BEA WebLogic Integration™

Programming BPM Plug-Ins for WebLogic Integration

Release 2.1 Service Pack 1
Document Date: January 2002

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Programming BPM Plug-Ins for WebLogic Integration

Part Number	Date	Software Version
N/A	January 2002	2.1 Service Pack 1

Contents

About This Document

This document explains how to develop plug-in applications that extend the features of the business process management (BPM) functionality of WebLogic Integration.

This document is organized as follows:

- Chapter 1, “Introduction to BPM Plug-In Development,” provides an introduction to developing plug-in applications. Specifically, this chapter offers an overview of both the BPM Plug-in Manager and plug-in API, explains how the BPM learns about a deployed plug-in, provides a summary of the main tasks in the plug-in application development process, and describes the plug-in sample from which the code examples presented in this document are taken.
- Chapter 2, “Plug-In Development Fundamentals,” describes the fundamental tasks required for plug-in development, including how to import packages and interfaces, connect to and disconnect from the Plug-in Manager, access the Plug-in Manager version, and use the plug-in value objects.
- Chapter 3, “Defining the Plug-In Session EJB,” explains how to define the plug-in session EJB.
- Chapter 4, “Defining Plug-In Components,” explains how to define plug-in components.
- Chapter 5, “Using Plug-In Notifications,” explains how to use plug-in notifications.
- Chapter 6, “Processing Plug-In Events,” explains how to process plug-in events.
- Chapter 7, “Managing Plug-Ins,” explains how to view, load, and configure plug-ins.
- Chapter 8, “Defining Plug-In Online Help,” explains how to define plug-in online help.

- Chapter 9, “Deploying the Plug-In,” explains how to deploy a plug-in.
- Chapter 10, “BPM Plug-In Sample,” describes the plug-in sample provided with BPM in detail.
- Appendix A, “Plug-In Component Definition Roadmap,” summarizes the steps required to define each type of plug-in component.
- Appendix B, “Plug-In Value Object Summary,” describes the BPM plug-in value objects and their methods.
- Appendix C, “BPM Graphical User Interface Style Sheet,” provides information to help you design custom plug-ins based on Java Swing classes.

What You Need to Know

This document is intended for application developers who are interested in creating custom plug-in applications. It is assumed that the reader is familiar with the WebLogic Integration product, Java programming, and XML.

e-docs Web Site

BEA product documentation is available on the BEA corporate Web site. From the BEA Home page, click on Product Documentation or go directly to the Product Documentation page at the following URL:

`http://e-docs.bea.com`

How to Print the Document

You can print a copy of this document from a Web browser, one file at a time, by using the File—>Print option on your Web browser.

A PDF version of this document is available on the WebLogic Integration documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the WebLogic Integration documentation Home page, click the PDF files button and select the document you want to print.

If you do not have the Adobe Acrobat Reader, you can get it for free from the Adobe Web site at the following URL:

`http://www.adobe.com/`

Related Information

The following WebLogic Integration documents contain information that may be helpful to programmers who are using and interfacing with the WebLogic Integration BPM client applications, the Studio and Worklist. These applications have been built using the BPM component of the WebLogic Integration API.

- *Programming BPM Client Applications*
- *Using the WebLogic Integration Studio*
- *Using the WebLogic Integration Worklist*
- *Learning to Use BPM with WebLogic Integration*
- BEA WebLogic Integration API Javadoc

For general information about Java applications, go to the Sun Microsystems, Inc. Java Web site at the following URL:

`http://java.sun.com/`

For general information about XML and XML parsers, go to the O'Reilly & Associates, Inc. XML.com Web site at the following URL:

`http://www.xml.com/`

Contact Us!

Your feedback on the WebLogic Integration documentation is important to us. Send us e-mail at **docsupport@bea.com** if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the WebLogic Integration documentation.

In your e-mail message, please indicate which release of the WebLogic Integration documentation you are using.

If you have any questions about this version of WebLogic Integration, or if you have problems installing and running WebLogic Integration, contact BEA Customer Support through BEA WebSupport at **www.bea.com**. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Item
boldface text	Indicates terms defined in the glossary.
Ctrl+Tab	Indicates that you must press two or more keys simultaneously.
italics	Indicates emphasis or book titles.

`monospace text`	Indicates code samples, commands and their options, data structures and their members, data types, directories, and file names and their extensions. Monospace text also indicates text that you must enter from the keyboard.
	Examples:
	`#include <iostream.h> void main ( ) the pointer psz`
	`chmod u+w *`
	`\tux\data\ap`
	`.doc`
	`tux.doc`
	`BITMAP`
	`float`

`monospace boldface text`	Identifies significant words in code. *Example*: `void` **`commit`** `( )`
`monospace italic text`	Identifies variables in code. *Example*: `String` *`expr`*
UPPERCASE TEXT	Indicates device names, environment variables, and logical operators. *Examples*: LPT1 SIGNON OR

Convention	Item
`{ }`	Indicates a set of choices in a syntax line. The braces themselves should never be typed.
`[ ]`	Indicates optional items in a syntax line. The brackets themselves should never be typed. *Example*: `buildobjclient [-v] [-o name ] [-f file-list]... [-l file-list]...`
`|`	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.
`...`	Indicates one of the following in a command line: ■ That an argument can be repeated several times in a command line ■ That the statement omits additional optional arguments ■ That you can enter additional parameters, values, or other information The ellipsis itself should never be typed. *Example*: `buildobjclient [-v] [-o name ] [-f file-list]... [-l file-list]...`
. . .	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

CHAPTER

1 Introduction to BPM Plug-In Development

This section provides an introduction to business process management (BPM) plug-ins and their development. It includes the following topics:

- What Is a Plug-In?
- How BPM Supports Plug-In Development
- How BPM Discovers a Deployed Plug-In
- BPM Plug-In Development Tasks
- BPM Plug-In Sample

What Is a Plug-In?

A *plug-in* consists of a set of run-time loadable Java classes that extends the business process management (BPM) features and functionality of WebLogic Integration.

Using plug-ins, you can modify the design or run-time behavior of the following BPM workflow components:

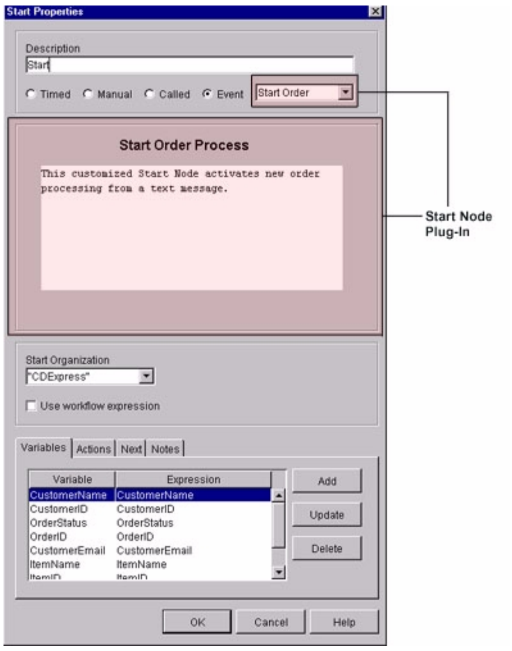
- Start, Event, and Done nodes
- Task actions
- Properties of workflow templates and template definitions

- Functions
- Message types
- Variable types

For example, you may want to trigger the execution of a business process by sending an e-mail message or other nonXML event, rather than by using any of the standard Start node trigger methods available to you with the WebLogic Integration Studio. You can accomplish this by designing a plug-in that extends the behavior of the Start node to include support for this new nonXML trigger.

The following figure provides an example of a plug-in that modifies the design and run-time behavior of a Start node. It illustrates the plug-in-defined areas of the Start Properties dialog box.

Figure 1-1 Plug-In Example: Start Node



Description: Start

Timed ○ Manual ○ Called ○ Event ● — Start Order

Start Order Process

This customized Start Node activates new order processing from a text message.

Start Organization: "CDExpress"

☐ Use workflow expression

Variable	Expression
CustomerName	CustomerName
CustomerID	CustomerID
OrderStatus	OrderStatus
OrderID	OrderID
CustomerEmail	CustomerEmail
ItemName	ItemName
ItemID	ItemID

In this example:

- The Start Properties dialog box lists the Start Order event as a potential trigger on the Event pull-down menu, and displays the event information defined by the plug-in in the center of the dialog box when the event is selected. This behavior shows how the *design* of the Start node has been customized.
- At run-time, the Start Order event may trigger the workflow to start. This functionality shows how the *run-time behavior* of the Start node has been customized.

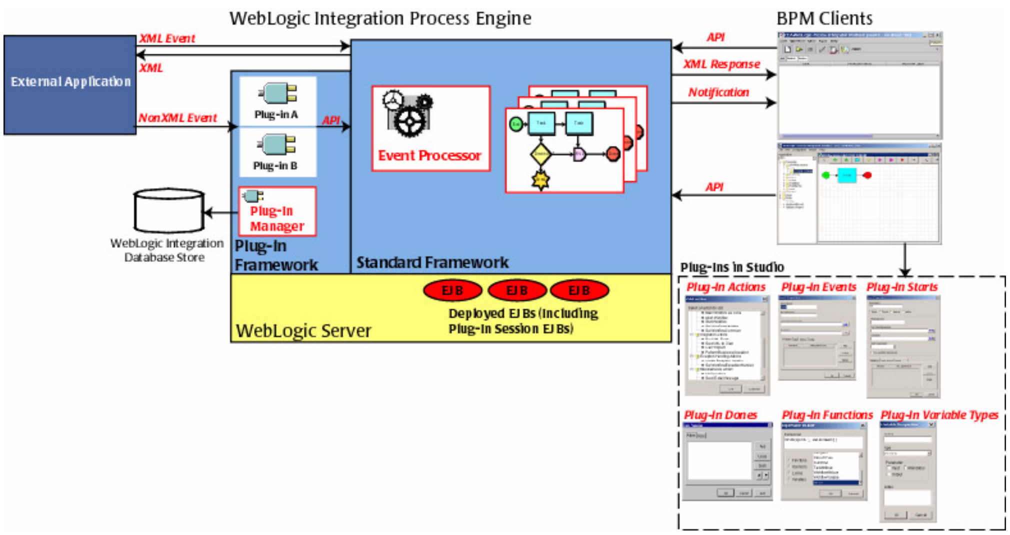
Note: For more information about the Start node plug-in illustrated in the previous figure, see “BPM Plug-In Sample” on page 10-1.

How BPM Supports Plug-In Development

In addition to the standard framework for designing, executing, and monitoring business processes, WebLogic Integration supports a *plug-in framework* for BPM functionality, enabling you to create plug-ins that customize the existing software, and achieve powerful and seamless integration with other products and technologies.

The following figure illustrates the BPM plug-in framework within the overall BPM architecture.

Figure 1-2 BPM Plug-In Framework



Note the following in this figure:

- BEA WebLogic Server manages the deployment of the BPM and plug-in EJBs.

  To make the plug-in available to BPM users, you simply deploy the plug-in as a session EJB on WebLogic Server. *No additional installation is required on the WebLogic Integration process engine or BPM client.*

  The plug-in must be deployed as part of the WLI application by editing the `config.xml` file. For more information about deploying a plug-in, see "Deploying the Plug-In" on page 9-1.

- The BPM Plug-in Manager stores plug-in configuration information in the WebLogic Integration database.

- An external application interacts with the process engine via an *XML event* or *nonXML event*. NonXML events are supported via the plug-in framework.

- The Event Processor manages both XML and nonXML events. For more information about plug-in event handling, see "Processing Plug-In Events" on page 6-1.

