method if the function handler is private.

Function Handler Example

The following example of function handler class implements the "max" function:

```
package com.comergent.apps.configurator.functionHandlers;

import com.comergent.api.appservices.rulesEngine.*;
import com.comergent.apps.configurator.model.*;
import java.util.*;
```

```
/**
 * Handles the logics of <i>Max</i> function for a <code>Property
 * </ code>, given the <code>State</code>.
 *
 *  @author Comergent Technologies
 *  @version    1.0
 *
 *  @see    Value
 *  @see    Property
 *  @see    State
 */

public class MaxFunctionHandler extends AbstractRuleFunctionHandler
{
    /**
     *  Name of the function, this particular handler serves.
     */
    private static final String m_name = "max"/*I18NOK:23c81106*/;

    /**
     *  Return the name of the function this handler supports
     *  @return the function name
     */
    public String getFuncName()
    {
        return m_name;
    }

    /**
     * Return the value type this particular function handler
     * returns.
     * Returns <code>Value.NUMERIC</code>, as the type.
     * @return      the numeric value.
     * @see    Value the container for different types.
     */
    public int getType()
    {
        return IValue.NUMERIC;
    }

    /**
     * Return the <i>Maximum</i> value asigned to the property,
     * given <code>State</code>.
     * <code>Value</code> is returned as a result.
     * Extracts all the matching properties given the name, and
     * sorts them and extracts
     * the maximum value.
     *
     * Returns <i>null</i>, if the requested <code>Property</code>
```

```
 * does not exist.
 *
 * @param  state   the property pool
 * @param  prop    the property to evaluate the function.
 * @return    Value   the <code>Property</code>, that contains
 * the maximum value.
 */
public IValue handle(IState state, String prop)
{
    //double max = 0;
    double [] propList = state.getMatchingNumericProperties(prop);
    if (propList != null)
    {
        Arrays.sort(propList);
        return new ConfigValue(new Double(
            propList[propList.length -1]),IValue.NUMERIC);
    }
    return null;
}
}}
```

In this example, the *handle()* method calculates the maximum value of a property by sorting the list of property values and then returns the last value in the sorted array. The function returns a number of type IValue.NUMERIC. It is a public function handler.

Filters

This chapter describes how you can use filters. It covers:

Filters Overview

A filter is an object that performs filtering tasks on either the request to a resource (a servlet or static content), or on the response from a resource, or both. They are defined as part of the J2EE 2.3 specification.

Filters perform filtering in the *doFilter()* method. Every Filter has access to a FilterConfig object from which it can obtain its initialization parameters, a reference to the ServletContext which it can use, for example, to load resources needed for filtering tasks.

Filters are configured in the deployment descriptor of a Web application. Examples of typical filters include:

- Authentication Filters
- Logging and Auditing Filters
- Image conversion Filters

- Data compression Filters

- Encryption Filters

- Tokenizing Filters

- Filters that trigger resource access events

- XSLT filters

- Mime-type Chain Filters

Available Filters

This section describes some of the filters provided in the Comergent eBusiness System. All the filters are part of the com.comergent.dcm.core.filters package. It covers:

- "DosFilter" on page 452

- "WSDLFilter" on page 453

DosFilter

This filter can be used as the basis for filters to protect the Web application from denial-of-service attacks.

To use this filter, write a class that extends the com.comergent.dcm.core.filters.DosFilter class, and in it, override the *isRequestDenied()* method to implement the logic you want to use to identify and block denial-of-service attacks.

Then, modify the **web.xml** configuration file, to declare your implementing class as a filter like this:

```
<filter>
    <filter-name>DosFilter</filter-name>
    <filter-class>
        com.comergent.dcm.messaging.CustomDosFilter
    </filter-class>
</filter>
```

