Visual Modeler

Implementation Guide

Release 9.1

Copyright

Contents

Introduction

The topics in this section of the guide provide information required for you to implement the Visual Modeler at your enterprise.

Implementation Methodology

The Visual Modeler implementation methodology consists of phases that ensure that implementation can be planned and tracked through to completion.

The Visual Modeler Implementation Methodology table provides a summary of the phases and the activities to complete in each phase. A standard set of documents can be used to track each phase.

Implementation phase	**Description**
Plan	Plan the implementation: set a timeline, milestones, and identify risks and dependencies
Analyze	Organization and administration, define business rules, user interface, messaging protocols, data sources, e-commerce flow planning, training needs, rollout strategy, environment preparation, operations planning
Design and configure	Installation, configuration, integration, unit testing, and training development
Test and deploy	Testing server configuration, enterprise to partner communication, partner to enterprise communication; cut over to production systems, distributor training, documentation delivery, support
Improve	Ongoing enhancement activities, partner training, and support

Implementing the Visual Modeler Integration

The Visual Modeler is designed to integrate channel partners into an e-commerce network. Organizations in the network act as enterprises and partners. Each organization acting as an enterprise installs their copy of the enterprise server to transfer information to their channel partners seamlessly.

Each reseller or distributor may work with more than one enterprise, and their installation of the enterprise server must be able to receive and respond to messages from different enterprise servers.

The following table summarizes the main activities for an implementation of the Visual Modeler:

Implementation phase	**Task**
Plan	Project analysis
Analyze	◆ Configuration analysis ◆ Integration analysis ◆ Requirements analysis
Design and configure	◆ Preparation of servlet container environment ◆ Installation of Knowledgebase ◆ Knowledgebase setup ◆ Visual Modeler configuration ◆ Role and security definition ◆ System administrator authentication ◆ XML schema creation ◆ Customizing of BizAPIs, BLCs, and controllers ◆ Customizing JSP pages
Testing and deployment	◆ Product integration ◆ Testing server configuration ◆ Testing enterprise to partner communication ◆ Testing partner to enterprise communication ◆ Release to production systems
Improve	Assess and enhance

Implementation Steps

The main tasks you perform in implementing Visual Modeler are:

- Project analysis: Agree to a schedule for the implementation project that sets a timeline. Identify milestones to measure the progress of the implementation and identify dependencies and risks that might prevent the implementation from completing on time.
- *Configuration analysis*: Determine a suitable Visual Modeler configuration (the number of machines to be used and their location on internal networks in relation to firewalls and proxy servers). See "High Availability and Load Balancing" for further details about a clustered implementation.
- *Integration analysis*: Identify integration points with existing e-commerce systems.
- *Requirements analysis*: Check hardware and software requirements to make sure that the machines are sufficiently powerful to support the anticipated traffic and response times required.
- *Installation of Visual Modeler*: Install the Visual Modeler on the designated machine(s). See "Installation Overview" for more information.
- *Knowledgebase setup*:

  a. Installation of Knowledgebase: Install the Knowledgebase schema in the designated database server.

  b. Knowledgebase setup: Check connectivity to the Knowledgebase database server and populate it with all your e-commerce-related information. This must include the partner profiles for your partners, your product catalog, and price list information.

  See the following sections for more information:

  - "Gathering the Database Information"
  - "Creating the Knowledgebase Schema"
  - "Populating the Knowledgebase"
  - "Logging in to the Visual Modeler"
- *Visual Modeler configuration*: Modify configuration files to define the system configuration in your production environment.
- *Role and security definition*: Define groups and roles and modify configuration files and ACL scripts accordingly. These determine the security privileges for your enterprise server users.
- *Schema creation*: Create the business object schema to provide data source information. The data layer manages access between the enterprise server and the external systems.
- *Customizing BLCs and controllers*: Modify business logic and controller classes to support your business logic. In some cases, you need to modify the Java classes in order to implement business processes specific to your organization.
- *Customizing JSP pages*: Modify templates to meet your “look-and-feel”, search, and static page requirements. The JSP pages provided by the Visual Modeler are used to display the browser pages and may be customized to meet the needs of your organization.
- *Product integration*: Import product information into the Knowledgebase or provide punch-out integration. If your implementation is to support product ordering from a non-Sterling product, then you need to provide a means of integrating the product data with the Visual Modeler.

- *Testing server configuration*: Before you deploy the Visual Modeler, thoroughly test the system. We provide a number of scripts to test the chief functional components.
- *Testing enterprise to partner communication*: Send test messages from the enterprise server to other enterprise servers.
- *Testing partner to enterprise communication*: Send test messages from other enterprise servers to your enterprise server.
- *Assess and enhance*: Once the Visual Modeler is deployed, you must plan for an ongoing process of analyzing its usage and performance.

Integrating the Visual Modeler with IBM Sterling Selling and Fulfillment Foundation

In some instances, complex products may have to be configured before they can be bought by customers. In some other instances, such products may have optional components that customers can configure based on their requirements. Visual Modeler enables you to create models that define the configurable options of a product, and to associate products to these models. The IBM® Sterling Configurator is a tool that is used to display the configurable products along with the available options to the end user.