- The main components of the plug-in framework include:
  - Plug-in Manager, which supports the management of plug-ins
  - Plug-in API, which supports the design and development of plug-ins

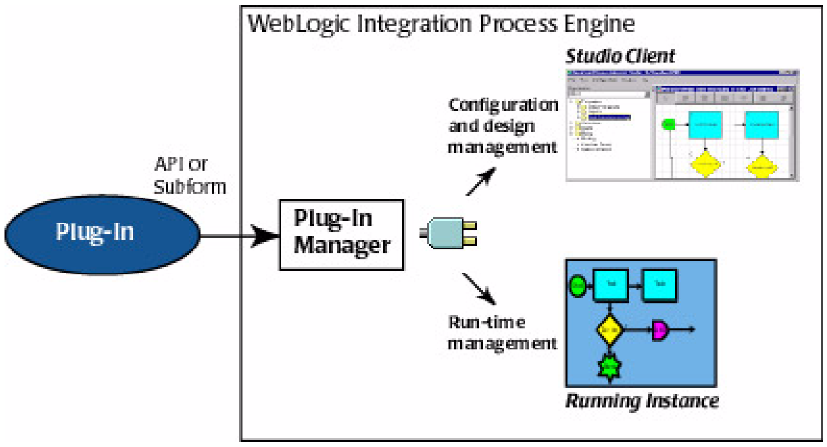
The following sections describe these two components.

For additional information about the role of the process engine, see "WebLogic Integration Process Engine" in "Business Process Management API Development" in *Programming BPM Client Applications*.

Plug-In Manager

The Plug-in Manager is a dedicated part of the WebLogic Integration process engine that supports the configuration, design, and run-time management of plug-ins, as shown in the following figure.

Figure 1-3 Plug-In Manager



The Plug-in Manager oversees the loaded plug-ins and their interactions with the WebLogic Integration process engine and BPM clients, and routes all plug-in related requests.

For example, a subset of the *configuration and design* responsibilities of the Plug-in Manager includes:

- Supporting the import and export of workflows containing plug-in-specific content.
- Managing the simultaneous loading of multiple plug-ins.
- Enabling the BPM expression evaluator to interpret incoming plug-in data in any format, using common expression grammar.
- Enforcing the dependencies between templates and template definitions, and plug-ins, and handling cases in which the prerequisite plug-ins are not available.
- Enforcing strict type checking during the definition of a business operation that uses plug-in-defined variable types.
- Providing a generic message-handling facility that enables workflows to respond to an unlimited variety of message-handling data formats and sources.

For example, a subset of the *run-time* responsibilities of the Plug-in Manager includes:

- Enabling the plug-in to store and retrieve plug-in-specific workflow instance data.
- Preventing the instantiation of a workflow when a required plug-in is not loaded.
- Supports custom license checking during initialization.

When designing plug-ins, you access the management features of the Plug-in Manager using the following session EJBs:

- `com.bea.wlpi.server.plugin.PluginManager`
- `com.bea.wlpi.server.plugin.PluginManagerCfg`

These EJBs are part of the plug-in API described in the next section.

Plug-In API

The BPM API provides support for configuration, design, and run-time interactions between the process engine and a deployed plug-in.

The *plug-in API* consists of two session EJBs, a set of run-time management classes, and one package, as described in the following table.

Table 1-1 BPM Plug-In API Components

Component	Description
`com.bea.wlpi.server.plugin.PluginManager`	Stateless session EJB providing run-time management of plug-ins during workflow execution: ■ Provides meta-data about the deployed plug-ins ■ Enables access to plug-in design and run-time components ■ Handles event notifications to and from plug-ins
`com.bea.wlpi.server.plugin.PluginManagerCfg`	Stateless session EJB providing configuration and design management of plug-ins: ■ Maintains plug-in configuration information ■ Manages the plug-in framework cluster-wide state transitions ■ Maintains a list of plug-ins that are registered for the system event notifications
`com.bea.wlpi.server.plugin.*`	Classes providing run-time management of plug-ins during workflow execution.
`com.bea.wlpi.common.plugin`	Package providing client and process engine functions, including value object classes. All members of this package are serializable to allow for exchanges between the client and process engine.

Note: For a complete description of the BPM API, see *Programming BPM Client Applications* or the *BEA WebLogic Integration Javadoc*.

How BPM Discovers a Deployed Plug-In

To make the plug-in available to BPM users, you simply package and deploy it as a session EJB on WebLogic Server.

If no additional installation is required on the process engine or BPM client, how does BPM discover a deployed plug-in and its implementation details? The plug-in is responsible for providing the functionality that enables BPM to:

- Detect the plug-in
- Manage lifecycle tasks and cache information about the plug-in
- Access the plug-in implementation to read, display, and save the plug-in within the design client
- Execute the plug-in at run time

Detecting the Plug-In

At startup, the process engine detects the plug-in via JNDI based on the plug-in's `com.bea.wlpi.server.plugin.PluginHome` home interface. All plug-in beans must use the `PluginHome` as their home interface.

For more information about the `PluginHome` interface, see "Plug-In Home Interface" on page 3-5.

Managing Lifecycle Tasks

Once the plug-in is detected, the Plug-in Manager performs the following tasks:

- Initializes the plug-ins using the `init()` method.

  The plug-in can be initialized only once during its lifecycle.

- Gets information about the plug-in, including configuration and dependency information, using the following methods:
  - `getPluginInfo()` - gets basic information about the plug-in
  - `getPluginConfiguration()` - gets default configuration information for the plug-in

    You can subsequently set the configuration information using the `setConfiguration()` method, as described in "Managing Plug-Ins" on page 7-1.
  - `getDependencies()` - gets dependencies for the plug-in

    The Plug-in Manager ensures that all dependencies are loaded before loading the plug-in.
- Loads or unloads plug-ins, using the `load()` or `unload()` methods, respectively, based on their configuration.

  The plug-in is available only after it is loaded. When the plug-in is loaded, the Plug-in Manager calls the `getPluginCapabilitiesInfo()` method to get detailed plug-in information, and the plug-in can register for notification messages, as described in "Using Plug-In Notifications" on page 5-1. At this time, all plug-in classes become visible to the BPM client.
- At shutdown, unloads and deinitializes all deployed plug-ins, using the `unload()` and `exit()` methods, respectively.

  The plug-in can be deinitialized only once during its lifecycle.

Each of the lifecycle methods listed in the previous list are remote interface methods that must be implemented by the plug-in bean. For more information, see "Plug-In Remote Interface" on page 3-6.

Accessing the Plug-In Implementation

The plug-in must implement the following classes for each component to define its functionality to the design client:

- Plug-in panel class to define the plug-in GUI component that is displayed in the design client. For more information, see "Displaying the Plug-In GUI Component" on page 4-23.

- Plug-in data interface to read and save plug-in data. For more information, see "Reading and Saving Plug-In Data" on page 4-9.

Because no additional installation is required on the BPM design client (for example, the Studio), the design client has no knowledge of the concrete classes that the plug-in defines. It is the responsibility of the plug-in to implement a public constructor that requires no arguments to support remote class loading on the design client.

Remote class loading on the design client proceeds as follows:

- The design client uses the value objects, obtained and cached by the Plug-in Manager at startup, to retrieve the plug-in Java class names based on the plug-in component name and ID.
- Subsequently, the class is loaded from the server by a custom class loader and instantiated by the design client using its no-argument constructor.

For example, in the figure "Plug-In Example: Start Node" on page 1-2, when a user selects the Start Order event as the Start node trigger, the Plug-in Manager loads the plug-in panel class, `StartNodePanel`, and it is instantiated by the Studio client using the no-argument constructor. The Studio client subsequently displays the plug-in GUI component in the Start Properties dialog box.

Executing the Plug-In at Run Time (Context Passing)

To define plug-in execution characteristics, you must implement a run-time interface for the plug-in component. At run time, the plug-in communicates with the process engine and client using a process called *context passing*: the Plug-in Manager obtains an instance of the plug-in component run-time interface and passes the context to it.

Each *context* interface provides restricted access to the Plug-in Manager, enabling the plug-in to execute and manage its own application logic, and introduce the plug-in instance data into the BPM run-time environment.

For more information about implementing the run-time interface, see "Executing the Plug-In" on page 4-59. For more information about implementing context interfaces, see "Using Plug-In Run-Time Contexts" on page 4-94.

BPM Plug-In Development Tasks

To develop a BPM plug-in, you need to first create a session EJB that defines the required plug-in classes and interfaces, and then package and deploy the session EJB on WebLogic Server.

The following steps outline the BPM plug-in development tasks in more detail.

Note: In addition to accomplishing the steps outlined below, you should also review the plug-in development fundamentals described in "Plug-In Development Fundamentals" on page 2-1.

Step 1: Identify design-time and run-time customization requirements.

Keep in mind that you can modify the design and run-time behavior of the following workflow components using plug-ins:

- Start, Event, and Done nodes
- Properties of the workflow templates and template definitions
- Task actions
- Functions
- Message types
- Variable types

The plug-in sample provides a set of plug-in classes that represent common plug-in scenarios. For more information, see "BPM Plug-In Sample" on page 1-13.

Step 2: Define the plug-in session EJB:

1. Implement the `javax.ejb.SessionBean` interface methods, including:

   ```
   ejbActivate()
   ejbPassivate()
   ejbRemove()
   setSessionContext(SessionContext ctx)
   ```

2. Implement the `com.bea.wlpi.server.plugin.PluginHome` home interface `ejbCreate()` method.

The home interface has been defined for you by the BPM plug-in framework via the `com.bea.wlpi.server.plugin.PluginHome` interface. The BPM home interface is described in detail in "Plug-In Home Interface" on page 3-5.

Note: At this time, you can set up a custom plug-in icon for the Studio interface view when implementing the `ejbCreate()` method, as described in the table "Home Interface Method" on page 3-5.

3. Implement the `com.bea.wlpi.server.plugin.Plugin` remote interface methods.

   The remote interface has been defined for you by the BPM plug-in framework via the `com.bea.wlpi.server.plugin.Plugin` interface. The BPM remote interface is described in detail in "Plug-In Remote Interface" on page 3-6.

   Note: At this time, you must define the plug-in component value objects, and customize the action tree (if defining a plug-in action) when implementing the `getPluginCapabilitiesInfo()` method, as described in the table "Remote Interface Plug-In Information Methods" on page 3-10.

This step is described in detail in "Defining the Plug-In Session EJB" on page 3-1.

Step 3: Define the plug-in component:

1. Implement the plug-in data interface to define the methods used for reading and saving the plug-in data.
2. Define the plug-in panel class to display the plug-in GUI component within the design client.
3. Define the plug-in run-time component class to define the run-time execution characteristics.

This step is described in detail in "Defining Plug-In Components" on page 4-1.

Step 4: Set up notification management.

This step is described in detail in "Using Plug-In Notifications" on page 5-1.

Step 5: Implement and register an event handler to process incoming plug-in events.

This step is described in detail in "Processing Plug-In Events" on page 6-1.

Step 6: Manage the plug-in, customizing the plug-in configuration requirements, if desired.

This step is described in detail in "Managing Plug-Ins" on page 7-1.

Step 7: Develop context-sensitive online help for the plug-in.

This step is described in detail in “Defining Plug-In Online Help” on page 8-1.

Step 8: Package all plug-in Java classes in EJB JAR and WAR files, and deploy the plug-in.

This step is described in detail in “Deploying the Plug-In” on page 9-1.