and

```
<filter-mapping>
    <filter-name>DosFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

WSDLFilter

The WSDLFilter class is used to transform the Web service WSDLs if they are accessed using the standard URLs: http://server:port/Comergent/dXML/5.0/ OrderInterface.wsdl, and so on.

CHAPTER 39 *Managing and Displaying Constrained Fields*

This chapter covers the topic of managing constrained data fields which can take only one of a number of values: we called these data fields *constrained*. Examples include partner levels (such as "Gold", "Silver", and so on), partner territories (such as "North-west", "Benelux", and so on), and skill levels (such as "Expert", "Certified", and so on). You can manage these data fields in different ways in the Comergent eBusiness System. Your choice depends on how they are to be maintained and used.

Options

You have the following options to specify a constrained data field and the permitted data fields:

- Maintain the data field as a set of values in a database table. Assign values to business objects either by a cross-reference table or by references to a key for each value in the business object table.

- Maintain the values as a constraint element in the XML schema (declared in the **DsConstraints.xml** file). Specify the constraint as an attribute of the DataElement associated with the data field.

- Embed the permitted values as values of a <SELECT> form element in an HTML template.

We recommend that you maintain the permitted values for a field as a database table unless:

- the values are not going to be modified at run-time

- the data field may take only one value in each business object

- the values can be displayed in a natural order that is determined by the values themselves such as their alphabetical order.

We recommend against using the third option for the following reasons:

- It becomes a maintenance problem to update templates or application code if you want to modify the list of permitted data values.

- It represents a security problem because users may modify the HTML to pass back forbidden values. You have to either add Javascript (that a user can remove) to validate the selection or validate the returned value as part of the business logic.

Criteria

Your selection depends on the functionality of the data field. Ask yourself these questions to determine how the data field is being used:

1. Can you assign a business object only one or multiple values of a constrained data field?

   If your answer is that multiple values may be assigned to the same business object (example: a partner that may operate in multiple territories), then you *must* use a database table for the field values and a cross-reference table to assign values to the business object.

2. Can you enter new values of the data field when creating a new business object or do you need to verify that a value entered for the data field is a valid member of the constraint set?

   If only single values are permitted, and your answer to Question 2 is that new values are permitted, then you *must* use a database table to hold the field values. However, you do not have to use a cross-reference table to assign data field values to business objects. You cannot dynamically add values to the list of permitted values of a constraint element through the current Comergent eBusiness System interface.

3. Are the possible values that the constrained data field may take maintained dynamically or are they read once at start-up?

   If your answer to Question 1 was single value, and your answer to Question 2 is that new values are not permitted, but you do require dynamic updating, then you *must* use a database table. If the constrained values are unchanged once the Comergent eBusiness System has started, then you can use a constraint element.

4. Do you need to sort the constrained data values for display? If yes, then is it sorted by value (say, alphabetically) or by some defined order that cannot be inferred from the values themselves?

   Finally, if the data field values need to be sorted by an order not inherent in the values themselves, then this ordering information must be maintained in a database table. However, if you only order the values using some self-evident ordering (such as alphabetical), then you can use the constraint element choice.

CHAPTER 40 *Deprecated Concepts*

This chapter covers concepts that have been covered in earlier versions of this document, but which have now become deprecated. The material here is intended to be used to support legacy applications and should not be used to create new functionality.

DsElement Tree

This section describes methods to retrieve metadata about databeans. It also describes the DsElement tree used to store data in the data object and business object classes. It is covered here only to support legacy applications: all new applications that use the data bean classes should not need to be concerned with it.

Data objects are created as objects of data bean classes. Each data object holds its content as a tree of components called DsElements (see "DsElements" on

page 460). Their content is retrieved from external systems using the XML schema, and the recipes and data sources defined in the XML schema.
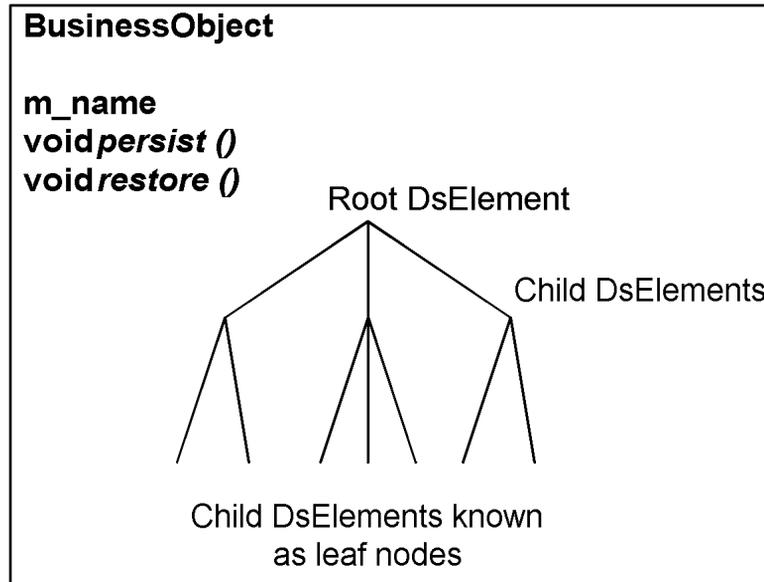
FIGURE 18. Business Object

When the DataManager creates a data bean or business object, it uses the XML schema to determine the structure of its DsElement tree. The DsElement tree is the Java representation of the structure of the business object. The schema also determines the data types that may be inserted at leaf nodes and whether constraints are placed on the values of the node. You access the DsElement tree by invoking the business object method *getRootElement()*.

DsElements

Each DsElement contains data and a DataMap that defines how its data corresponds to its data source. A DsElement may be the child of another DsElement (its *parent*). A DsElement tree is a collection of DsElements, all but one of which have another element in the tree as its parent. By definition, the DsElement with a null parent is the *root* DsElement.

```
DsElement

m_children
m_parent
m_dataMap
m_value

DsElement cloneDsElement ()
DsElement addChild (DataMap dataMap)
void delete ()
String getName ()
int getType ()
DsElement getParent ()
DsElement getByName (String s)
void deleteChild (DsElement child)
```

FIGURE 19. DsElement Methods

The DsElement class provides various additional methods to support navigating through a DsElement tree, notably *children()* that returns an Iterator of the child DsElements of a given DsElement. As well as *getRootElement()*, the business object class also provides the *getElementByName()* method to access directly a named DsElement in its tree.

All DsElements that have the same name, for example child_name, and which are children of a DsElement must have a parent whose name is <child_name>List. The XML schema identifies such elements by defining their ordinality to be "n" as opposed to "1". A DsElement maintains its children in a Vector called m_children.

The DsElement has these important methods:

• *addChild()*: adds a new DsElement defined by the DataMap of this DsElement.

• *cloneDsElement()*: returns a copy of this DsElement.

• *delete()*: sets the DsElemState to DsElemState.DELETED.

- *deleteChild()*: removes a child from the vector m_children by specifying it as a DsElement.

- *getName()*: returns the name of the element as defined by its MetaData.

- *getParent()*: returns the parent of this DsElement.

- *getType()*: returns the type of the element as defined by its DataMap.

DsElement MetaData

It is sometimes useful to retrieve information about a data field and its underlying DsElement. You can use the IData interface method *getMetaData(String elementName)* to this. It returns an object that implements the IMetaData interface. This interace supports the following methods:

- public int getDataType(): returns values as defined in DsDataTypes

- public long getMaxLength(): returns maximum length in bytes

- public long getMaxCharLength(Locale locale): returns maximum length in characters

- public Object getMinValue(): returns the minimum allowed value (or null if there is no minimum)

- public Object getMaxValue(): returns the maximum allowed value (or null if there is no maximum)

- public int getCountAllowedValues()

- public ListIterator getAllowedValueIterator()

- public Object getDefaultValue()

Note that each generated DataBean class implements the IData interface, and so these methods are available to all the generated data beans.

BusinessObject Methods

Use of business objects is deprecated. This section provides information about some business object methods for reference only.

You can create each business object as an instance of the Java class BusinessObject. The BusinessObject class is a sub-class of the CacheableAdapter class. This super

class provides a means of caching information during the lifetime of the business object. Each business object has a *type*: it defines the structure of the data it holds.

Note:	In general, where possible, you should avoid the use of business object classes: they are primarily a legacy of Release 4.0 and earlier releases.

In Release 6.3.1 and earlier releases, you could retrieve a business object representation of a business entity from the corresponding data bean by calling the data bean's *getBizobj()* method. This method is not supported in later releases.

In Release 6.3.1 and earlier releases, you could create a data bean from a business object by using the DataBean constructor method that takes the business object as its one argument. This constructor throws an InvalidBizobjException if the business object type does not match the data bean. This method is not supported in later releases.

restore() Method

This section provides description of the main forms of the BusinessObject *restore()* method.