The integration between the Visual Modeler and IBM® Sterling Selling and Fulfillment Foundation is necessary to enable them to exchange information. The integration is required to ensure that the correct product information, as maintained in Sterling Selling and Fulfillment Foundation, is used for defining the models in the Visual Modeler. The prices applied on the products are based on the price list and the currency associated with the guest user. For more information about associating the price list, refer to the *Sterling Business Center: Pricing Administration Guide*.

To integrate the Visual Modeler with Sterling Selling and Fulfillment Foundation, you must perform certain configurations in the Visual Modeler application and the Applications Manager.

Configuring the Visual Modeler Properties

You must configure the values of certain properties in the Visual Modeler in order to enable it to obtain the correct product information from Sterling Selling and Fulfillment Foundation.

To configure the properties in the Visual Modeler:

1. Point your browser to the following URL:

   `http://<hostname>:<port>/<context_root>/en/US/enterpriseMgr/admin`

   Here, hostname is the IP address, port is the listening port of the machine in which the Visual Modeler is installed, and context_root is the context root of the hosted Visual Modeler application.

   The Login page is displayed.
2. Log in as an administrator by entering your login ID and password, and clicking **Log In**.
3. Click the **System Services** hyperlink. The System properties page is displayed.
4. Click the **Fulfillment** hyperlink. The Properties for Fulfillment page is displayed.
5. Set the Sterling Order Fulfillment System URL property to `http://<hostname>:<port>/smcfs/interop/InteropHttpServlet`. This URL pertains to the Interop servlet of the Sterling Selling and Fulfillment Foundation.

6. Set the Sterling Configurator URL property to:

   `http://<hostname>:<port>/sbc/configurator/configure.action`

   Here, hostname is the IP address of the machine in which the Sterling Selling and Fulfillment Foundation is installed, and port is the listening port of the machine in which the Sterling Selling and Fulfillment Foundation is installed.

7. Set the following properties appropriately:

   - User name for the Sterling Fulfillment system
   - Password for the Sterling Fulfillment system

   The values of these properties determine the user name and password that will be used to communicate with the Sterling Selling and Fulfillment Foundation server.

Configuring the IBM Sterling Configurator Rules

To enable the Sterling Configurator to obtain the model information of the products from the Visual Modeler, you must specify the location of the models, properties, and rules pertaining to models in the Applications Manager.

To configure the Sterling Configurator rules:

1. In the Sign In page, log in as an administrator by entering your login ID and password, and clicking **Sign In**. The Application Console home page is displayed.
2. From the menu bar, navigate to **Configurations** > **Launch Applications Manager**. The Applications Manager is launched in a new browser window.
3. From the Applications Manager menu bar, navigate to **Applications** > **Application Platform**. The Application Rules side panel is displayed.
4. In the Application Rules side panel, select **System Administration** > **Item Configurator**.
5. Specify the paths to the location where the models, properties files, and rules are stored.

Notes:

- All the paths specified in the Applications Manager for the model repository are shared by the Sterling Selling and Fulfillment Foundation and the Visual Modeler. If the Sterling Selling and Fulfillment Foundation and the Visual Modeler reside on different machines, the paths should be mounted on a drive that is accessible to both. For more information about model repository, refer to the *Sterling Selling and Fulfillment Foundation: Application Platform Configuration Guide*.
- In IBM® Sterling Business Center, a model can be assigned to the item definition of a bundle item. The model name is saved in the item definition. If you change the model name at any point of time after it has been saved to the item definition, the item definition needs to be changed to point to the modified model name. This situation could arise when a user edits the model definition in Visual Modeler.

Introduction to J2EE Web Applications

This topic 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 topic.

Architecture

The Visual Modeler 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 Visual Modeler 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 users' 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 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.

  Note: In the Visual Modeler, 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:

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

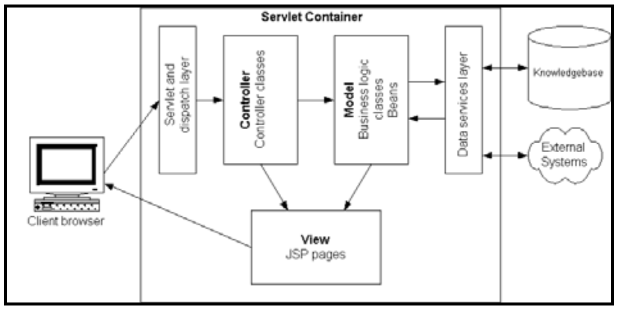
Data is passed to a JSP page using a variety of mechanisms, the most important of which are implicit objects and beans.

- 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 Visual Modeler 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.