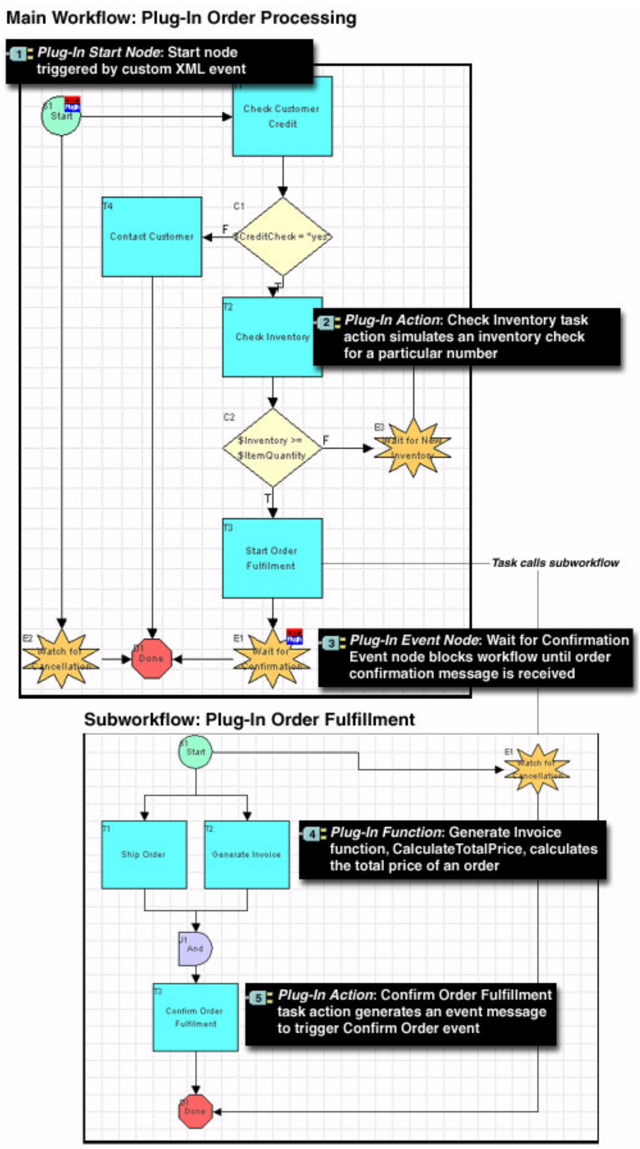
BPM Plug-In Sample

The BPM plug-in sample provides a set of plug-in classes that represent common plug-in scenarios, and is provided with the software. The sample includes two workflow templates, Plug-in Order Processing and Plug-in Order Fulfillment, which are stored in the `WLI_HOME/samples/bpm_api/plugin/sample_plug_in.jar` file.

Note: The plug-in sample is loosely based on a generic Web-based sales order scenario that is described in detail in “Introduction to WebLogic Integration and the Example Workflows” in *Learning to Use BPM with WebLogic Integration*.

The following figure shows the plug-in sample workflow templates and identifies the plug-ins that have been added.

Figure 1-4 Plug-In Sample Workflow Templates


Main Workflow: Plug-In Order Processing
1 Plug-In Start Node: Start node triggered by custom XML event
S1 Start
T1 Check Customer Credit
C1 $CreditCheck = "yes"
F
T
T4 Contact Customer
T2 Check Inventory
2 Plug-In Action: Check Inventory task action simulates an inventory check for a particular number
C2 $Inventory >= $ItemQuantity
F
T
E3 Wait for New Inventory
T3 Start Order Fulfillment
Task calls subworkflow
E2 Watch for Cancellation
D1 Done
E1 Wait for Confirmation
3 Plug-In Event Node: Wait for Confirmation Event node blocks workflow until order confirmation message is received
Subworkflow: Plug-In Order Fulfillment
S1 Start
E1 Watch for Cancellation
T1 Ship Order
T2 Generate Invoice
4 Plug-In Function: Generate Invoice function, CalculateTotalPrice, calculates the total price of an order
J1 And
T3 Confirm Order Fulfillment
5 Plug-In Action: Confirm Order Fulfillment task action generates an event message to trigger Confirm Order event
D1 Done

The previous example illustrates the following:

- The Plug-in Order Processing workflow template includes three plug-in components from the sample plug-in: a Start node that is triggered by a plug-in-defined event, a task action that simulates an inventory check, and an Event node that blocks until an order confirmation message is received.
- The Plug-in Order Fulfillment workflow template includes two plug-in components from the sample plug-in: a function that calculates the total price of an order and a task action that generates an event message to trigger the Confirm Order event.

Note: The Order Processing Trigger workflow template provided as part of the generic example (described in *Learning to Use BPM with WebLogic Integration*) is not used by the plug-in. Instead, the Plug-in Order Processing workflow template is triggered via a plug-in-defined XML event.

Excerpts from the plug-in sample are included throughout this document. For complete information about the BPM plug-in sample and its directory structure, and instructions for importing and running the sample, see "BPM Plug-In Sample" on page 10-1. For more information on the generic Web-based sales order scenario, see *Learning to Use BPM with WebLogic Integration*.

Note: In the previous figure, the following custom icon is shown in the upper-right corner of the Start and Event nodes to indicate that they contain customized plug-in properties.

Sample
Plugin

Icons such as this are displayed when the Studio interface view is enabled. For information about how to enable the interface view, see "Using the Studio Interface" in *Using the WebLogic Integration Studio*.

To specify a custom plug-in icon when creating a remote plug-in object interface, see the `create()` method description in "Plug-In Home Interface" on page 3-5.

CHAPTER

2 Plug-In Development Fundamentals

This section describes the fundamental tasks required for plug-in development. It includes the following topics:

- Importing Packages and Interfaces
- Connecting to the Plug-In Manager
- Getting the Plug-In Framework Version
- Using Plug-In Value Objects
- Disconnecting from the Plug-In Manager

You may also want to review the following chapters in *Programming BPM Client Applications*:

- Accessing Process Engine Information - explains how to obtain information about the WebLogic Integration process engine
- Establishing JMS Connections - explains how to establish a connection to WebLogic Java Messaging Service (JMS)
- Understanding the BPM Transaction Model - explains how transactions are handled in BPM applications

Importing Packages and Interfaces

Import the BPM packages and interfaces, and general Java packages, as desired.

Review "Importing Packages and Interfaces" in *Programming BPM Client Applications* for a description of the packages and interfaces that you may import, including the following which are used for plug-in management:

- `com.bea.wlpi.server.plugin.PluginManager` interface
- `com.bea.wlpi.server.plugin.PluginManagerCfg` interface
- `com.bea.wlpi.common.plugin` package

Connecting to the Plug-In Manager

To connect to the BPM Plug-in Manager, use the `PluginManager` and/or `PluginManagerCfg` session EJBs.

As with any EJB, to access the `PluginManager` and/or `PluginManagerCfg` EJBs, you must use the home and remote interfaces. To do so, perform the following steps:

1. Look up a session EJB home interface in JNDI.
2. Create a remote session object (`EJBObject`) using the home interface.

Review "Connecting to the Process Engine" in *Programming BPM Client Applications* for a description of how to access API session EJBs (in this case the `PluginManager` and `PluginManagerCfg` EJBs).

The following code listing is an excerpt from the plug-in sample that shows how to connect to the Plug-in Manager through the following two-step procedure:

1. Create an initial context and use the JNDI context `lookup()` method to access the session EJB home interface.
2. Create a remote session object using the home interface.

This excerpt is taken from the `SamplePluginBean.java` file in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory. Notable lines of code are shown in **bold**.

Note: For more information about the initial context, see the `javax.naming.InitialContext()` Javadoc.

Listing 2-1 Connecting to the Plug-In Manager

```
private final static String PLUGIN_MANAGER_CFG_HOME =
    "java:comp/env/ejb/PluginManagerCfg";
  .
  .
  .
private PluginManagerCfg getPluginManagerCfg() throws PluginException {

    PluginManagerCfg pm = null;
    InitialContext ic = null;

    try {
        ic = new InitialContext();

        PluginManagerCfgHome pmHome =
            (PluginManagerCfgHome)ic.lookup(PLUGIN_MANAGER_CFG_HOME);

        pm = pmHome.create();
    } catch (Exception e) {
        e.printStackTrace();

        throw new PluginException(SamplePluginConstants.PLUGIN_NAME,
                                  "Unable to get PluginManagerCfg");
    } finally {
        try {
            ic.close();
        } catch (Exception e) {
        }
    }

    return pm;
}
```

For more information about the plug-in sample, see "BPM Plug-In Sample" on page 10-1.

Getting the Plug-In Framework Version

To get the plug-in framework version, use one of the following methods:

```
public com.bea.wlpi.common.VersionInfo
com.bea.wlpi.server.plugin.PluginManager.getFrameworkVersion(
) throws java.rmi.RemoteException
```

```
public com.bea.wlpi.common.VersionInfo
com.bea.wlpi.common.plugin.PluginInfo.getPluginFrameworkVersion(
) throws java.rmi.RemoteException
```

These methods return the Plug-in Manager version as a `com.bea.wlpi.common.VersionInfo` object. To obtain information about the version, use the `VersionInfo` object methods described in "VersionInfo Object" in "Value Object Summary" in *Programming BPM Client Applications*.

For example, the following code gets the Plug-in Manager version using the `PluginManager` object method and saves it to the `version` object. In the examples, `pm` represents the `EJBObject` reference to the `PluginManager` EJB:

```
VersionInfo version = pm.getFrameworkVersion();
```

For more information about the `getFrameworkVersion()` method, see the `com.bea.wlpi.server.plugin.PluginManager` Javadoc. For more information about the `getPluginFrameworkVersion()` method, see the `com.bea.wlpi.common.plugin.PluginInfo` Javadoc.

Using Plug-In Value Objects

The `com.bea.wlpi.common.plugin` package provides *Info* classes, or *value objects*, for obtaining plug-in object data at both definition time and run time. Using plug-in value objects, the process engine and BPM client applications request plug-in object data for a specified locale, enabling the plug-in to localize display strings and other resources appropriately.

Value objects play an important role in remote class loading on the design client. Specifically, the BPM design client uses value objects:

1. To retrieve, based on the plug-in component name and ID, the names of the Java classes included in the plug-in.
2. To initiate remote class loading on the design client, based on the returned values, to obtain an instance of the defined classes.

For more information about remote class loading, see “Accessing the Plug-In Implementation” on page 1-9.

The following table lists the value objects that can be used to create and use plug-in object data.

Table 2-1 Plug-In Value Objects

Use this value object . . .	To access plug-in . . .
`com.bea.wlpi.common.plugin.ActionCategoryInfo`	Action or action category information.
`com.bea.wlpi.common.plugin.ActionInfo`	Action information.
`com.bea.wlpi.common.plugin.CategoryInfo`	Action category information.
`com.bea.wlpi.common.plugin.ConfigurationData`	Configuration data.
`com.bea.wlpi.common.plugin.ConfigurationInfo`	Configuration information.
`com.bea.wlpi.common.plugin.DoneInfo`	Done node information.
`com.bea.wlpi.common.plugin.EventHandlerInfo`	Event handler information.
`com.bea.wlpi.common.plugin.EventInfo`	Event node information.
`com.bea.wlpi.common.plugin.FieldInfo`	Message type information.
`com.bea.wlpi.common.plugin.FunctionInfo`	Evaluator function information.
`com.bea.wlpi.common.plugin.HelpSetInfo`	Online help information.
`com.bea.wlpi.common.plugin.InfoObject`	Abstract base class for all plug-in value objects.
`com.bea.wlpi.common.plugin.PluginCapabilitiesInfo`	Capabilities information.
`com.bea.wlpi.common.plugin.PluginDependency`	Dependency information.
`com.bea.wlpi.common.plugin.PluginInfo`	Basic plug-in information.