```
public void restore(BusinessObject queryObj, int maxResults,
        boolean accessCheck)
```

The principal form of the *restore()* method. Use the queryObj parameter to specify query to be executed by the restore operation. The maxResults parameter determines the maximum number of objects returned. Use the accessCheck parameter to specify whether to check that the current user has the correct entitlements to perform this operation. Once the access check has been performed, then the *restore(BusinessObject queryObj, int maxResults)* is called.

```
public void restore(BusinessObject queryObj, int maxResults)
```

This method calls the *restore()* method *restore(this, queryObj, maxResults, false)*of the underlying data object.

```
public void restore(BusinessObject queryObj)
```

This is equivalent to calling *restore(queryObj, 0)*.

```
public void restore()
```

This form of the method calls the *restore(null, 0)* method.

persist() Method

This section provides descriptions of the main forms of the BusinessObject *persist()* method.

```
public void persist(boolean synch, boolean commit,
```

```
boolean accessCheck)
```

The boolean parameters determine respectively whether the persist operation is synchronized, should be committed to the underlying data source, and whether an access check should be performed prior to persisting.

```
public void persist(boolean synch, boolean commit)
```

This form of the method is equivalent to *persist(synch, commit, false)* for business objects whose Version attribute is 4.0 or less. It is equivalent to *persist(synch, commit, true)* for business objects whose Version attribute is 5.0 or more.

```
public void persist()
```

This form of the method calls *persist(false, true)*.

The BusinessObject class also has these methods:

*   *delete()*: empties the business object by deleting its DsElement tree.

*   *getRootElement()*: returns the root DsElement of the DsElement tree.

*   *getType()*: returns the name of the root element of the DsElement tree. This is the type of the business object.

*   *setRootElement()*: sets the root element of this business object.

Business Logic Classes

The base business logic class is BLC class. It implements the ApplicationObject interface and in particular defines *prolog()* and *service()* methods.

Note:	The use of BLCs is deprecated. This section is provided to support legacy applications that still use BLCs.

```
┌─────────────────────────────────────────────┐
│                                               │
│  Business logic class                         │
│                                               │
│  void prolog(BizO bjTable list, OutputStream os)│
│  BizObjTable service ()                       │
│                                               │
│  BizObjTable                                  │
│     ┌─────────────────────────────────────┐  │
│     │                                       │  │
│     │   Session                             │  │
│     │   User                                │  │
│     │   Partner                             │  │
│     │   ShoppingCart                        │  │
│     │   Workspace                           │  │
│     │                                       │  │
│     │   ...                                 │  │
│     │                                       │  │
│     └─────────────────────────────────────┘  │
│                                               │
└─────────────────────────────────────────────┘
```
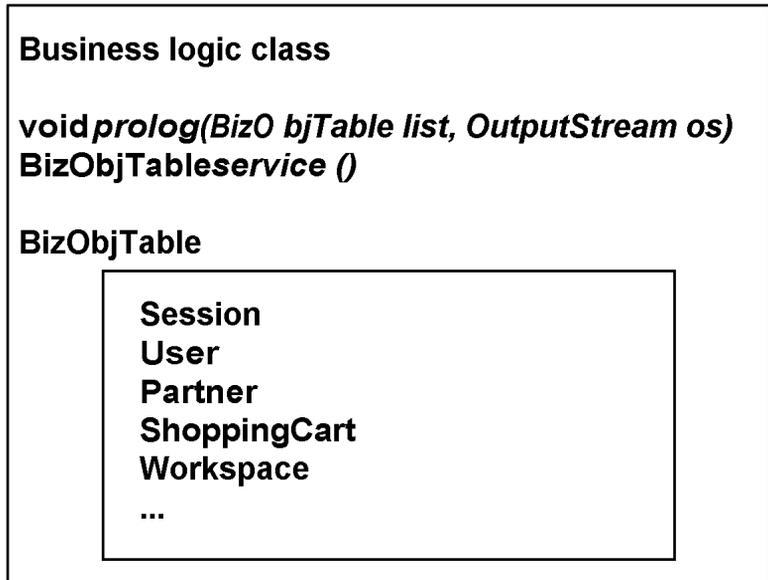
FIGURE 20. Business Logic Class

The BLC class implements *prolog()* by populating its business object table, and by recovering the session object. The business object table class, BizObjTable, is passed a vector of business objects when the controller creates the business logic class.

The BLC base class implements *service()* by returning null. Its subclasses overwrite *service()* to process the business objects.

In turn, each BLC class that extends BLC defines exactly how the *prolog()* and *service()* methods process the business object table it receives. Typically, the *prolog()* method of a BLC processes the business object table to look for an business object of the correct business object type, and then it creates a business object of that type. The *service()* method of the BLC then calls the *persist()* or *restore()* method of the corresponding business object class to either save the data to the database or to retrieve data from an external system.

Business Logic Class Example

As an example, consider the code for the BLC PriceAvailabilityGet. The purpose of this BLC is to process a request for price availability.

The *prolog()* method receives a set of business objects in the form of a BizObjTable and parses it to find a business object of type PriceAvailability. It places the business object in the m_PriceAvailability variable.

When the *service()* method of the BLC is called, it invokes the *restore()* method of the PriceAvailability object that retrieves the price availability information. A new BizObjTable is created and returned.