The following figure illustrates the arcitecture of the model:



- *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 Visual Modeler 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 finishes 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:

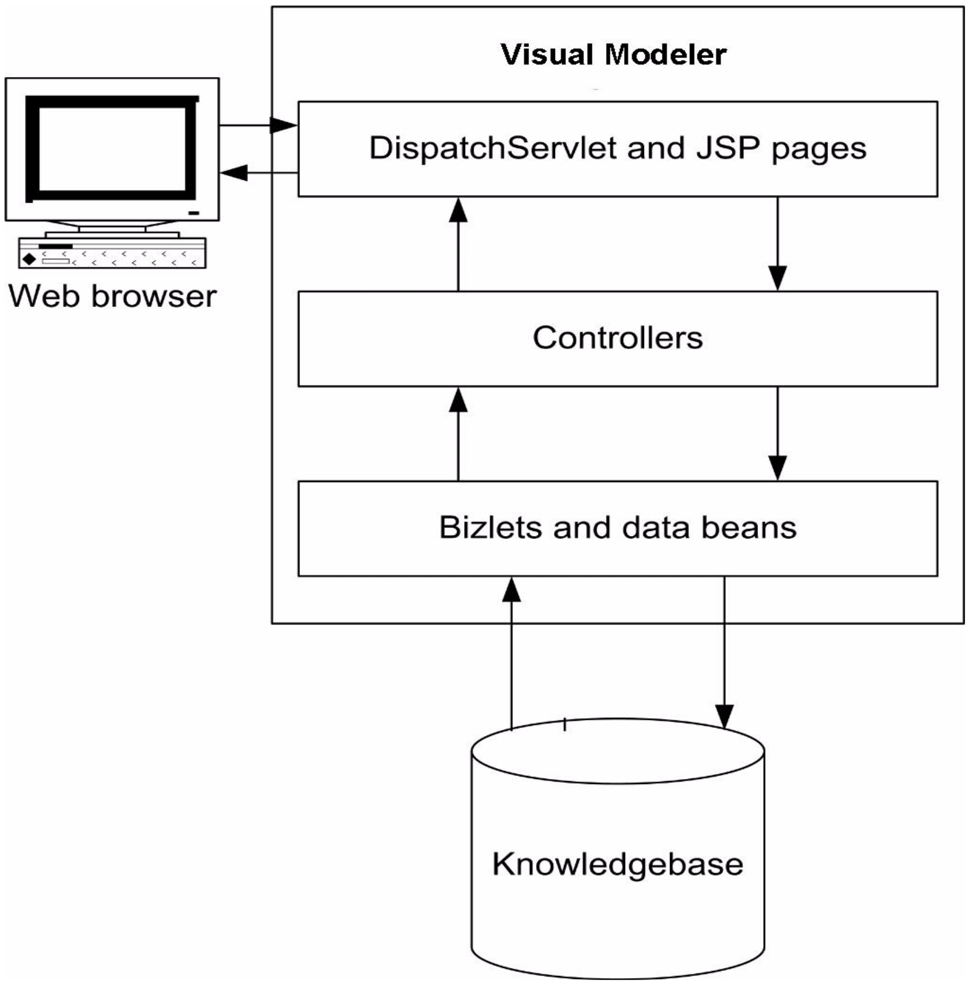
- Hall, *Core Servlets and JavaServer Pages*, Second Edition, Prentice Hall, 2003
- Hunter, *Java Servlet Programming*, Second Edition, O’Reilly, 2001
- Fields and Kolb, *Web Development with JavaServer Pages*, Second Edition, Manning, 2001

System Architecture

This topic describes the Visual Modeler architecture and introduces some of the important Java classes that the Visual Modeler and its applications use. It assumes a thorough understanding of the J2EE architecture.

This topic is intended to help you to modify or extend existing applications or write new applications. Note that not all parts of the Visual Modeler conform to this architectural description.

The following figure displays theVisual Modeler Architecture.



Visual Modeler Web Application

When you install the Visual Modeler into your servlet container, it installs as a WAR file, **Sterling.war**. When the WAR file deploys, it unjars into a directory called **Sterling/**. The **WEB-INF/** sub-directory contains the **web.xml** file for the application.

The most important configuration settings in this file are:

- The definition of the InitServlet and DispatchServlet:
  - InitServlet loads when the servlet container starts. InitServlet reads in all of the configuration information for the Visual Modeler using the value of the propertiesFile element: by default this is **Comergent.xml**.

  - 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 Visual Modeler 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 Visual Modeler. 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.

Processing Requests

When the Visual Modeler 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:

```
../Sterling/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.

Note: 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" 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.
- 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.

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

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, 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 Visual Modeler applications all have the same structure: they are composed of controllers, business objects and bizlets, and JSP pages.

Wrapper Classes

Several of the standard classes used in J2EE Web applications have been wrapped in 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 Visual Modeler. 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 comergentRequest)
  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.
- DebsDispatchServlet: this servlet is used to process XML messages posted from another system to the Visual Modeler. 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 Visual Modeler 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 Visual Modeler 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 Visual Modeler. 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 "Introducing Data Beans and Business Objects" topic 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 Visual Modeler XML schema may be used to generate a DataBean class by running the generateBean target (see the "Software Development Kit" topic 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" topic for a discussion of how these beans are used.

ObjectManager and OMWrapper Classes

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*/Sterling/WEB-INF/properties/**) to determine which type of object is created from the object name provided in the *getObject()* method.

Note: 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"), 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", then a new object is created rather than re-using an object from the pool.

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 Visual Modeler 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.

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

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".
- *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. Its logging function has been replaced by the log4j API: see "Logging" topic 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

Use this interface to define a cache that provides access to cached objects used by all Visual Modeler applications. It can be used to support a clustered environment in which the Visual Modeler 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 Visual Modeler: com.comergent.dcm.cache.impl.AppContextCache.

You access the implementation of the GlobalCache interface by:

```
GlobalCache globalCache = GlobalCacheManager.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.