Table 2-1 Plug-In Value Objects (Continued)

Use this value object . . .	To access plug-in . . .
`com.bea.wlpi.common.plugin.StartInfo`	Start node information.
`com.bea.wlpi.common.plugin.TemplateDefinitionPropertiesInfo`	Template definition properties information.
`com.bea.wlpi.common.plugin.TemplateNodeInfo`	Template node (Done and Start) information.
`com.bea.wlpi.common.plugin.TemplatePropertiesInfo`	Template properties information.
`com.bea.wlpi.common.plugin.VariableTypeInfo`	Variable type information.

The following sections explain how to create and use value objects.

For more information about the value object constructors and the associated get and set methods, see "Plug-In Value Object Summary" on page B-1. For a list of common characteristics shared by value objects, and an explanation of how to sort them, see "Using Value Objects" in *Programming BPM Client Applications*.

Defining Plug-In Value Objects

To define the plug-in value object, use the associated constructor. For each of the plug-in value objects described in the table "Plug-In Value Objects" on page 2-5, one or more constructors for creating object data are provided. The constructors for creating value objects are described in "Plug-In Value Object Summary" on page B-1.

You must pass the plug-in value objects for each of the plug-in components when defining the `com.bea.wlpi.common.plugin.PluginCapabilitiesInfo` object. For details, see the description of the `getPluginCapabilitiesInfo()` method, in the table "Remote Interface Plug-In Information Methods" on page 3-10.

When creating a value object, you must specify:

- Plug-in ID, which must be unique for the plug-in and object type.
- Description of the plug-in features.

- Globally unique internal identifier, formed by appending one or more dot-separated strings to the vendor reverse-DNS name (for example, `com.somedomain.someproduct.myplugin`).
- Array containing the Java class names associated with the plug-in. The design client can retrieve this information using the plug-in name and ID, as described in "Getting and Setting Object Data" on page 2-8. Subsequently, the design client can initiate remote class loading to access the implementation classes. For more information about remote class loading, see "Accessing the Plug-In Implementation" on page 1-9.
- The icon used by the WebLogic Integration Studio to represent the plug-in when the interface view is enabled. Custom icons can be defined when you implement the home interface `ejbCreate()` method, as described in "Plug-In Home Interface" on page 3-5.

For example, the following code creates a `StartInfo` object and assigns the resulting object to `si`:

```
si = new StartInfo(SamplePluginConstants.PLUGIN_NAME, 5,
          bundle.getString("startOrderName"),
          bundle.getString("startOrderDesc"), ICON_BYTE_ARRAY,
          SamplePluginConstants.START_CLASSES, orderFieldInfo);
```

The `START_CLASSES` array defines the plug-in data, plug-in panel, and run-time component class names for the plug-in Start node. The `START_CLASSES` array is defined within the `SamplePluginConstants.java` class file as follows:

```
final static String[] START_CLASSES = {
       START_DATA,
       START_PANEL,
       START_NODE };
```

In this example, the array variable values are defined in the `SamplePluginConstants.java` file, as follows:

```
final static String START_NODE =
       "com.bea.wlpi.tour.po.plugin.StartNode";
final static String START_DATA =
       "com.bea.wlpi.tour.po.plugin.StartNodeData";
final static String START_PANEL =
       "com.bea.wlpi.tour.po.plugin.StartNodePanel";
```

The `ICON_BYTE_ARRAY` variable specifies a byte array representation of the graphical image (icon) that is used by the Studio to represent the plug-in when the Studio interface view is enabled.

For more information about value object constructors and associated get and set methods, see “Plug-In Value Object Summary” on page B-1.

Getting and Setting Object Data

Each plug-in value object described in the table “Plug-In Value Objects” on page 2-5 provides various methods for getting and setting object data. These methods are described in “Plug-In Value Object Summary” on page B-1.

For example, the following code gets the `PluginTriggerPanel` implementation class for a plug-in Start node and saves it to the `startpanel` string:

```
java.lang.String startpanel = si.getClassName(KEY_PANEL);
```

In this example, `si` represents a reference to the `com.bea.wlpi.common.plugin.StartInfo` value object for the plug-in Start node.

For more information about plug-in value object methods, see “Plug-In Value Object Summary” on page B-1.

Disconnecting from the Plug-In Manager

To disconnect from the BPM Plug-in Manager, perform the following steps:

1. Remove session EJB references to make the system space available for use by other EJBs.

   For example, the following excerpt from the plug-in sample shows how to remove the `PluginManagerCfg` EJB reference. In this example, `pm` represents the `EJBObject` reference to the `PluginMangerCfg` EJB:

   ```
   try {
      if (pm != null)
      pm.remove();
   } catch (Exception e) {}
   ```

   For more information, see “Removing Session EJB References” in “Disconnecting from the Process Engine” in *Programming BPM Client Applications*.

2. Remove additional resources, including JMS connections (if applicable), and closing the context.

   For example, the following code closes the JNDI context resources:

   ```
   try {
       ic.close();
   { catch(Exception exp) {}
   ```

   For more information, see “Releasing Other Resources” in “Disconnecting from the Process Engine” in *Programming BPM Client Applications*

CHAPTER

3 Defining the Plug-In Session EJB

This section explains how to define the plug-in session EJB. It includes the following topics:

- Overview
- Session EJB Interface
- Plug-In Home Interface
- Plug-In Remote Interface

Overview

To define the plug-in session EJB, you must implement the three predefined interfaces described in the following table.

Table 3-1 Interfaces Required by the Plug-In Session EJB

Name	Interface	Description
Session EJB	`javax.ejb.SessionBean`	Interface that must be implemented by all session EJBs.
Plug-In Home Interface	`com.bea.wlpi.server.plugin.PluginHome`	Extension of the `javax.ejb.EJBHome` interface; it defines the home interface for all plug-ins.

Table 3-1 Interfaces Required by the Plug-In Session EJB (Continued)

Name	Interface	Description
Plug-In Remote Interface	`com.bea.wlpi.server.plugin.Plugin`	Extension of the `javax.ejb.EJBObject` interface; it defines the remote interface for all plug-ins.

The following sections describe each interface and the methods that you must implement when defining the plug-in session EJB. Excerpts from the plug-in sample are included.

Session EJB Interface

By definition, a session EJB must implement the `javax.ejb.SessionBean` and its methods.

Note: The contents of the `SessionBean` interface methods may be empty, or they may simply return a message to the log; but they must be implemented.

The following table lists the session EJB methods that you must implement.

Table 3-2 Session EJB Methods

Method	Description
`public void ejbActivate() throws javax.ejb.EJBException, java.rmi.RemoteException`	Activates instance.
`public void ejbPassivate() throws javax.ejb.EJBException, java.rmi.RemoteException`	Passivates instance.
`public void ejbRemove() throws javax.ejb.EJBException, java.rmi.RemoteException`	Removes instance.

Table 3-2 Session EJB Methods (Continued)

Method	Description
`public void setSessionContext(javax.ejb.SessionContext ctx) throws javax.ejb.EJBException, java.rmi.RemoteException`	Sets associated session context. The method parameter is defined as follows: *`ctx`*: `javax.ejb.SessionContext` object that specifies the session context.

For more information about these methods, see the `javax.ejb.SessionBean` Javadoc.

The following code listing is an excerpt from the plug-in sample that shows how to implement the `javax.ejb.SessionBean` interface and its methods. This excerpt is taken from the `SamplePluginBean.java` file in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory. Notable lines of code are shown in **bold**.

Listing 3-1 Implementing the Session EJB Interface

```
package com.bea.wlpi.tour.po.plugin;

import com.bea.wlpi.common.VersionInfo;
import com.bea.wlpi.common.plugin.*;
import com.bea.wlpi.common.plugin.PluginData;
import com.bea.wlpi.server.plugin.InstanceNotification;
import com.bea.wlpi.server.plugin.Plugin;
import com.bea.wlpi.server.plugin.PluginManagerCfg;
import com.bea.wlpi.server.plugin.PluginManagerCfgHome;
import com.bea.wlpi.server.plugin.TaskNotification;
import com.bea.wlpi.server.plugin.TemplateDefinitionNotification;
import com.bea.wlpi.server.plugin.TemplateNotification;
import java.lang.reflect.Constructor;
import java.util.MissingResourceException;
import java.util.ResourceBundle;
import java.util.Locale;
import java.net.URL;
import javax.ejb.CreateException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;
import java.io.*
/**
 * @homeInterface com.bea.wlpi.server.plugin.PluginHome
 * @remoteInterface com.bea.testplugin.SamplePlugin
 * @statemode Stateless
 */
public class SamplePluginBean implements SessionBean {
    private SessionContext ctx;
    private final static String PLUGIN_MANAGER_CFG_HOME =
        "java:comp/env/ejb/PluginManagerCfg";
    private static byte[] ICON_BYTE_ARRAY;

    // implements javax.ejb.SessionBean

    public void ejbActivate() {
    }

    // implements javax.ejb.SessionBean

    public void ejbRemove() {
    }

    // implements javax.ejb.SessionBean

    public void ejbPassivate() {
    }

    // implements javax.ejb.SessionBean

    public void setSessionContext(SessionContext ctx) {
        this.ctx = ctx;
    }
    .
    .
    .
```

For more information about the plug-in sample, see “BPM Plug-In Sample” on page 10-1.

Plug-In Home Interface

The home interface has been defined for you by the BPM plug-in framework. The `com.bea.wlpi.server.plugin.PluginHome` interface extends the `javax.ejb.EJBHome` interface and defines the home interface for all plug-ins.

The following table describes the method defined by the `PluginHome` home interface.

Table 3-3 Home Interface Method

Method	Description

`public com.bea.wlpi.server.plugin.Plugin create() throws java.rmi.RemoteException java.rmi.RemoteException`	Create a remote plug-in object interface.
	
	You can also set up a custom plug-in icon for the WebLogic Integration Studio interface view using the `imageStreamToByteArray()` method to the `com.bea.wlpi.common.plugin.InfoObject`. For example:
	
	`ICON_BYTE_ARRAY =`
	`InfoObject.imageStreamToByteArray(`
	`  getClass().getResourceAsStream(`
	`  “image”)`
	`);`
	
	Here *`image`* specifies the custom plug-in icon. For more information about the `imageStreamToByteArray()` method, see “InfoObject Object” on page B-28.
	
	When creating value objects for certain plug-in components, you can specify the icon to be used by the Studio to represent the plug-in component when the interface view is enabled. For more information, see “Defining Plug-In Value Objects” on page 2-6.
	
	This method returns a `com.bea.wlpi.server.plugin.Plugin` object.
	
	For information about how to enable the interface view within the Studio, see “Using the Studio Interface” in *Using the WebLogic Integration Studio*.

For more information about the home interface, see the `com.bea.wlpi.server.plugin.PluginHome` Javadoc.

The following code listing is an excerpt from the plug-in sample that shows how to implement `ejbCreate()` method for the single `create()` method that is declared in the home interface, and define a custom plug-in icon for the Studio interface view. This excerpt is taken from the `SamplePluginBean.java` file in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory.

Listing 3-2 Implementing the Home Interface Method

```
// implements javax.ejb.SessionBean

public void ejbCreate() throws CreateException {
    try {
        ICON_BYTE_ARRAY = InfoObject.imageStreamToByteArray(
                          getClass().getResourceAsStream("Sample.gif"));
    } catch (IOException ioe) {
        ioe.printStackTrace();
    }
}
```

For more information about the plug-in sample, see “BPM Plug-In Sample” on page 10-1.