```
/**
* Copyright (c) 1999 Comergent Technologies Inc. All Rights Reserved.
*/
package com.comergent.dcms.blc;

import java.util.*;
import java.io.*;

import com.comergent.dcm.caf.blc.*;
import com.comergent.dcm.core.*;
import com.comergent.dcm.util.ICCException;
import com.comergent.dcm.dataservices.*

/**
 * A class which performs the pricing and availability on the given
 * shopping cart
 *
 * @author Comergent Technologies
 * @version 1.0
 * @see BLC
 */
public class PriceAvailabilityGet extends BLC
{
/**
Initialize the object with the business objects in the given vector.
@param v     the array of business objects
@param os    the output stream to which the html is written
*/
    public void prolog(BizObjTable vector, OutputStream os)
        throws ICCException
    {
        super.prolog(vector, os);
         // find the price availability BizObj in the vector
        for (Enumeration enum = vector.elements();
            enum.hasMoreElements();)
        {
            Object object = enum.nextElement();
            if (object instanceof BusinessObject)
            {
                BusinessObject bizObj = (BusinessObject) object;
                if (bizObj.getType().equals(
```

```
                              BusinessObjectTypes.PRICE_AVAILABILITY))
                       m_priceAvailability = bizObj;

               else
               {
                   throw new ICCException(
                       this.getClass().getName() + ":" +
               BLCResourceBundle.getDefaultBLCResourceBundle().getString(
                           BLCErrors.NON_BUSINESS_OBJECT));
               }
           }

           if (m_priceAvailability == null)
           {
                throw new ICCException(
                    this.getClass().getName() + ":" +
               BLCResourceBundle.getDefaultBLCResourceBundle().getString(
                           BLCErrors.NO_PRICE_AVAILABILITY_OBJECT));
           }
       }

/**
* Perform the requested operation
*
* @return a vector containing the populated PricingAvailability
* business object.
*/
       public BizObjTable service() throws ICCException
       {
/**
On the enterprise server, a restore on pa object will send a message
to the partner server and retrieve the data. On the partner server, a
restore on pa object will obtain the actual numbers from a local data
source.
*/
           m_priceAvailability.restore();
           BizObjTable vector = new BizObjTableDefault();
           vector.put(m_priceAvailability);
           return vector;
       }
       protected BusinessObject m_priceAvailability;
}
```

Global Class

The Global class is used for two main purposes: it provides access to the log stream and it provides values for system-wide parameters.

Logging

You can output logging information to the log stream by invoking one of the static log*Level* methods: *logVerbose()*, *logInfo()*, *logWarning()*, and *logError()*. These methods take the logging message as a String parameter.

You can specify a package flag as part of the method call:

```
Global.logVerbose("<Package name>", "<Log message>");
```

For example:

```
Global.logVerbose("SQLTRACE", "Starting query " + queryString);
```

By setting the names of particular package flags in the packageFlags element of **Comergent.xml**, you can manage which logging messages are generated as the Comergent eBusiness System runs. This is particularly useful if you want to customize some code, and track its behavior as you debug the customization because you can isolate the logging messages from other logging messages using a custom flag.

You can use any string you like as a package flag: for example:

```
Global.logVerbose("MyProject", "This is my customization");
```

The following flags are pre-defined in Release 7.0.2 of Comergent eBusiness System:

TABLE 19. Current Package Flags

Flag	Usage
AUTH	Authentication (user, login, logout, and so on)
CONFIGURATOR	Configurator
CONVERTER	Converter (converting XML message into internal representation)
CORE	Core packages (wrapper, dispatch servlet, and so on)
CRON	Cron jobs
DATASERVICES	Data layer
ENTL	Entitlement
Events	Event bus
GlobalCache	Global caching
LOADDATOBJ	Schema loader
MC	Not used
MESSAGING	XML messaging layer (include RosettaNet)

Flag	Usage
METADATA	Data layer
ModelTabXMLHandler	Visual Modeler
MSG	Messaging controller
MSGT	MessageTypes
PunchOut	PunchOut
RearrangeLookUpValues	Data layer
RULES	Congigurator rules
showHeader	Show incoming HTTP headers
showRequest	Show incoming HTTP request (URl with query strings)
SQLTRACE	Show SQL statements
UTIL	Not used
VM	Visual Modeler
XMLU	XML

Note that if you are using the SDK, then you can specify which flags should be logged using the DEBUG_FLAGS property set in your properties files.

Parameters

When the Comergent eBusiness System starts up, it reads in system configuration parameters from the **Comergent.xml** configuration file and its ancillary configuration files. These parameters are accessed by the Comergent eBusiness System applications using the static methods *Global.getString()* and *Global.getBoolean()*. These methods take a String as an argument: the String must uniquely identify the name of a configuration parameter.For example, the call Global.getString ("General.ServerName") returns the value of the ServerName child element of the General element defined in the **Comergent.xml** file.

If you need to retrieve a number from one of the configuration file parameters, then you retrieve it as a String using the *Global.getString()* method and then you must use one of the standard methods to convert it to a number. You must capture and handle any exceptions that can be thrown. For example:

```
int requestTimeout = -1;
String strRequestTimeout =
    Global.getString("C3_Commerce_Manager.General.-
        partnerRequestTimeout");
if (strRequestTimeout != null)
```

```
{
    try
    {
        requestTimeout = (new Integer(strRequestTimeout)).intValue();
    }
    catch (NumberFormatException nfe)
    {
        requestTimeout = -1;
    }
}
```

If the top-level element of the parameter points to an ancillary file, then the second and subsequent components of the parameter name point to elements in the ancillary file. For example, consider the call:

```
Global.getString("DataServices.General.schemaRepository")
```

This accesses the child element schemaRepository of the General element in the configuration file whose location is specified by the DataServices element of **Comergent.xml**.

Both *getString()* and *getBoolean()* can take an optional second parameter that specifies a default value to be returned if the parameter does not exist in the configuration files.

CHAPTER 41 *Upgrading Legacy Comergent eBusiness Systems*

This chapter provides a description of the legacy issues concerned with upgrading earlier releases of the Comergent eBusiness System. Current upgrades (from Release 6.7 and later are covered in CHAPTER 16, "Upgrading the Comergent eBusiness System".

Overview of Upgradability

The Comergent eBusiness System has been designed from the ground up to meet the dual challenges of providing out-of-the-box application functionality against common business scenarios while providing the necessary flexibility to handle the extensions and customizations that occur in the normal course of deployment.

The software installation includes all the necessary source code, configuration files, data initialization scripts, and other tools necessary to perform the kinds of modifications described in this *Comergent eBusiness System Developer Guide*. This chapter describes upgrade considerations and process for each supported customization technique as described in CHAPTER 15, "Tailoring the Comergent eBusiness System".

The Comergent eBusiness System System supports a progressive sequence of customization techniques designed to make the most common customizations the easiest to implement initially and to roll forward during an upgrade.

Customer Upgrade Scenarios

This section enumerates and explores the common reasons for upgrading the Comergent eBusiness System. The motivations for upgrade can directly impact the style and scope of the upgrade activity.

Upgrade Motivations

The following motivations are considered and occasionally referenced in the material which follows:

- Upgrading to obtain stability and performance benefits from the latest release.

- Upgrading to implement a new module which requires the latest Platform version.

- Upgrading to obtain across the board functionality and/or usability enhancements.

- Upgrading to obtain additional platform or standards support.

- Upgrading to implement a specific feature enhancement which involves inter-module communication and interaction.

- Upgrading to obtain inter-enterprise communication enhancements.

- Upgrading to obtain enhanced administration tools.

Upgrade Considerations by Customization Technique

The following categories of customization are supported by the Comergent eBusiness System architecture.

Upgrading Presentation

Presentation is typically the most extensively customized area. At minimum, the application of custom-branding is generally required during deployment to make the reference UI conform to the customer's Web UI style guidelines and standards.

Re-applying existing customizations to a new version of the Reference UI requires the fairly manual but mechanical process of comparing and merging individual JSP pages and Controller files. In that upgrade scenario, upgrade cost will be directly proportional to the extent of page reorganization and modification.

Alternatively, upgrade may focus on retaining the previously customized UI on the latest server-side APIs with minimal functional modifications to existing pages. This may be the case where upgrade is driven primarily by the desire to implement additional Comergent eBusiness System modules. In this case, upgrade focuses on backward compatibility of existing, customized JSP pages and Java-based controllers.

Upgrade Considerations for Customized JSP Files

The Model 2 JSP architecture is the foundation of Presentation in Comergent eBusiness System Releases 4.x and higher. This employs a Model/View/Controller, or MVC, design pattern with JSP as the Web page templating language and Java-based Controllers as the orchestrators of page flow and request routing and processing. JSP pages have dependencies on specific Data and Logic Beans, for example, the Order Detail page depends on the OrderPresentationBean from which it acquires its data for display. The Order detail page itself controls specific layout and display of the Order. In this case, the OrderPresentationBean provides both formatting specific logic and access to the underlying Order business data.

The following are typical presentation customizations which must be re-applied or otherwise accounted for during upgrade.

- Application of custom branding in the form of images and text styles
- Addition of surrounding page content to the reference UI, for example, adding a site-wide navigation frame and branding header to Comergent eBusiness System Order Management
- Re-organization of page content
- Page flow modification

Page flow changes are accomplished by modifying or creating new Java Controllers, so typically some degree of Controller customization is performed along with JSP customization.

Administration pages are not intended to be branded or otherwise customized and therefore should not be affected during the upgrade process.

Specific Considerations for Upgrading Presentation for Release 3.x

Upgrading HTML Templates to JSP

Releases of the Comergent eBusiness System prior to Release 4.x predated the availability of mature, third-party servlet containers. Instead a lightweight,

Comergent Technologies proprietary, JSP-like presentation templating scheme overlaid on HTML was implemented and employed by those early releases of the Comergent eBusiness System. With Release 4.x, these proprietary implementations of presentation templating and request dispatch were replaced with standard JSP and servlet container usage and dependence (see "Overview of changes for Releases 4 and 5" on page 478). Along with the many benefits of this necessary technology transition, a consequence of the shift is that upgrading presentation from Release 3.x to Release 5.x will require re-application of existing presentation customizations to the new JSP format. This may result in roughly the same cost as the initial implementation for the presentation portion.

Also prior to Release 4.x, page flow and parameter validation was orchestrated by lightweight java logic classes known as Presentation Logic Classes, or PLCs. Page flow customization implemented through PLCs will have to be re-applied to Java Controller classes under the JSP Model 2 Architecture of Release 4.x and higher. The implementation effort will range from simple cut-and-paste of source code to re-implementation of the original page flow customizations.

An additional complication is that the end-user User Interface was re-designed for Release 5.x based on customer and user feedback on early deployments.

For all the reasons previously noted, the full scope of presentation upgrade from Release 3.x is likely to amount to re-implementation of the original presentation customizations.

Upgrading Business Objects and XML Messaging

Data objects are essentially XML-based data maps, mapping the logical schema used by applications to the actual data sources, local and remote, within the customer deployment environment. As such, they are easy to extend, and the cost of performing the mappings comprises a small fraction of the cost of a complete deployment. Upgrading consists of re-applying the original mappings by merging the existing schema mapping files into the new reference schema mapping files. Using standard file merge tools, this process should result in few conflicts and thus require minimal manual intervention.

Starting with Release 5, data object extensions can be accomplished through an inheritance mechanism which isolates extensions into separate XML schema map files. Customizations done in this way need only be reapplied by copying the files over to the new installation. An object factory mechanism known as the Object Manager is used to enable the product code to create and manipulate the customized, superclassed data objects.