Preferences Class

The Preferences module provides the mechanism for accessing Visual Modeler properties. It is one of the modules provided in the platform modules: see "Preferences Service" 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 Visual Modeler 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, suppose 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.myApplication.MyClass.class);
String temp_MyPropertyString =
temp_Preferences.getString("MyProperty");
```

Transactions

The Visual Modeler 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.

Support for Lookup Codes

The Visual Modeler uses lookup codes to provide a mechanism for maintaining and displying 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.

What lookup support does the Visual Modeler provide?

The Visual Modeler 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.

Platfform Modularity

The Visual Modeler modular architecture is designed to make implementations easy to customize and upgrade. This topic provides an overview of modular architecture, platform modules, and the module interfaces, and descibes each module. It covers the following topics:

- XML Message Converter
- XML Message Service
- XML Services

Overview

The Visual Modeler platform architecture enables building 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.

The benefits of providing a means of delivering platform functionality in platform modules and requiring that modules make calls to other modules only through their external interfaces areas follows:

- It is easier to compartmentalize the functionality of applications.
- It is easier to understand and manage the dependencies between parts of the Visual Modeler.
- 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 Visual Modeler.

Platform Modules

The Visual Modeler 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 Visual Modeler such as a service or a component of the Visual Modeler 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 are assembled into 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 the 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.

Access Policy

This module provides the service used to check access policies.

Authentication

This module provides the APIs used to authenticate credentials and users.

Base64

This module provides the classes used to convert data to and from Base 64 notation.

Classpath Appender

This module provides classes used to add paths to the classpath.

Cryptography Service

This module provides the services used to encrypt and decrypt data.

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 Visual Modeler to manage its properties. Connection pooling has been unified into one pool, and is tunable. Pagination has been updated, and no longer relies on pbtagination files being written to the file system.

Dispatch Authorization

This module manages access checking that enusres that each user sees only those parts of the application to which they have been granted access.

Dispatch Framework

This module manages the dispatch framework of the Visual Modeler classes that wrap the servlet request, response, context, and session classes together with the base controller classes used by the dispatch mechanism.

Email Service

This module provides the basic APIs to initiate sending email from the Visual Modeler.

Event Service

This module provides the classes used by the EventBus and Events.

Exception Service

This module provides the basic exception framework and classes used by the Visual Modeler.

Global Cache Service

This module provides the APIs to be used to access the cache.

Help

This module provides the ComergentHelpBroker class: this is a simple wrapper class to the ServletHelpBroker class of the JavaHelp 2.0 implementation.

Initialization Service

The Initialization module provides the Initialization service. This is a package that helps you initialize the Visual Modeler 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.objmgr.ObjManagerInitHelper.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.

Internationalization

This module provides basic support for the internationalization capabilites provided by the Visual Modeler.

Logging

This module provides access to the logging service used to record activity in the Visual Modeler. 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.ComergentRollingFileAppender
#log4j.appender.CMGT=com.comergent.logging.ComergentDailyRollingFileAppender

#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.

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 Translator - 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.

Memory Monitor

This module provides classes used to monitor and log memory consumption.

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.dispatchAuthorization package. This package contains factory classes, interfaces, and exceptions needed for the service. The implementation classes are in the com.comergent.dispatchAuthorization 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.dispatchAuthorization.EntitlementRepository;
import com.comergent.api.dispatchAuthorization.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" for details.

Out Of Band Response

This module is used to send output to output streams other than the standard JSP pages.

Preferences Service

The Preferences module is used to retrieve and set configuration properties used by the Visual Modeler. 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.

Tag Libraries

The tag libraries provided by the Visual Modeler are produced as a platform module.

Thread Management

This module provides a centralized facility for handling threads: their creation, obtaining their status, and re-use. It is provided by the backport-util-concurrent.jar library. 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());
```

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.

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.

Introducing Data Beans and Business Objects

This topic presents a brief tutorial that demonstrates how you can use the Visual Modeler to work easily with data beans and business objects.

Note: In Release 6.4 and later, the use of business objects is not supported. You should use data beans wherever possible.

What are Data Beans?

A data bean is a data source-independent representation of a real-world entity in the Visual Modeler. The Visual Modeler 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" 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.

Life Cycle 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:

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

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.

Note: 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 Visual Modeler applications.

Defining the Structure of a Data Object

Each data object must have a defined structure to enable the Visual Modeler 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 Visual Modeler 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 "Software Development Kit" topic for more details. See "Extending Data Objects" section for alternate ways to extend data objects.

Data Bean and Business Object Creation

The Visual Modeler’s ObjectManager and OMWrapper classes create data beans, and business logic classes and controllers process them. See "ObjectManager and OMWrapper Classes" topic 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 determining the number of results on each page (pagination). 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:

- 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.