Plug-In Remote Interface

The remote interface is defined for you by the BPM plug-in framework. The `com.bea.wlpi.server.plugin.Plugin` interface extends the `javax.ejb.EJBObject` interface and defines the remote interface for all plug-ins.

When defining the plug-in session EJB, you must implement the `Plugin` remote interface, including its methods.

The `Plugin` remote interface defines methods in the following categories:

- Lifecycle management
- Notification
- Plug-in information
- Object manufacturing

The following sections describe, by category, the `Plugin` remote interface methods that you must implement.

Note: The contents of the remote interface methods may remain empty, or they may simply return a message to the log; but they must be implemented.

Lifecycle Management Methods

The following table defines the lifecycle management methods that are defined by the remote interface that you must implement. For more information about the BPM plug-in lifecycle, see "Managing Lifecycle Tasks" on page 1-8.

Table 3-4 Remote Interface Lifecycle Management Methods

Method	Description
`epublic void exit() throws java.rmi.RemoteException, com.bea.wlip.common.plugin.PluginException`	Deinitializes the plug-in.
`public void init() throws java.rmi.RemoteException, com.bea.wlip.common.plugin.PluginException`	Initializes the plug-in. The Plug-in Manager calls this method only once during startup, regardless of the plug-in configuration or start mode.

Table 3-4 Remote Interface Lifecycle Management Methods (Continued)

Method	Description
`public void load( com.bea.wlpi.common.plugin.PluginObject` *`config`*`) throws java.rmi.RemoteException, com.bea.wlip.common.plugin.PluginException`	Loads the plug-in using the specified configuration, and registers the plug-in as a notification listener, if specified. For information about registering the plug-in as a notification listener, see "Using Plug-In Notifications" on page 5-1. The method parameter is defined as follows: *`config`*: `com.bea.wlpi.common.plugin.ConfigurationInfo` object, provided by the `com.bea.wlpi.common.plugin.PluginObject` object, that specifies the plug-in configuration data. **Note:** Plug-ins are responsible for replicating their private cluster-wide state throughout the cluster, if necessary.
`public void unload() throws java.rmi.RemoteException, com.bea.wlip.common.plugin.PluginException`	Unloads the plug-in, and unregisters it as a notification listener, if required. For information about registering and unregistering the plug-in as a notification listener, see "Using Plug-In Notifications" on page 5-1.

For more information about the methods of managing the remote interface lifecycle, see the `com.bea.wlpi.server.plugin.Plugin` Javadoc.

Notification Methods

The following table defines the notification methods that are defined by the remote interface that you must implement.

Note: In order to receive notifications, the plug-in must register as a notification listener. For more information, see "Using Plug-In Notifications" on page 5-1.

Table 3-5 Remote Interface Notification Methods

Method	Description
`public void instanceChanged( com.bea.wlpi.common.plugin.InstanceNotification` *`notify`*`) throws java.rmi.RemoteException, com.bea.wlpi.common.plugin.PluginException`	Notifies the plug-in of a change to a workflow instance. The method parameter is defined as follows: *`notify`*: `com.bea.wlpi.server.plugin.InstanceNotification` object that specifies the workflow instance change.
`public void taskChanged( com.bea.wlpi.common.plugin.TaskNotification` *`notify`*`) throws java.rmi.RemoteException, com.bea.wlpi.common.plugin.PluginException`	Notifies the plug-in of a change to a task instance. The method parameter is defined as follows: *`notify`*: `com.bea.wlpi.server.plugin.TaskNotification` object that specifies the task change.
`public void templateChanged( com.bea.wlpi.common.plugin.TemplateNotification` *`notify`*`) throws java.rmi.RemoteException, com.bea.wlpi.common.plugin.PluginException`	Notifies the plug-in of a change to a template. The method parameter is defined as follows: *`notify`*: `com.bea.wlpi.server.plugin.TemplateNotification` object that specifies the template change.
`public void templateDefinitionChanged( com.bea.wlpi.common.plugin.TemplateDefinitionNotification` *`notify`*`) throws java.rmi.RemoteException, com.bea.wliwlpip.common.plugin.PluginException`	Notifies the plug-in of a change to a template definition. The method parameter is defined as follows: *`notify`*: `com.bea.wlpi.server.plugin.TemplateDefinitionNotification` object that specifies the template definition change.

For more information about remote interface notification methods, see the `com.bea.wlpi.server.plugin.Plugin` Javadoc.

Plug-In Information Methods

The following table defines the plug-in information methods that are defined by the remote interface that you must implement.

Table 3-6 Remote Interface Plug-In Information Methods

Method	Description
`public java.lang.Class classForName( java.lang.String` *`className`*`) throws java.rmi.RemoteException, java.lang.ClassNotFoundException, com.bea.wlpi.common.plugin.PluginException`	Instantiates the plug-in-defined class by calling one of the plug-in-supplied metadata objects. The method parameter is defined as follows: *`className`*: `java.lang.String` object that specifies the fully qualified Java class name to instantiate. This method returns the class with the specified name.
`public com.bea.wlpi.common.plugin.PluginDependency[] getDependencies() throws java.rmi.RemoteException`	Gets the plug-ins on which the current plug-in depends. The Plug-in Manager ensures that all dependent plug-ins are loaded before attempting to load the current plug-in. This method returns a list of `com.bea.wlpi.common.plugin.PluginDependency` objects. To access information about each plug-in dependency, use the `PluginDependency` object methods described in "PluginCapabilitiesInfo Object" on page B-29.
`public java.lang.String getName() throws java.rmi.RemoteException`	Gets the globally unique name of the current plug-in, in reverse DNS format. **Note:** Reverse DNS format prevents global namespace collisions. This method returns a `java.lang.String` object that specifies the unique plug-in name.

Table 3-6 Remote Interface Plug-In Information Methods (Continued)

Method	Description
`public com.bea.wlpi.common.plugin.PluginCapabilitiesInfo getPluginCapabilitiesInfo( java.util.Locale` *`lc`*`, com.bea.wlpi.common.plugin.CategoryInfo` *`info`*`) throws java.rmi.RemoteException`	Gets detailed information about the plug-in capabilities. Use this method to: ■ Define the plug-in component value objects, as defined in “Using Plug-In Value Objects” on page 2-4. ■ Customize the action tree, if defining plug-in actions, as described in “Customizing the Action Tree” on page 4-68. ■ Register the plug-in exception handler, as described in “Defining the Plug-In Event Handler” on page 6-11. The method parameters are defined as follows: ■ *`lc`*: `java.util.Locale` object that specifies a locale in which to localize display strings. ■ *`info`*: `com.bea.wlpi.common.plugin.CategoryInfo` object that specifies the existing action category tree containing both predefined and plug-in defined actions. A NULL value specifies that the action category information should not be included. This method returns a `com.bea.wlpi.common.plugin.PluginCapabilitiesInfo` object, including new categories and actions to be inserted in the action tree, if requested. To access information about the plug-in capabilities, use the `PluginCapabilitiesInfo` object methods described in “PluginCapabilitiesInfo Object” on page B-29.

Table 3-6 Remote Interface Plug-In Information Methods (Continued)

Method	Description
`public com.bea.wlpi.common.plugin.PlugInfo getPluginInfo(java.util.Locale lc) throws java.rmi.RemoteException`	Gets basic information about the plug-in. The method parameter is defined as follows: *`lc`*: `java.util.Locale` object that specifies a locale in which to localize display strings. This method returns a `com.bea.wlpi.common.plugin.PluginInfo` object. To access information about the plug-in, use the `PluginInfo` object methods described in "PlugInfo Object" on page B-33.
`public com.bea.wlpi.common.VersionInfo getVersion() throws java.rmi.RemoteException`	Gets the plug-in version information. This method returns a `com.bea.wlpi.common.VersionInfo` object. To access information about the plug-in, use the `VersionInfo` object methods described in "VersionInfo Object" in "Value Object Summary" in *Programming BPM Client Applications*.
`public void setConfiguration(com.bea.wlpi.common.plugin.PluginObject config) throws java.rmi.RemoteException`	Sets the plug-in configuration information. The method parameter is defined as follows: *`config`*: `com.bea.wlpi.common.plugin.PluginInfo` object that specifies the plug-in configuration information. For more information, see "Managing Plug-Ins" on page 7-1.

For more information about remote interface plug-in information methods, see the `com.bea.wlpi.server.plugin.Plugin` Javadoc.

Object Manufacturing Method

The following table defines the object manufacturing method that are defined by the remote interface that you must implement.

Table 3-7 Remote Interface Object Manufacturing Method

Method	Description

`public java.lang.Object getObject(`
`java.util.Locale` *`lc`*`,`
`java.lang.String` *`className`*`) throws`
`java.rmi.RemoteException,`
`java.lang.ClassNotFoundException,`
`com.bea.wlip.common.plugin.PluginException`
	Gets the plug-in-defined object by calling one of the plug-in-supplied metadata objects.
	The method parameters are defined as follows:
	■ *`lc`*: `java.util.Locale` object that specifies a locale in which to localize display strings.
	■ *`className`*: `java.lang.String` object that specifies the fully qualified Java class name to instantiate.
	This method returns a `java.lang.Object` instance of the named class.

For more information about remote interface object manufacturing method, see the `com.bea.wlpi.server.plugin.Plugin` Javadoc.

Example of Implementing the Remote Interface

The following code listing is an excerpt from the plug-in sample that shows how to implement the remote interface and its methods. This excerpt is taken from the `SamplePluginBean.java` file in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory. Notable lines of code are shown in **bold**.