Data objects are often directly associated with XML message definitions to support integration with internal and external data sources. At runtime, incoming XML messages are converted to Business Objects by an extensible Converter subsystem. Conversions are implemented as XSL/XSLT-based XML translations. Conversely, outgoing XML messages are generally converted from data objects to a corresponding XML message format, also by the Converter.

In general, standard message formats, including Comergent Technologies' dXML formats, are supersets of all possible fields required to execute particular operations. For this reason, it is unlikely that any message mapping customizations will have to be re-applied during upgrade. Existing conversions can simply be re-registered in the new system. In cases where a new release contains messaging version upgrades, backward compatibility conversions are supplied to support messaging with previous message versions. This means that existing message-based integrations need not be upgraded unless features of the new messaging are desired. Proprietary message formats can be similarly supported during upgrade.

Upgrading Business Logic

Business logic modifications in the form of Java source code become necessary during customer deployment for a variety of reasons including

- To modify existing decision logic (conditionals), possibly to include custom data fields in the decision logic

- To alter default behaviors that are not controllable via system configuration

- To enhance default behaviors where product functionality falls short of the specific deployment needs

- To enhance default behaviors to align with functionality needed now but planned for a future product release

- To implement new logic specific to a given deployment

- To implement custom Data Services for data integrations specific to a given deployment

- To modify Controllers to alter page flow, parameter validation and error handling behavior (see "Upgrading Presentation" on page 472)

As with other aspects of upgrade, the general upgrade approach is to reapply existing logic customizations to the new release. The effort needed to bring logic modifications forward to the latest release is directly dependent on how those modifications were originally implemented. Modifications in the field are

performed in a number of ways which have progressively more impact on upgrade. In all cases, if the logic modifications reference customized data objects, those data object customizations must first be brought forward for the logic to execute properly. The following is a brief summary of upgrade impact for each customization technique:

- Extending an existing or implementing a new Data Service – changes of this kind should be forward compatible.

- Extending existing logic to add or change behavior without any product source modifications – changes of this kind should be forward compatible with the caveat that the sub-classed method or class may become deprecated in a future release.

- Extending existing logic to add or change behavior by creating a new method in the product source – changes of this kind should be forward compatible when approved by Comergent Technologies Support with the caveat that the sub-classed method or class may become deprecated in a future release.

- Implementing distinct, new functionality by creating new message types and logic handler classes – changes of this kind should execute in the new release; however, changes in underlying product behavior may break the customization or render it obsolete.

- Overloading existing logic by replicating product source in a subclass – changes of this kind should execute but will hide any enhancement to the base class implementation which is likely to create problems at runtime and hurt supportability of the deployment going forward. In most if not all cases, the overload can either be discarded, if made obsolete by the latest release, or should be converted to a proper extension of the product code for increased upgradability going forward.

- Modifying product source directly – this should never be done without a specific strategy to intersect with a supported patch or future product release of the Comergent eBusiness System.

Other Considerations for Upgrade

In addition to migrating customizations forward, the following areas of deployment activity have considerations for upgrade.

Data Loading and Migration

Migration of business data from one schema version to another is a fairly straightforward process generally best handled by SQL scripts based on table level

comparisons of the releases. Where tables have been merged or schema names and types altered, Business Object level (XML) migration is a good option, which may be preferable for a future release. Business Object changes are documented for each new release.

In some cases, Comergent Technologies provides a set of reference migration scripts which will migrate data from the reference version of a previous release to a reference version of the current release. They are intended to serve as a foundation from which specific migration scripts can be derived. Automated and manual comparison of the source schema version against the target schema version would be performed in order to produce a final migration SQL script. The magnitude of effort would be roughly one to two person-weeks.

Customizations to scripts to accomplish the creation of the underlying schema must be re-applied to the new reference schema creation script.

Configuration and Converter Migration

Modifications made to configuration files to support logic customizations during deployment must be reapplied for those logic customizations that remain relevant. Again, this is a simple matter of file merge for those files modified. Configuration modifications include any updates needed to the Converter Map to pick up any custom XSL/XSLT conversions. In all cases, configuration files are in XML format.

A Sample Upgrade Task Flow

The following is a likely sequencing of top level activities in performing an upgrade of the Comergent eBusiness System:

1. Installation of the latest release.

2. Data migration.

3. Reapplication of Professional Services Stored Procedures.

4. Reapplication of Business Object customizations.

5. Reapplication of Messaging customizations.

6. Reapplication of Business Logic customizations.

7. Reapplication of Presentation customizations.

8. Test, certify, rollout.

Specific Upgrade Scenarios

Overview of changes for Releases 4 and 5

Major changes to the underlying technology, user interface, and page flow occurred in Releases 4 and 5. These pose particular challenges to upgrade of previous releases.

Changes for Release 4

To bring Release 4 into compliance with maturing technologies under the Java 2 Enterprise Edition (J2EE) standards umbrella, a major technology shift from Release 3 was effected, namely:

*   The replacement of proprietary request dispatch mechanism (Listener/ Dispatcher) with reliance on and conformance to the J2EE servlet container API.

*   The replacement of proprietary presentation template processing with Java Server Pages (JSP).

*   To enable JSP-based presentation, statically and dynamically generated Java Bean wrappers around Release 3 Business Objects were employed by applications. This represents a shift from pure message-based dispatch of logic to a mix of method and message-based invocation.

Although Release 4 maintained backward compatibility support for the business logic and presentation application frameworks of Release 3, the previous mechanisms are deprecated and largely incompatible with the new ones. Further, underlying logic differences in Release 4 may cause Release 3-based logic customizations to behave differently than intended or not at all without some modification. For this reason, re-applying existing customizations is only recommended for code areas where further development is not expected, that is, for stable and isolated functions which are themselves not expected to require further development. For all other customizations, re-implementation in the latest framework is recommended to insure the maintainability of those customizations going forward.

Changes for Release 5

No fundamental technology shifts were made for Release 5; however, significant refinement of in the form of enhancement of platform services and a redesign of the end-user facing User Interface was performed.