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 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) sets the maximum number of results returned for non-paginated results
- int getMaxResults( ) gets the maximum number of results to return for non-paginated results
- void setMaxPaginatedResults(int maxResults) sets the maximum number of results returned for paginated results
- int getMaxPaginatedResults( ) gets the maximum number of results to return for paginated results
- void setNumPerPage(int numPerPage)
- int getNumPerPage( )

If an application wants to use the data services default limits, the appropriate property in DataContext must be set to **DsConstants.USE_DEFAULT**. The following are the default values:

- maxResults: 125
- maxPaginatedResults: 125
- numPerPage: 25

If the application does not specify a value for numPerPage, then the value specified in prefs.xml will be used. If a value is not set by the application nor the prefs.xml file, a value of -1 will be used, which means the request will not be paginated.

In addition, the following methods provide result set limits that are passed directly to the database as part of the SQL query. Since the Visual Modeler may discard results as part of its access policy checking (for example, does the user have the right to see this data?), these methods allow you to set a higher result set limit.

- public void setDBResultLimit(int limit)
- public int getDBResultLimit( )

You can also set the DataServices.General.LimitDBResults preference. If LimitDBResults is set to true, results are automatically limited to the number allowed by MaxResults (or by MaxPaginatedResults for paginated results). Access policies must be expressed as SQL to use this mechanism. For Oracle databases, do not set the LimitDBResults preference to true.

Our access policies are handled in one of two ways. Many are converted to SQL WHERE clauses that are applied to the query. This allows the database to handle the access policy. If the policy is too complex (for example, it relies on a hierarchy of partners), then the access policy can be applied only when processing the results from the database. Such policies cannot be converted to SQL.

With Oracle, there are some cases in which the SQL generation will require that column aliases be defined in the XML schema. This is necessary only when the query joins multiple tables that use the same column name. This is not an issue for SQL Server or DB2.

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:

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

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" 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.

Application, Entity, and Presentation Beans

There are several main sorts of data beans used in the Visual Modeler: 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" 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 Visual Modeler. 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.

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 life cycle. See "Generated Interfaces" for more information.

Use the access policy security mechanism to provide access control.

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" 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" for information on the use of the DataContext class in the *restore()* method.
- *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 Visual Modeler saves the data held in the data object’s DsElement tree to its external data source(s). Note that the Visual Modeler 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 Visual Modeler 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.

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 Visual Modeler 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 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 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:
```

3. 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)

4. 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.
5. 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 9.0, 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"/>
```

1. 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).

5. 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>
   ```

6. 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.

7. 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" 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:

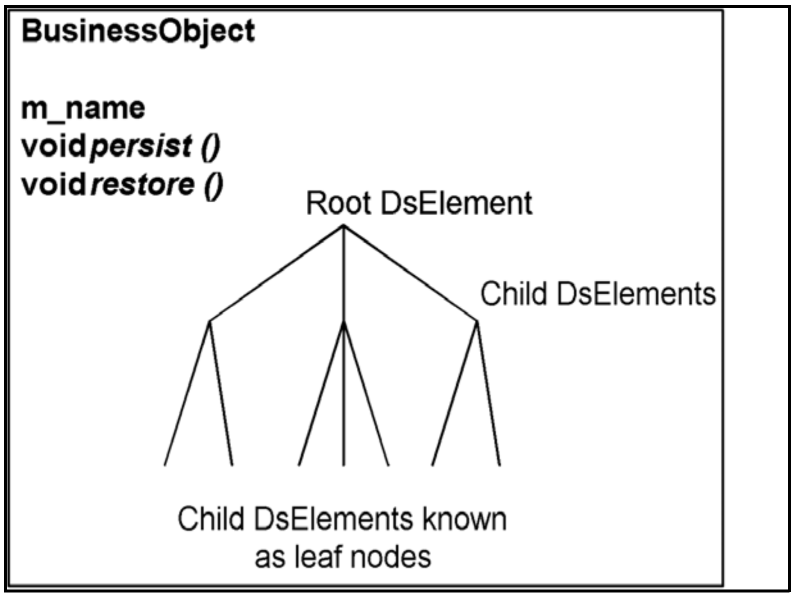
```
IAccEnquiry temp_IAccEnquiry = (IAccEnquiry)
OMWrapper.getObject("com.comergent.bean.simple.IAccEnquiry");
```

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"). Their content is retrieved from external systems using the XML schema, and the recipes and data sources defined in the XML schema. The following figure displays the business object:

.

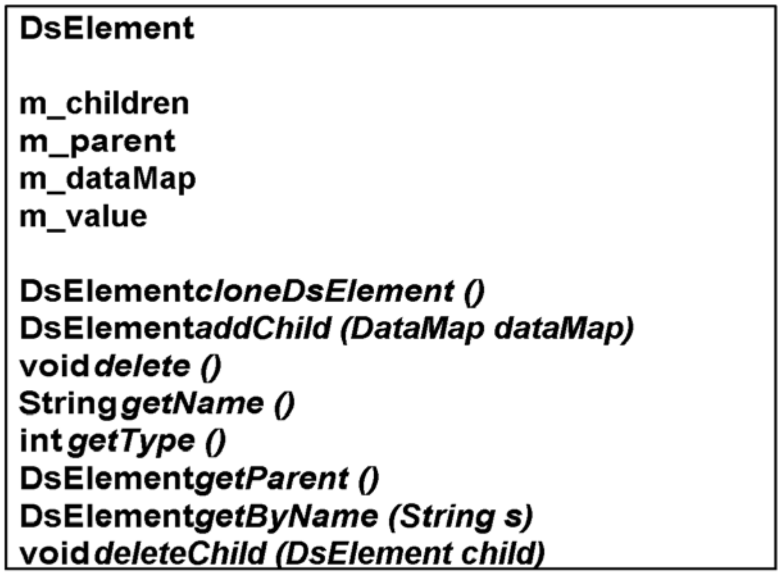


When the DataManager creates a data bean or business object, it uses the XML schema to determine the structure of its DsElement tree. The DsElement tree is the Java representation of the structure of the business object. The schema also determines the data types that may be inserted at leaf nodes and 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. The following figure displays the DsElements methods:



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)*

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: 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.

Logging

This topic describes the logging mechanism provided by the Visual Modeler. It enables application writers to log activity in the Visual Modeler. 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" for more information.

Overview

The log4j API provides a flexible and extensible logging framework to manage the logging behavior of the Visual Modeler. This section describes the use of the framework as you customize and extend the Visual Modeler.

Note that this framework replaces the previous framework used by the Visual Modeler: this used the Global class and its *log*Level*()* methods. These are now deprecated.

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.

log4j.debug System Property

By setting the log4j.debug system property to true, you can echo out the current log settings. For example, include the following in the servlet container startup script:

```
-Dlog4j.debug=true
```

```
On startup, you should see logging output like this:
```

```
log4j: Trying to find [log4j.xml] using context classloader
sun.misc.Launcher$AppClassLoader@136228.
```

```
log4j: Trying to find [log4j.xml] using
sun.misc.Launcher$AppClassLoader@136228 class loader.
```

```
log4j: Trying to find [log4j.xml] using ClassLoader.getSystemResource().
log4j: Trying to find [log4j.properties] using context classloader
sun.misc.Launcher$AppClassLoader@136228.
log4j: Using URL
[jar:file:/home/hle/ws/32-cmgt-modules/modules.cryptography-tool/target/cmgt-c
ryptography-tool-2.0.0-SNAPSHOT-app.jar!/log4j.properties] for automatic log4j
configuration.
log4j: Reading configuration from URL
jar:file:/home/hle/ws/32-cmgt-modules/modules.cryptography-tool/target/cmgt-cr
yptography-tool-2.0.0-SNAPSHOT-app.jar!/log4j.properties
log4j: Parsing for [root] with value=[WARN, A1].
log4j: Level token is [WARN].
log4j: Category root set to WARN
log4j: Parsing appender named "A1".
log4j: Parsing layout options for "A1".
log4j: Setting property [conversionPattern] to [%-4r [%t] %-5p %c %x - %m%n].
log4j: End of parsing for "A1".
log4j: Parsed "A1" options.
log4j: Finished configuring.
```

Auditing Changes to Data Objects

In many implementations, you may want to provide an audit trail that tracks changes made to data in the Visual Modeler. 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, 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 - PartnerKey: 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 CotermDayMonth -> 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 InheritedPartnerStatusCode -> 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 RemotePrices -> 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 Visual Modeler through the administration UI. 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 Visual Modeler.

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.comergent.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 Visual Modeler, then you could specify:

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

```
log4j.appender.com.comergent.bean.simple.UserContactBean.port=4321
```

Modularity and Generated Interfaces

The Visual Modeler has the following features which are designed to make implementations easier to customize and upgrade:

- Modules
- Generated Interfaces

These features are related in that the interfaces are organized by modules and that changes to interfaces may be contained to changes within individual modules.

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 Visual Modeler.
- 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 Visual Modeler.

Modules

The Visual Modeler 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 Visual Modeler such as an application or a component of the Visual Modeler 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. Additionally, the com.comergent.appservices.appServiceUtils.OFApiHelper is used to call the Sterling Selling and Fulfillment Foundation XAPIs.
  - 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 Visual Modeler. 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 Visual Modeler 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 Visual Modeler. See "Generated Interfaces" 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 Visual Modeler, use one of the following techniques to do this:

- Using the Object Manager
- Using Factory Classes

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 Visual Modeler.

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.

Implementing Logic Classes

This topic and the next two topics present a description of how to implement business logic classes (BLCs) at an implementation of the Visual Modeler. Before reading this topic, you must have a working understanding of the basic architecture of the Visual Modeler 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 Visual Modeler works as an application, you must understand its architecture.

An installation of Visual Modeler processes requests as they are received from users' browsers, and messages from other Visual Modelers and from external systems. You must configure the Visual Modeler to process each type of request and message.

The core of the Visual Modeler is the Sterling 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 Visual Modeler 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 Visual Modeler servers and external systems.

To customize the Visual Modeler 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 Visual Modeler 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 Visual Modeler. 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 Visual Modeler 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 Visual Modeler 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.

Business Objects

See "Introducing Data Beans and Business Objects" topic for more information.

XML Schema

You should manage data access using the the schema and Service classes.

Naming Service

To retrieve parameters at runtime, the Visual Modeler 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 Visual Modeler 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: By default, the type of NamingService created is a NamingServiceDatabase object because in **Comergent.xml** the NamingManager defaultType element is set to "database".

Software Development Kit

You can use the Visual Modeler Software Development Kit (SDK) to install and customize your implementation of the Visual Modeler. The HTML documentation provided with each version of the SDKprovides an overview of how the SDK works and how to use it to manage projects. This topic describes the basic structure of a customization project. 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 Visual Modeler. 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 appopriate sub-directory of the project.
- Create the file manually in the appopriate sub-directory of the project.

See "Project File and Directory Locations" for information about where files must be located.

Project File and Directory Locations

In this section, we assume that you created a project called *project*, and that you have a project directory called ***sdk_home*/projects/*project*/**. Ensure 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 Visual Modeler.
- JSP pages: these 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

In the ***project*/src/** directory, follow these guidelines to organize your customizations to the Visual Modeler:

- Use the com/comergent/api/ packages to add your extensions to the Visual Modeler 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

In the ***project*/WEB-INF/web/** directory, follow these guidelines to organize your customizations to the Visual Modeler:

- 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

In the ***project*/WEB-INF/schema/custom/** directory, follow these guidelines to organize your customizations to the Visual Modeler:

- To add new data objects:
  - Put the XML definition of the data object in ***project*/WEB-INF/schema/custom/**. For example, create the file ***project*/WEB-INF/schema/custom/CustComment.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="CustComment" Version="6.0"
         Description="CustComment 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="CustComment" Description="Customer Comment data
object"
         DataType="HEADER"/>
   <DataElement Name="CustCommentList" Description="Customer Comment list
data
         object" DataType="HEADER"/>
   <DataElement Name="CustCommentKey" Description="Customer 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="CustComment" BusinessObject="CustComment"
         Description="Default Approvals List Recipe" Version="6.0">
         <DataObjectList>
               <DataObject Name="CustComment" 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="CustCommentKey" KeyProcedureName="CUSTCOMMENTKEY"
         GeneratorType="PROCEDURE" />
</Schema>
```