Listing 3-3 Implementing the Remote Interface

```
// Plugin implementation

/**
```

```
     * Initialize the Plugin.
     */
    public void init() {
        log("init called");
    }

    /**
     * Deinitialize the Plugin.
     */
    public void exit() {
        log("exit called");
    }

    /**
     * Load the Plugin. The plugin should register its
     * interest in various system events at this point.
     * @param pluginData Plugin configuration data.
     * @see PluginManager#addTemplateListener
     * @see PluginManager#addTemplateDefinitionListener
     * @see PluginManager#addInstanceListener
     * @see PluginManager#addTaskListener
     * @throws PluginException
     */
    public void load(PluginObject pluginData) throws PluginException {

        log("load called");
        // Enable this block to subscribe to notifications.

        /*
        PluginManagerCfg pm = null;
        try {
            pm = getPluginManagerCfg();
            Plugin plugin = (Plugin)ctx.getEJBObject();
         pm.addInstanceListener(plugin, PluginConstants.EVENT_NOTIFICATION_ALL);
            pm.addTaskListener(plugin, PluginConstants.EVENT_NOTIFICATION_ALL);
            pm.addTemplateDefinitionListener(plugin,
PluginConstants.EVENT_NOTIFICATION_ALL);
         pm.addTemplateListener(plugin, PluginConstants.EVENT_NOTIFICATION_ALL);
        } catch (Exception e) {
            e.printStackTrace();
           throw new PluginException(SamplePluginConstants.PLUGIN_NAME, "Unable
to get PluginManager");
        } finally {
            try {
                if (pm != null)
                    pm.remove();
            } catch (Exception e) {
            }
        }
        */
        log("loaded");
    }

    /**
     * Unload the plugin.  The plugin framework will deregister the plugin
     * if it had subscribed to any event notifications.
     */
    public void unload() {
        log("unload called");
    }

    public PluginDependency[] getDependencies() {

        log("getDependencies called");

        return null;
    }

    public String getName() {
        return SamplePluginConstants.PLUGIN_NAME;
    }

    public VersionInfo getVersion() {
        return SamplePluginConstants.PLUGIN_VERSION;
    }

    /**
     * Return descriptive information about the plugin
     * @param lc The locale in which to localize display strings.
     * @return Descriptive information about the Plugin.
     */
    public PluginInfo getPluginInfo(Locale lc) {

        log("getPluginInfo called");

        return createPluginInfo(lc);
    }

    public void setConfiguration(PluginObject config) throws PluginException {
    }

    /**
     * Return a complete description of the plugins capabilities. To add new
     * subcategories and actions, the plugin should use the category IDs
     * passed in via the <code>info</code> parameter passed to identify the
     * parent categories, and {@link ActionCategoryInfo#ID_PLUGIN} as the new
category ID.
     * The PluginManager merges the CategoryInfo array returned by this call with
     * the current structure (as passed via the <code>info</code> parameter), and
     * replaces references to {@link ActionCategoryInfo#ID_PLUGIN} with
newly-assigned
     * unique category IDs. The predefined categories have the following IDs:
    * {@link ActionCategoryInfo#ID_TASK}, {@link ActionCategoryInfo#ID_WORKFLOW},
     * {@link ActionCategoryInfo#ID_INTEGRATION}, {@link
ActionCategoryInfo#ID_MISCELLANEOUS},
     * {@link ActionCategoryInfo#ID_EXCEPTION}.
     * @param lc Locale in which to localize display strings.
     * @param info The existing action category tree, containing
     * categories and actions both predefined and plugin-defined
     * (i.e., by previously loaded plugins).
     * @return New categories and actions to be inserted in the tree.
     */
    public PluginCapabilitiesInfo getPluginCapabilitiesInfo(Locale lc,
            CategoryInfo[] info) {

        PluginInfo pi;
        FieldInfo orderFieldInfo;
        FieldInfo confirmFieldInfo;
        FieldInfo[] fieldInfo;
        FunctionInfo fi;
        FunctionInfo[] functionInfo;
        StartInfo si;
        StartInfo[] startInfo;
        EventInfo ei;
        EventInfo[] eventInfo;
        SampleBundle bundle = new SampleBundle(lc);

        log("getPluginCapabilities called");

        pi = createPluginInfo(lc);
        orderFieldInfo =
            new FieldInfo(SamplePluginConstants.PLUGIN_NAME, 3,
                          bundle.getString("orderFieldName"),
                          bundle.getString("orderFieldDesc"),
                          SamplePluginConstants.ORDER_FIELD_CLASSES, false);
        confirmFieldInfo =
            new FieldInfo(SamplePluginConstants.PLUGIN_NAME, 4,
                          bundle.getString("confirmFieldName"),
                          bundle.getString("confirmFieldDesc"),
                          SamplePluginConstants.CONFIRM_FIELD_CLASSES, false);
        fieldInfo = new FieldInfo[]{ orderFieldInfo, confirmFieldInfo };
        ei = new EventInfo(SamplePluginConstants.PLUGIN_NAME, 6,
                           bundle.getString("confirmOrderName"),
                          bundle.getString("confirmOrderDesc"), ICON_BYTE_ARRAY,
                           SamplePluginConstants.EVENT_CLASSES,
                           confirmFieldInfo);
        eventInfo = new EventInfo[]{ ei };
        fi = new FunctionInfo(SamplePluginConstants.PLUGIN_NAME, 7,
                              bundle.getString("calcTotalName"),
                              bundle.getString("calcTotalDesc"),
                              bundle.getString("calcTotalHint"),
                              SamplePluginConstants.FUNCTION_CLASSES, 3, 3);
        functionInfo = new FunctionInfo[]{ fi };
        si = new StartInfo(SamplePluginConstants.PLUGIN_NAME, 5,
                           bundle.getString("startOrderName"),
                           bundle.getString("startOrderDesc"), ICON_BYTE_ARRAY,
                           SamplePluginConstants.START_CLASSES, orderFieldInfo);
        startInfo = new StartInfo[]{ si };

        PluginCapabilitiesInfo pci = new PluginCapabilitiesInfo(pi,
                                         getCategoryInfo(bundle), eventInfo,
                                         fieldInfo, functionInfo, startInfo,
                                         null, null, null, null, null);

        return pci;
    }

    /**
     * Return a plugin-defined class.  The caller retrieves the class name by
     * calling on one of the plugin-supplied metadata objects.
     * @param className The fully qualified Java class name to instantiate.
     * @return The class with the specified name.
     * @throws ClassNotFoundException if the plugin could not load the class.
     * @see ActionInfo
     * @see EventInfo
     * @see FunctionInfo
     * @see FieldInfo
     * @see PluginInfo
     * @see StartInfo
     * @see DoneInfo
     * @see VariableTypeInfo
     * @see TemplatePropertiesInfo
     * @see TemplateDefinitionPropertiesInfo
     * @throws PluginException
     */
    public Class classForName(String className)
            throws ClassNotFoundException, PluginException {

        log("classForName called");

        return Class.forName(className);
    }

    /**
     * Return a plugin-defined object.  The caller retrieves the class name by
     * calling on one of the plugin-supplied metadata objects.
     * @param lc The locale in which to localize display strings.
     * @param className The fully qualified Java class name to instantiate.
     * @return
     * @see ActionInfo
     * @see EventInfo
     * @see FunctionInfo
     * @see FieldInfo
     * @see PluginInfo
     * @see StartInfo
     * @see DoneInfo
     * @see VariableTypeInfo
     * @see TemplateInfo
     * @see TemplateDefinitionInfo
     * @throws ClassNotFoundException
     * @throws PluginException
     */
    public Object getObject(Locale lc, String className)
            throws ClassNotFoundException, PluginException {

        log("getObject called with class name " + className);

        try {
            Class cls = Class.forName(className);
            Object obj;

            try {
                // see if there is a ctor that takes a Locale object
                Class[] paramType = new Class[]{ lc.getClass() };
                Constructor ctor = cls.getConstructor(paramType);
                Object[] param = new Object[]{ lc };

                obj = ctor.newInstance(param);

                return (obj);
            } catch (Exception e) {
            }

            // if not ctor takes a Locale just call the no-arg ctor
            return Class.forName(className).newInstance();
        } catch (InstantiationException ie) {
            throw new PluginException(SamplePluginConstants.PLUGIN_NAME,
                                      "Unable to instantiate class: " + ie);
        } catch (IllegalAccessException iae) {
            throw new PluginException(SamplePluginConstants.PLUGIN_NAME,
                                      "Unable to instantiate class: " + iae);
        }
    }

    /**
```

```
     * Notifies a plugin of a change in a template.
     * @param e An event object describing the change.
     */
    public void templateChanged(TemplateNotification e) {
        log("templateChanged called");
    }

    /**
     * Notifies a plugin of a change in a template definition.
     * @param e An event object describing the change.
     */
    public void templateDefinitionChanged(TemplateDefinitionNotification e) {
        log("templateDefinitionChanged called");
    }

    /**
     * Notifies a plugin of a change in a workflow instance.
     * @param e An event object describing the change.
     */
    public void instanceChanged(InstanceNotification e) {
        log("instanceChanged called");
    }

    /**
     * Notifies a plugin of a change in a task instance.
     * @param e An event object describing the change.
     */
    public void taskChanged(TaskNotification e) {
        log("taskChanged called");
    }


    private PluginInfo createPluginInfo(Locale lc) {

        HelpSetInfo helpSet;
        PluginInfo pi;
        SampleBundle bundle = new SampleBundle(lc);
        String name = bundle.getString("pluginName");
        String desc = bundle.getString("pluginDesc");
        String helpName = bundle.getString("helpName");
        String helpDesc = bundle.getString("helpDesc");

        helpSet = new HelpSetInfo(SamplePluginConstants.PLUGIN_NAME, helpName,
                                  helpDesc,
                                  new String[]{ "htmlhelp/Sample", "index" },
                                  HelpSetInfo.HELP_HTML);
        pi = new PluginInfo(SamplePluginConstants.PLUGIN_NAME, name, desc, lc,
                            SamplePluginConstants.VENDOR_NAME,
                            SamplePluginConstants.VENDOR_URL,
                              SamplePluginConstants.PLUGIN_VERSION,
                              SamplePluginConstants.PLUGIN_FRAMEWORK_VERSION,
                              null, null, helpSet);

        return pi;
    }

    // It is necessary to create these objects afresh each time, because we
    // relinquish ownership of the result and the plugin framework assigns
    // a system ID to each item.  Reassignment will cause an
    // IllegalStateException.
    private CategoryInfo[] getCategoryInfo(SampleBundle bundle) {

        ActionInfo checkInventoryAction =
            new ActionInfo(SamplePluginConstants.PLUGIN_NAME, 1,
                           bundle.getString("checkInventoryName"),
                        bundle.getString("checkInventoryDesc"), ICON_BYTE_ARRAY,
                           ActionCategoryInfo.ID_NEW,
                           ActionInfo.ACTION_STATE_ALL,
                           SamplePluginConstants.CHECKINV_CLASSES);
        ActionInfo sendConfirmAction =
            new ActionInfo(SamplePluginConstants.PLUGIN_NAME, 2,
                           bundle.getString("sendConfirmName"),
                           bundle.getString("sendConfirmDesc"), ICON_BYTE_ARRAY,
                           ActionCategoryInfo.ID_NEW,
                           ActionInfo.ACTION_STATE_ALL,
                           SamplePluginConstants.SENDCONF_CLASSES);
        ActionCategoryInfo[] actions =
            new ActionCategoryInfo[]{ checkInventoryAction, sendConfirmAction };
        CategoryInfo[] catInfo =
          new CategoryInfo[]{ new CategoryInfo(SamplePluginConstants.PLUGIN_NAME,
                                                 0, bundle.getString("catName"),
                                                 bundle.getString("catDesc"),
                                                 ActionCategoryInfo.ID_NEW,
                                                 actions) };

        return catInfo;
    }

    private void log(String msg) {
        System.out.println("SamplePlugin: " + msg);
    }
```

For more information about the plug-in sample, see “BPM Plug-In Sample” on page 10-1.

CHAPTER

4 Defining Plug-In Components

This section explains how to define plug-in components. It includes the following topics:

- Overview
- PluginObject Interface
- Reading and Saving Plug-In Data
- Displaying the Plug-In GUI Component
- Executing the Plug-In
- Using Plug-In Run-Time Contexts
- Defining the Plug-In Component Value Objects

Overview

As described in "How BPM Discovers a Deployed Plug-In" on page 1-8, the plug-in is responsible for enabling the BPM to:

- Access the plug-in implementation to read, display, and save the plug-in within the design client
- Execute the plug-in

This functionality is provided by the plug-in component. The following table describes the plug-in component requirements for supporting the specified functionality.

Table 4-1 Plug-In Component Requirements

To enable the BPM to . . .	You must . . .
Read (parse) and save plug-in data in XML format	Implement the plug-in data interface. For example, see `EventNodeData.java` in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory of the plug-in sample.
Display the plug-in GUI component within the design client	Define the plug-in panel class. For example, see `EventNodePanel.java` in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory of the plug-in sample.
Execute the plug-in	Define the run-time component class. For example, see `EventNode.java` in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory of the plug-in sample.

To enable the plug-in to read (parse) incoming data, both the plug-in data interface and run-time component class must implement the `load()` (parsing) method of their parent interface, `com.bea.wlpi.common.plugin.PluginObject`.

Lastly, you must define the plug-in component value object to describe the component data.

The following sections describe the `PluginObject` interface, explain how to define the plug-in component to support the functionality listed in the previous table, and define the plug-in component value object.

Note: For a summary of the steps that must be accomplished to define each type of plug-in component, see "Plug-In Component Definition Roadmap" on page A-1.