In particular, the following changes are of interest with respect to upgrade:

- Business Objects continued their shift towards Java Beans through Data Integration Layer enhancements to support dynamically generated, updateable beans with inheritance. For applications, all new and most existing applications were implemented or modified to rely on the new Release 5 Business Objects.

- To enhance usability and ease integration, the base User Interface for end-user pages was redesigned to be frameless and to support task-based user flow via a wizard metaphor. The changes are most significant for Order Management customers.

- To leverage standard technology, the Converter Service was redesigned to be based on XSL/XLST transformations. This enables XML to XML conversions to be implemented as declarative descriptions (Style Sheets), removing the need for programming in order to accomplish and maintain messaging transformations.

The direct impact of the Business Object enhancements on customizations is limited as Release 5 is backward compatible with previous schema definition formats. However, it is recommended that customizations that will be modified as part of an upgrade also be re-implemented to use the Release 5 Business Objects or Data Beans. In particular, Business Object customizations previously made directly to a particular reference Business Object definition should be reapplied to the equivalent Release 5 definition via the new sub-classing mechanism to increase maintainability going forward.

The redesign of the Release 4 User Interface for Release 5 will further complicate any upgrade of presentation from Release 3 since both the syntax and structure of the pages will differ. Essentially, reapplying User Interface customizations will be roughly equivalent to re-implementation of those customizations in Release 5 excepting that the overall scope of changes should be significantly reduced by an improved, frameless Release 5 starting point.

Upgrading Release 3

As previously stated, major changes to the underlying technology and user interface and page flow occurred in Releases 4 and 5 of the Comergent eBusiness System. For Release 3, these add up to an upgrade effort that will likely be tantamount to re-implementation with respect to coding effort. Design effort should be substantially reduced as the initial implementation generally involves significant analysis of and some refinement to existing business practices. The specific, top level task flows for each area of customization are enumerated below.

Data Migration

Data Migration of Release 3 to Release 5 will consist of the following steps:

1. Creation of the Release 5 schema in a new table space or database via supplied script.

2. Manual construction of SQL-based conversion scripts specific to the deployment from analysis of the new Release 5 schema.

3. Execution of those scripts to migrate existing business data to the new Release 5 schema.

Business Objects and Messaging

Upgrade of Business Object customizations from Release 3 to Release 5 will consist of the following steps:

1. Install Release 5 on a development server.

2. Perform a difference analysis between deployed schema and Release 5 reference schema.

    Depending on the scope of the changes, the following approaches may be taken:

    a. Reapply the changes through standard file compare/merge techniques.

    b. Re-implement the changes via inheritance starting from the Release 5 reference schema and using the deployment schema definitions as a guide.

3. Messaging and Converter customizations will be compatible with Release 5. However, for maintainability going forward, it is advisable to upgrade any Java-based conversions to XSL/XSLT-based conversions. This will enable the conversions to be maintained through updates to the XML-based conversion map instead of any Java programming.

Business Logic

Although Release 3 Business Logic Extensions should execute without modification in Release 5, changes to logic flow in the referenced APIs underlying the customizations are likely to be so substantial as to cause some degree of regression. In those cases, re-implementation in Release 5 is recommended over reapplication of Release 3 changes to Release 5 to increase maintainability going forward.

In cases where reapplication of Business Logic from Release 3 to Release 5 is advisable, the upgrade will consist of the following steps:

1. Perform migration of data, Business Objects, and Messages on a development server.

2. Copy customized BLCs and Java classes to an executable location on development server.

3. Reapply message mapping modifications to link in the customized BLCs.

4. Start Release 5 and unit test BLCs with XML messages or UI.

5. Evaluate degree of regression and consider re-implementation where regression is substantial or customization was minor.

Presentation

Due to the substantial changes made to both the implementation technology and design of the User Interface since Release 3, reapplication of Release 3 presentation templates to Release 5 is not recommended. Instead, presentation should be re-implemented starting from the Release 5 reference UI. Because the Release 5 User Interface is JSP-based, frameless, and improved with respect to usability, it is anticipated that far fewer modifications will be necessary during upgrade.

Upgrading Release 4.x

As described in "Changes for Release 5" on page 478, major usability enhancements were made in the Release 5 release, including moving all end user facing pages to a frameless UI. Modifications to JSP pages written in the Release 4 release will have to be mapped conceptually to the new reference UI to determine if they are still relevant. Simple branding and style modifications should be reapplied to the new UI. More substantial changes may no longer be relevant or may need to be re-implemented under the new UI scheme in order to properly leverage the new Release 5 UI. Modifications to page flow made to Java Controllers will have to be reapplied to Release 5. Due to the nature of UI page flow changes for Release 5, it is unlikely that customized Release 4 Controllers will execute without modification within Release 5.

Data Migration

Data Migration of Release 4 to Release 5 consists of the following steps:

1. Creation of the Release 5 schema in a table space or database via supplied script.

2. Manual construction of SQL-based conversion scripts specific to the deployment from analysis of the new Release 5 schema.

3. Execution of those scripts to migrate existing business data to the new Release 5 schema.

Business Objects and Messaging

Upgrade of Business Object customizations from Release 4 to Release 5 consists of the following steps:

1. Install Release 5 on a development server.

2. Perform a difference analysis between deployed schema and Release 5 reference schema.

    Depending on the scope of the changes, the following approaches may be taken:

    a. Reapply the changes through standard file compare/merge techniques.

    b. Re-implement the changes via inheritance starting from the Release 5 reference schema and using the deployment schema definitions as a guide.

3. Messaging and Converter customizations will be compatible with Release 5. However, for maintainability going forward, it is advisable to upgrade any Java-based conversions to XSL/XSLT-based conversions. This will enable the conversions to be maintained through updates to the XML-based conversion map instead of any Java programming.

Business Logic

In many cases, Release 4 Business Logic Extensions should execute without modification in Release 5. However, there may be cases where changes to page flow for improved usability caused underlying logic flow changes which render a particular Release 4 logic customization incompatible with Release 5. In those cases, re-implementation in Release 5 is recommended over reapplication of Release 4 changes to Release 5 to increase maintainability going forward.

In cases where reapplication of Business Logic from Release 4 to Release 5 is advisable, the upgrade consists of the following steps:

1. Perform migration of data, Business Objects, and Messages on a development server.