Visual Modeler Localization

This topic describes localization issues to consider while you work on Visual Modeler applications.

Overview

The Visual Modeler has built-in support for:

- multiple currencies
- multiple languages
- number and date formats
- character sets

You can also 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.

When users log in to the Visual Modeler, 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 and the change takes take effect the next time they log in. User administrators can change a user's preferred locale just as they can change other aspects of a user's profile.

The system default locale is specified in the **Internationalization.xml** configuration file using the defaultSystemLocale element. You can specify a default locale for each language: see "Failover Behavior" for more information.

The Visual Modeler 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" for more details.

Supporting Locales

If you plan to implement the Visual Modeler 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 Visual Modeler, 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 Visual Modeler **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 Visual Modeler 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.

Note: 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 Visual Modeler, the default system locale is set to en_US. Consequently a full set of JSP pages is provided under ***debs_home*/SterlingWEB-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.channelCartDisplay.ChannelCartDisplayDataResources'>Bui
```

```
ld 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*****/Sterling/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/Sterling/WEB-INF/web/en/US/** to ***debs_home*****/Sterling/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*****/Sterling/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" 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.

      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 "&lt;" 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*****/Sterling/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.

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 Visual Modeler. See "Javascript" and "JSP Pages".
- 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.
4. <en visible="false">
   <defaultCountry ...>US</defaultCountry>
   </en>
5. 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.
6. 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 Visual Modeler’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 Visual Modeler 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 Visual Modeler.

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: 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 Visual Modeler uses email templates to generate messages that are sent to users, then bear in mind that these need to be translated.

Release 6.4 has introduced the ability to use JSP pages to generate email messages: 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 Visual Modeler: these are located in ***debs_home*/Sterling/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" 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 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*/Sterling//*la*/*CO*/js/**. For example, to support the de_DE locale, create a file called ***debs_home*/Sterling/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();
```

Style Sheets

The Visual Modeler 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 Visual Modeler, and so you should use the language of the default system locale for these files.

Resource Bundles and Formats

PropertyResourceBundles and Properties Files

The Visual Modeler makes extensive use of properties files to manage locale-specific data. These have replaced the use of ResourceBundle Java classes. See "Supporting Locales" 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 Visual Modeler is deprecated. You should use properties files as described in "Supporting Locales".

These are classes that manage locale-specific information. ResourceBundle classes used in the Visual Modeler 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>
```

Customize Controls

Controls are used to determine how the option classes and option items are displayed and behave in the user interface (UI). You can modify an existing control or add a new control.

Each control corresponds to a JSP page and the behavior of the option items. This correspondence is defined in the `controls.properties` configuration file under the `Comergent/WEB-INF/properties` folder located in the deployment directory.

Following is a sample entry defined in the `controls.properties` file:

```
RADIO.name=Radio Button

RADIO.jsp=controls/radio.jsp

RADIO.behavior=single
```

In this example, for the radio button control, the `radio.jsp` JSP page is used to render the option class in the UI. The `behavior` property determines how the Sterling Configurator™ will handle picks in this control. Based on how the `behavior` property is defined, the Sterling Configurator handles picks as follows:

- 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.

Modify a Control

You can customize an existing control by modifying the corresponding entry in the `controls.properties` file.