PluginObject Interface

The `com.bea.wlpi.common.plugin.PluginObject` interface enables the plug-in to read (parse) the plug-in data.

This interface must be extended by:

- Plug-in data interface described in "Reading and Saving Plug-In Data" on page 4-9
- Run-time component class described in "Executing the Plug-In" on page 4-59

The `PluginObject` interface defines one method, `load()`, as shown in the following table.

Table 4-2 PluginObject Interface Method

Method	Description
`public void load(org.xml.sax.XMLReader` *`parser`*`)`	Notifies the plug-in to prepare to load its data from an XML document. The method parameter is defined as follows. *`parser`*: `org.xml.sax.XMLReader` object that specifies a valid SAX parser. To use multiple content handlers while parsing data (useful when parsing deeply nested elements , the plug-in can save this value and call the `setContentHandler()` method on the specified *`parser`* object.

The Plug-in Manager calls the `load()` method when it encounters the plug-in section (for example, a `<plugin-data>` element) in an XML document. This might happen, for example, when the Plug-in Manager opens a template, template definition, or plug-in configuration XML document in the WebLogic Integration Studio.

Note: For information about the BPM DTDs, see "DTD Formats" in *Programming BPM Client Applications*.

You must also implement required content handler methods, including the `startElement()` and `endElement()` methods. The Plug-in Manager sets the plug-in as the parser content handler, and uses the `startElement()` and `endElement()` methods as the first and last calls to the content handler when a `<plugin-data>` element is reached. The content handler uses the intervening SAX notifications to store the plug-in-specific data. For more information about the content handler methods, see the `org.xml.sax` Javadoc.

In the plug-in sample, a separate class file is provided for certain plug-in components that extends the `PluginObject` interface and defines the required methods. This file does not need to be defined separately. It is useful in this case, however, because it provides a single definition for the multiple classes in the example that share the file.

The following sections provide code examples showing how the `PluginObject` interface for plug-in Done and Start nodes is implemented.

In addition to these examples, refer to the following files in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory:

This file . . .	Illustrates PluginObject implementation for a . . .
`EventObject.java`	Plug-in event
`CheckInventoryActionObject.java`	Plug-in action
`SendConfirmActionObject.java`	Plug-in action

For more information about the plug-in sample, see "BPM Plug-In Sample" on page 10-1.

Done Node Example

The following code listing shows how to define a class that implements the `PluginObject` interface for a Done node. The input to the example code is a user response to a decision dialog box (yes or no). Notable lines of code are shown in **bold**.

Note: This class is not available as part of the plug-in sample.

Listing 4-1 Implementing the PluginObject Interface for a Done Node

```
package com.bea.wlpi.test.plugin;

import java.io.IOException;
import com.bea.wlpi.common.plugin.PluginObject;
import org.xml.sax.*;

public class DoneObject implements PluginObject
{

   protected String yesOrNo = null;
   protected static String YESORNO_TAG = "yesorno";
   protected transient String        lastValue;

   public DoneObject()
   {
   }

   public DoneObject(String yesOrNo)
   {
      this.yesOrNo = yesOrNo;
   }

   public void load(XMLReader parser)
   {
   }

   void setYesOrNo(String decision)
   {
      yesOrNo = decision;
   }
   String getYesOrNo()
   {
      return yesOrNo;
   }
   public void setDocumentLocator(Locator locator)
   {
   }

   public void startDocument()
   throws SAXException
   {
   }

   public void endDocument()
   throws SAXException
   {
   }

   public void startPrefixMapping(String prefix, String uri)
   throws SAXException
   {
   }

   public void endPrefixMapping(String prefix)
   throws SAXException
   {
   }

   public void startElement(String namespaceURI, String localName, String qName,
Attributes atts)
   throws SAXException
   {
      lastValue = null;
   }

   public void endElement(String namespaceURI, String localName, String name)
   throws SAXException
   {
      if(name.equals(YESORNO_TAG))
         yesOrNo = lastValue;
   }

   public void characters(char[] ch, int start, int length)
   throws SAXException
   {
      String value = new String(ch, start, length);

      if(lastValue == null)
         lastValue = value;
      else
         lastValue = lastValue + value;
   }
   public void ignorableWhitespace(char[] ch, int start, int length)
   throws SAXException
   {
   }

   public void processingInstruction(String target, String data)
   throws SAXException
   {
   }

   public void skippedEntity(String name)
   throws SAXException
   {
   }
}
```

Refer to the following related example listings:

- “Implementing the PluginData Interface for a Done Node” on page 4-13 shows how to read and save the plug-in data in XML format. This example extends the class shown in the previous example.
- “Defining the PluginPanel Class for a Done Node” on page 4-32 shows how to display the plug-in GUI component in the design client.
- “Defining the Run-Time Component Class for a Done Node” on page 4-73 shows how to define the execution information for the plug-in.

Start Node Example

The following code listing is an excerpt from the plug-in sample that shows how to define a class that implements the `PluginObject` interface for a Start node. Note that the `load()`, `startelement()`, and `endelement()` method are defined. This excerpt is taken from the `StartObject.java` file in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory. Notable lines of code are shown in **bold**.

Listing 4-2 Implementing the PluginObject Interface for a Start Node

```
public class StartObject implements PluginObject {
.
.
.
    public void load(XMLReader parser) {
    }
.
.
.
   public void startElement(String namespaceURI, String localName, String qName,
Attributes atts)
            throws SAXException {
        lastValue = null;
    }

    public void endElement(String namespaceURI, String localName, String name)
            throws SAXException {
        if (name.equals(EVENTDESC_TAG))
            eventDesc = lastValue;
    }
     .
     .
     .
```

Refer to the following related example listings:

- "Implementing the PluginData Interface for a Start Node" on page 4-16 shows how to read and save plug-in data in XML format. This example extends the class shown in the previous example.
- "Defining the PluginTriggerPanel Class for a Start Node" on page 4-48 shows how to display the plug-in GUI component in the design client.
- "Defining the Run-Time Component Class for a Start Node" on page 4-91 shows how to define the execution information for the plug-in.
- "Using Plug-In Run-Time Contexts" on page 4-94 shows how to define the plug-in fields that can be referenced from an evaluator expression.

For more information about the plug-in sample, see "BPM Plug-In Sample" on page 10-1.

Reading and Saving Plug-In Data

To read (parse) and save plug-in data in XML format, you must implement the plug-in data interface.

Note: For each plug-in data interface class that is defined by a BPM plug-in value object via the `KEY_DATA`, `KEY_PANEL`, and `KEY_RENDERER` values, you must provide a public constructor that requires no arguments to support remote class loading on the client. This public constructor is not required to be supplied for the plug-in defined classes that are referenced by this class. For more information about using BPM plug-in value objects, see Chapter 2, “Plug-In Development Fundamentals.”

This is a requirement for WebLogic Integration Release 2.1 Service Pack 1. If you do not provide a no-argument constructor for classes generated using an earlier release of WebLogic Integration, the classes will be instantiated, but you may receive exceptions if the client and server platforms are incompatible.

To enable the plug-in to read (parse) incoming data, the plug-in data interface class must implement the `load()` (parsing) method of its parent interface, `com.bea.wlpi.common.plugin.PluginObject`.

To enable the plug-in to save its data in XML format, you must implement one of the plug-in data interfaces defined in the following table based on the type of plug-in component being defined. Data must be saved in XML format, for example, when you are saving a template, template definition, or plug-in configuration XML document in the Studio.

Note: You do not need to implement the plug-in data interface to read and save data for the following plug-in components: functions, message types, and variable types.

Table 4-3 Plug-In Data Interfaces

To define the following plug-in . . .	You must implement . . .
Any plug-in component	`com.bea.wlpi.common.plugin.PluginData` to enable the plug-in component to save its data in XML format. When defining actions, you should implement the `PluginActionData` interface, which extends this interface.
Action	`com.bea.wlpi.common.plugin.PluginActionData` to enable the plug-in action to save its data in XML format. This class is used by the Action Plugin dialog box in the Studio, which provides generic support for subactions. **Note:** `PluginActionData` extends the `PluginData` interface defined previously in this table.

Note: For information about the BPM DTDs and examples of plug-in-specific output, see "DTD Formats" in *Programming BPM Client Applications*.

Each plug-in data interface is defined in more detail in the following sections.

Implementing the PluginData Interface

You must implement the `com.bea.wlpi.common.plugin.PluginData` interface to enable the plug-in component to save its data in XML format.

Note: When defining actions, you should implement the `PluginActionData` interface, as described in "Implementing the PluginActionData Interface" on page 4-20.

The following table describes the methods defined by the `PluginData` interface that you must implement.

Note: The contents of the `PluginData` interface methods may be empty or simply return a message to the log, but they must be implemented.

Table 4-4 PluginData Interface Methods

Method	Description
`public java.lang.Object clone()`	Clones the plug-in data. This method returns a `java.lang.Object` instance that specifies a deep (recursive) copy of the graph for this object.
`public java.lang.String getPrintableData()`	Gets a printable description of the plug-in data. This method is typically used when a template definition is printed out. This method returns a `java.lang.String` object that specifies the printable data. This value should be localized ujsing the locale specified in the plug-in data constructor.

Table 4-4 PluginData Interface Methods (Continued)

Method	Description
`public java.util.List getReferencedPublishables(java.util.Map` *`publishables`*`)`	Gets the referenced publishable objects. Enables design-time clients to package a workflow definition with its dependencies so the resulting package can be imported and run. Publishable objects include: templates, template definitions, business calendars, business operations, event keys, and repository items. Plug-ins that contain references to these objects must declare them when this method is called. The user creating an export package can then specify which of the referenced objects should be included in the package. The method parameter is defined as follows. *`publishables`*: `java.util.Map` object that specifies a map of all publishable objects, keyed on the constants defined in the `com.bea.wlpi.common.Publishable` interface. The values in the map are homogenous `java.util.List` objects containing value objects of a type that matches their corresponding keys. The plug-in must add the appropriate objects in these lists to the returned list, as the design client expects a list of references to the actual objects. This method returns a list of `com.bea.wlpi.common.Publishable` objects. For more information about publishable objects, see "Publishing Workflow Objects" in *Programming BPM Client Applications*.

Table 4-4 PluginData Interface Methods (Continued)

Method	Description
`public void save(com.bea.wlpi.common.XMLWriter` *`writer`*`, int` *`indent`*`) throws java.io.IOException`	Saves data in an XML document. The Plug-in Manager calls this method when it encounters the plug-in section (for example, a `<plugin-data>` element) in an XML document. This occurs, for example, when a template, template definition, or plug-in configuration XML document is being saved in the Studio. The method parameters are defined as follows: ■ *`writer`*: `com.bea.wlpi.common.XMLWriter` object that specifies the XMLWriter to use to serialize the plug-in data. ■ *`indent`*: Integer value that specifies the indentation level. You should use this value to create a correctly indented XML document. The default indentation is two spaces.

The following sections provide code examples showing how the `PluginData` interface is implemented.

Done Node Example

The following code listing shows how to define a class that implements the `PluginData` interface for a Done node. Notable lines of code are shown in **bold**.

Note: This class is not available as part of the plug-in sample.