2. Copy customized BLCs and Java classes to an executable location on development server.

3. Reapply message mapping modifications to link in the customized BLCs.

4.   Start Comergent eBusiness System Release 5 and unit test BLCs and Java classes with XML messages or UI.

5.   Evaluate degree of regression and consider re-implementation where regression is substantial or customization is minor.

Presentation

In upgrading Release 4 to Release 5, impact to customized presentation can vary widely depending on the nature and purpose of the Release 4 customizations performed (see "Upgrading Presentation" on page 472). In particular, the following customization scenarios will lead to distinct upgrade paths:

•   *Case 1:* Customization was done primarily to apply branding and UI basic style conventions.

•   *Case 2:* Customization was done to remove frames and modify page flows in addition to application of branding and style.

•   *Case 3: C*ustomization involved a major reworking of the Comergent eBusiness System UI to conform to a substantially different corporate look and feel and/or to support substantial functional customizations during deployment in addition to application of branding and style.

The following sections detail the upgrade path for each of these cases.

Presentation Upgrade Case 1

Because the Release 5 UI redesign included the switch to a frameless UI, reapplication of branding should be more straightforward and less effort than that for the original deployment. The following steps will be necessary:

•   Perform migration of data, Business Objects, and Messages on a development server.

•   Perform upgrade and migration of any Logic customizations on which the presentation customizations rely.

•   Migrate customized graphics to the Release 5 development server.

•   Working from the Release 5 reference UI, re-apply branding headers as included HTML or JSP pages to the reference JSPs.

•   If image names were changed during customization of Comergent eBusiness System Release 4, then fix up any broken image references by renaming the customized image files, not the Comergent eBusiness System Release 5 image tag references.

- Start the server and test the new UI, noting any regressions. Identify any new Comergent eBusiness System reference buttons that will require re-branding.

- Identify and reapply any relevant changes to text descriptions, column header names, labels, etc. from the previous deployment to the new JSP pages.

Presentation Upgrade Case 2

Presentation customizations done to remove frames will not need to be reapplied since Release 5 reference UI is frameless. The following steps will be necessary:

1. Ignoring any customizations performed to remove frames, perform migration and upgrade of UI branding and style to Release 5.

2. Re-evaluate page flow customizations. Page flow improvements in Release 5 should render usability oriented page flow customizations obsolete. Discard those going forward. Page flow customization performed to insert additional pages into the flow to model customer-specific business process must be reapplied as follows:

   a. Convert the pages as appropriate to conform to the look and feel of the upgraded UI in Release 5. For example, remove frames if the added pages contained them.

   b. Reapply the Controller customizations to Release 5 reference Controllers via file merge/compare techniques. If minimal changes to flow were done or flow changes were performed through sub-classing of Controllers, migrating the customized Controllers as is to the development server is an option.

   c. Regress the changes and bug fix as needed.

Presentation Upgrade Case 3

Where substantial reworking of the UI has been done without regard to the resulting upgrade implications, presentation customizations must be either wholly re-implemented or migrated from the perspective of preserving the customized UI as is.

If the upgrade is timed to coincide with a general Web site face lift, then re-implementation of the UI customizations will be required in any case and should represent a small portion of that overall cost. If the overall site UI is not due for redesign, then it may be more cost effective to ignore any UI enhancements and

attempt to migrate the old UI to the new installation. For migration of the old UI, the following steps will be necessary:

1. Copy the previously customized UI including JSP pages and Controllers to the Release 5 development server.

2. Reapply message mapping changes to the Release 5 development server.

3. Regression test the old UI on the new Release 5 platform. Regression will occur where logic APIs have changed from Release 4 and backward compatible, deprecated APIs were not maintained. Repairing regression will consist of "rewiring" the old UI to the new Release 5 APIs.

4. Evaluate the scope of repairing the old UI on the new installation. If for any reason, the projected scope of repairing regression approaches or exceeds the cost of re-implementation, discontinue migration and perform re-implementation.

Upgrading Release 5.x to future Releases

Release 5 resulted in a new level of maturity with respect to both the features of the application suite itself and to the relatively young industry standard platforms on which it is built. Along with refinements to deployment process and methodology including the release of the Software Developer Kit (SDK), upgradability is also substantially improved over previous releases. The following sections describe some of the upgrade related improvements Release 5 customers will be able to leverage if customization procedures as outlined in this document are consistently followed.

Data Migration

For future releases, a roadmap of improved reference migration scripts and tools are planned including reference XML-based data bridges. These should substantially ease both migration of data for upgrade and implementation of data loading and synchronization from new sources.

Support and tools for automated schema profiling and data migration will be added to the reference migration process enabling both better analysis and smoother execution of data migration.

Business Objects and Messaging

Future releases will include automated support for profiling and performing schema map migration. These will further leverage the rich potential of the XML-based mappings by providing structured comparison and merging of changes.

Going forward from Release 5, all custom message conversions should be performed in XSL/XSLT removing programming from the maintenance and upgrade process.

Business Logic

Future releases will increase the emphasis on specific internal and external interface connections between modules and applications. This will enable automated profiling of Logic changes to greatly improve analysis and execution of upgrades.

The emphasis on module interfacing will enable improved documentation and deprecation of the Business Logic APIs to greatly extend backward compatibility coverage at the Business Logic level.

Additional user extension techniques and mechanisms including a formal User Callout mechanism will be provided to increase the isolation of customizations from the product code.

Presentation

A maturing and improved UI will require fewer if any structural modifications during deployment. A UI Style Guide will streamline the look-and-feel analysis and design phase of deployments.

Web UI presentation technology will mature to enable UI component definition and reuse. This will centralize and isolate look and feel changes reducing both development and maintenance costs related to presentation customization. For example, XML Style Sheets will likely play a greater role in controlling page formatting and layout, separating page appearance from structure.

The emphasis on more formal modularity and API versioning will enable future release to remain compatible with old UIs. This will not only ease migration to newer releases but also give more scope control to the customer during an upgrade project. For example, pages may not have to be altered during an upgrade, but the customer may choose to change or enhance them anyway to achieve improvements ultimately motivated by business goals.

Index