To modify an existing control:

1. Run the following target to retrieve the `controls.properties` file for customization:

   ```
   sdk customize WEB-INF/properties/controls.properties
   ```

   Running this target places the `controls.properties` file in your customization project.
2. Modify the entries in the `controls.properties` file, as required.
3. Run the following target to merge the customizations into the build:

   ```
   sdk merge
   ```
4. If you are deploying the Visual Modeler application as a WAR file, perform the following steps:
   a. Run the following target to re-create the WAR file:

      ```
      sdk distWar
      ```
   b. Deploy the `.war` file on your application server.

After these steps are completed, you must perform the required modifications in the Sterling Selling and Fulfillment Foundation. For more information, refer to the *Sterling Configurator: Application Guide*.

Add a Control

You can define a new control by adding the name of the control to the list of controls declared, and then defining the properties of the new control.

To add a new control:

1. Run the following target to retrieve the `controls.properties` file for customization:

   ```
   sdk customize WEB-INF/properties/controls.properties
   ```

   Running this target places the `controls.properties` file in your customization project.

2. Add the name of the new control to the comma-separated list of values for the `controls` attribute

   For example, to add a new ABC_CUSTOM control, the `controls` attribute may be defined as follows:

   ```
   controls=ABC_CUSTOM,RADIO,CHECKBOX,COMBOBOX,LISTBOX,MULTISELLISTBOX,ALLPICK
   ED,UEV,DISPLAY
   ```

3. Define the properties of the new control. For example, you may define the properties of the new ABC_CUSTOM control as follows:

   ```
   ABC_CUSTOM.name=Matrix Custom Control
   ABC_CUSTOM.jsp=controls/ABCCustom.jsp
   ABC_CUSTOM.behavior=single
   ```

4. Run the following target to merge the customizations into the build:

   ```
   sdk merge
   ```

5. If you are deploying the Visual Modeler application as a WAR file, perform the following steps:

   a. Run the following target to re-create the WAR file:

      ```
      sdk distWar
      ```

   b. Deploy the `.war` file on your application server.

After these steps are completed, you must perform the required modifications in the Sterling Selling and Fulfillment Foundation. For more information, refer to the *Sterling Configurator: Application Guide*.

Customize Function Handlers

Function handler classes are Java classes that are used to define custom functions that can be invoked by the Sterling Configurator rule engine. You can customize function handler classes.

The function handlers are defined in the `functionHandlers.properties` configuration file under the `Comergent/WEB-INF/properties` folder located in the deployment directory. This file includes a name for each function handler and the directory in which the function handler class is located.

Following is a sample fragment of the `functionHandlers.properties` file:

```
WEB-INF/classes/com/comergent/apps/configurator/functionHandlers=CheckLookupFu
nctionHandler,ChildSum,CountFunctionHandler,IsSelectedHandler,LengthFunctionHa
ndler,ListFunctionHandler,LookupFunctionHandler,MaxFunctionHandler,MinFunction
Handler,ParentFunctionHandler,PropValHandler,SumFunctionHandler,ValueFunctionH
andler,WebServiceLookupCheckLookupFunctionHandler=com.comergent.apps.configura
tor.function-Handlers.CheckLookupFunctionHandler
```

Add a Function Handler Class

You can add a new function handler class.

To add a new function handler class:

1. Run the following target to retrieve the `functionHandlers.properties` file for customization:

   ```
   sdk customize WEB-INF/properties/functionHandlers.properties
   ```

   Running this target places the `functionHandlers.properties` file in your customization project.

2. 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.

   Note: The new Java class must be provided in the classpath of the Visual Modeler application.

   The new Java class should 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. Therefore, 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. Therefore, you must only override this method if the function handler is private.

After these steps are completed, you must perform the required modifications in the Sterling Selling and Fulfillment Foundation. For more information, refer to the *Sterling Configurator: Application Guide*.

Exceptions

This topic describes the framework for exception handling in the Visual Modeler. You should follow this to ensure consistency across your implementation of the system, and to help other people working on the implementation.

ComergentException 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.

ICCExeption

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 Visual Modeler 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);
   }
```

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 Visual Modeler should not see exceptions: the appropriate subsystem must handle the exception gracefully by responding appropriately to the error condition.

The Visual Modeler 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.

Implementing Cron Jobs

This topic describes the creation of cron jobs that run as part of the Visual Modeler.

Overview

Certain tasks within an implementation of the Visual Modeler 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 Visual Modeler.

Cron jobs can be defined either as system cron jobs or as application cron jobs.

- A system cron job is run by the Visual Modeler and is not associated with any user. A system cron job calls Visual Modeler 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 Visual Modeler 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

Note: 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.

Parameters are passed in to the cron job using the same syntax as for HTTP request parameters. For example: Name1=Value1&Name2=Value2.

Filters

This topic describes how you can use filters. It covers:

- Filters Overview
- Available Filters

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 Visual Modeler. All the filters are part of the com.comergent.dcm.core.filters package. It covers:

- DosFilter
- WSDLFilter

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/s/dXML/5.0/OrderInterface.wsdl, and so on.

Managing and Displaying Constrained Fields

This topic 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 Visual Modeler. 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 Visual Modeler interface.

Are the possible values that the constrained data field may take maintained dynamically or are they read once at start-up?

3. 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 Visual Modeler has started, then you can use a constraint element.

   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?

4. 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.

Notices

Trademarks

Index