Listing 4-3 Implementing the PluginData Interface for a Done Node

```
package com.bea.wlpi.test.plugin;

import com.bea.wlpi.common.XMLWriter;
import com.bea.wlpi.common.plugin.PluginData;
import java.io.IOException;
import java.util.List;
import java.util.Map;
import org.xml.sax.*;


public class DoneNodeData extends DoneObject implements PluginData
{
    public static int count = 0;
    private int c;

    public DoneNodeData()
    {
        c=count++;
    }

    public DoneNodeData(String yesOrNo)
    {
        super(yesOrNo);
        c=count++;
    }

    public void save(XMLWriter writer, int indent) throws IOException
    {
        writer.saveElement(indent, YESORNO_TAG, yesOrNo);
    }
}
```

Refer to the following related example listings:

- “Implementing the PluginObject Interface for a Done Node” on page 4-5 shows how to read the plug-in data in XML format. This class is extended by the class shown in the previous example.
- “Defining the PluginPanel Class for a Done Node” on page 4-32 shows how to display the plug-in GUI component in the design client.
- “Defining the Run-Time Component Class for a Done Node” on page 4-73 shows how to define the execution information for the plug-in.

Event Node Example

The following code listing is an excerpt from the plug-in sample that shows how to define a class that implements the `PluginData` interface for an Event node. This excerpt is taken from the `EventNodeData.java` file in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory. Notable lines of code are shown in **bold**.

Listing 4-4 Implementing the PluginData Interface for an Event Node

```
package com.bea.wlpi.tour.po.plugin;

import com.bea.wlpi.common.XMLWriter;
import com.bea.wlpi.common.plugin.PluginData;
import java.io.IOException;
import java.util.List;
import java.util.Locale;
import java.util.ResourceBundle;
import java.util.Map;
import org.xml.sax.*;

public class EventNodeData extends EventObject implements PluginData {
    private SampleBundle bundle;

    public EventNodeData() {
        this(Locale.getDefault());
    }

    public EventNodeData(Locale lc) {
        eventDesc = SamplePluginConstants.CONFIRM_EVENT;
        bundle = new SampleBundle(lc);
    }

    public void save(XMLWriter writer, int indent) throws IOException {
        writer.saveElement(indent, EVENTDESC_TAG, eventDesc);
    }

    public List getReferencedPublishables(Map publishables) {
        return null;
    }

    public String getPrintableData() {
        return bundle.getString("confirmOrderName");
    }
    public Object clone() {
        return new EventNodeData(bundle.getLocale());
    }
}
```

Refer to the following related example listings:

- `EventObject.java` in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory shows how to read the plug-in data in XML format. This class is extended by the class shown in the previous example.
- "Defining the PluginTriggerPanel Class for an Event Node" on page 4-51 shows how to display the plug-in GUI component in the design client.
- "Defining the Run-Time Component Class for an Event Node" on page 4-78 shows how to define the execution information for the plug-in.

For more information about the plug-in sample, see "BPM Plug-In Sample" on page 10-1.

Start Node Example

The following code listing is an excerpt from the plug-in sample that shows how to define a class that implements the `PluginData` interface for a Start node. This excerpt is taken from the `StartNodeData.java` file in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory. Notable lines of code are shown in **bold**.

Listing 4-5 Implementing the PluginData Interface for a Start Node

```
package com.bea.wlpi.tour.po.plugin;

import com.bea.wlpi.common.XMLWriter;
import com.bea.wlpi.common.plugin.PluginData;
import java.io.IOException;
import java.util.List;
import java.util.Locale;
import java.util.ResourceBundle;
import java.util.Map;
import org.xml.sax.*;
```

```
public class StartNodeData extends StartObject implements PluginData {
    private SampleBundle bundle;

    public StartNodeData() {
        this(Locale.getDefault());
    }

    public StartNodeData(Locale lc) {
        eventDesc = SamplePluginConstants.START_ORDER_EVENT;
        bundle = new SampleBundle(lc);
    }

    public void save(XMLWriter writer, int indent) throws IOException {
        writer.saveElement(indent, EVENTDESC_TAG, eventDesc);
    }

    public List getReferencedPublishables(Map publishables) {
        return null;
    }

    public String getPrintableData() {
        return bundle.getString("startOrderLabel");
    }

    public Object clone() {
        return new StartNodeData(bundle.getLocale());
    }
}
```

Refer to the following related example listings:

- “Implementing the PluginObject Interface for a Start Node” on page 4-8 shows how to read plug-in data in XML format. This class is extended by the class shown in the previous example.
- “Defining the PluginTriggerPanel Class for a Start Node” on page 4-48 shows how to display the plug-in GUI component in the design client.
- “Defining the Run-Time Component Class for a Start Node” on page 4-91 shows how to define the execution information for the plug-in.
- “Using Plug-In Run-Time Contexts” on page 4-94 shows how to define the plug-in fields that can be referenced from an evaluator expression.

For more information about the plug-in sample, see “BPM Plug-In Sample” on page 10-1.

Workflow Template Properties Example

The following code listing shows how to define a class that implements the `PluginData` interface for workflow template properties. The code reads and saves the user’s response to a decision dialog box (yes or no). Notable lines of code are shown in **bold**.

Note: This class is not available as part of the plug-in sample.

Listing 4-6 Implementing the PluginData Interface for Workflow Template Properties

```
package com.bea.wlpi.test.plugin;

import com.bea.wlpi.common.XMLWriter;
import com.bea.wlpi.common.plugin.PluginData;
import java.io.IOException;
import java.util.List;
import java.util.Map;
import org.xml.sax.*;

public class TemplatePropertiesData extends DoneObject implements PluginData {

    public TemplatePropertiesData() {
    }

    public TemplatePropertiesData(String yesOrNo){
        super(yesOrNo);
    }

    public void save(XMLWriter writer, int indent) throws IOException {
        writer.saveElement(indent, YESORNO_TAG, yesOrNo);
    }

    public List getReferencedPublishables(Map publishables) {
        return null;
    }

    public String getPrintableData() {
        return null;
    }
}
```

Refer to the following related example listings:

- "Implementing the PluginObject Interface for a Done Node" on page 4-5 shows how to read the plug-in data in XML format. This class is extended by the class shown in the previous example.
- "Defining the PluginPanel Class for Workflow Template Properties" on page 4-35 shows how to display the plug-in GUI component in the design client.

Workflow Template Definition Properties Example

The following code listing shows how to define a class that implements the `PluginData` interface for workflow template definition properties. The code reads and saves the user's response to a decision dialog box (yes or no). Notable lines of code are shown in **bold**.

Note: This class is not available as part of the plug-in sample.

Listing 4-7 Implementing the PluginData Interface for Workflow Template Definition Properties

```
package com.bea.wlpi.test.plugin;

import com.bea.wlpi.common.XMLWriter;
import com.bea.wlpi.common.plugin.PluginData;
import java.io.IOException;
import java.util.List;
import java.util.Map;
import org.xml.sax.*;


public class TemplateDefinitionPropertiesData extends DoneObject implements
PluginData
{

    public TemplateDefinitionPropertiesData()
    {
    }
    public TemplateDefinitionPropertiesData(String yesOrNo)
    {
        super(yesOrNo);
    }

    public void save(XMLWriter writer, int indent) throws IOException
    {
        writer.saveElement(indent, YESORNO_TAG, yesOrNo);
    }

    public List getReferencedPublishables(Map publishables) {
        return null;
    }

    public String getPrintableData() {
        return null;
    }

    public Object clone() {
        return new TemplateDefinitionPropertiesData(yesOrNo);
    }
}
```

Refer to the following related example listings:

- "Implementing the PluginObject Interface for a Done Node" on page 4-5 shows how to read the plug-in data in XML format. This class is extended by the class shown in the previous example.
- "Defining the PluginPanel Class for Workflow Template Definition Properties" on page 4-38 shows how to display the plug-in GUI component in the design client.

Implementing the PluginActionData Interface

You must implement the `com.bea.wlpi.common.plugin.PluginActionData` interface to enable the plug-in action to save its data in XML format.

Note: The `PluginActionData` interface extends the `PluginData` interface. For more information about the `PluginData` interface methods, see the table "PluginData Interface Methods" on page 4-11.

The following table describes the method defined by the `PluginActionData` interface that you must implement.

Note: The contents of the `PluginActionData` interface methods may be empty or simply return a message to the log, but they must be implemented.

Table 4-5 PluginActionData Interface Method

Method	Description
`public java.lang.String getLabel()`	Gets the formatted label of the plug-in action that is specified in the actions list.

The following code listing is an excerpt from the plug-in sample that shows how to define a class that implements the `PluginActionData` interface. This excerpt is taken from the `CheckInventoryActionData.java` file in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory. Notable lines of code are shown in **bold**.

Note: Refer to `SendConfirmationActionData.java` file in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory for another example of how to define a class the implements the `PluginActionData` interface.

Listing 4-8 Implementing the PluginActionData Interface

```
package com.bea.wlpi.tour.po.plugin;

import com.bea.wlpi.common.XMLWriter;
import com.bea.wlpi.common.plugin.PluginData;
import com.bea.wlpi.common.plugin.PluginActionData;
import java.io.IOException;
import java.util.ResourceBundle;
import java.util.Locale;
import java.util.List;
import java.util.Map;
import org.xml.sax.*;

public class CheckInventoryActionData extends CheckInventoryActionObject
        implements PluginActionData {
    private SampleBundle bundle;
    public CheckInventoryActionData() {
        getBundle(Locale.getDefault());
    }

    public CheckInventoryActionData(Locale lc) {
        getBundle(lc);
    }

    public CheckInventoryActionData(Locale lc, String inputVariableName,
                                    String outputVariableName) {

        super(inputVariableName, outputVariableName);

        getBundle(lc);
    }

    public void save(XMLWriter writer, int indent) throws IOException {
        writer.saveElement(indent, INPUTVARIABLE_TAG, inputVariableName);
        writer.saveElement(indent, OUTPUTVARIABLE_TAG, outputVariableName);
    }

    private void getBundle(Locale lc) {
        bundle = new SampleBundle(lc);
    }

    public List getReferencedPublishables(Map publishables) {
        return null;
    }

    public String getPrintableData() {
        return bundle.getString("checkInventoryDesc");
    }
    public Object clone() {

        return new CheckInventoryActionData(bundle.getLocale(),
                   new String(this.inputVariableName),
                   new String(this.outputVariableName));
    }

    public String getLabel() {
        return bundle.getString("checkInventoryDesc");
    }
}
```

Refer to the following related example listings:

- `CheckInventoryActionObject.java` in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory shows how to read the plug-in data in XML format. This class is extended by the class shown in the previous example.
- "Defining the PluginActionPanel Class" on page 4-40 shows how to display the plug-in GUI component in the design client.
- "Defining the Run-Time Component Class for an Action" on page 4-66 shows how to define the execution information for the plug-in.
- "Customizing an Action Tree" on page 4-70 shows how to customize the actions and/or action categories listed in the action trees that are displayed in various dialog boxes within the Studio.

For more information about the plug-in sample, see "BPM Plug-In Sample" on page 10-1.

Displaying the Plug-In GUI Component

To display the plug-in GUI component within the design client, all plug-ins must define a class that extends the plug-in panel class.

For example, in the figure "Plug-In Example: Start Node" on page 1-2, when a user selects the Start Order event as the Start node trigger, the Plug-in Manager loads the plug-in panel class, `StartNodePanel`, and it is instantiated by the Studio client using the no-argument constructor. The Studio client subsequently displays the plug-in GUI component in the Start Properties dialog box. (For more information about remote class loading, see "Accessing the Plug-In Implementation" on page 1-9.)

Note: For each plug-in GUI component class that is defined by a BPM plug-in value object via the `KEY_DATA`, `KEY_PANEL`, and `KEY_RENDERER` values, you must provide a public constructor that requires no arguments to support remote class loading on the client.This public constructor is not required to be supplied for the plug-in defined classes that are referenced by this class. For more information about using BPM plug-in value objects, see Chapter 2, "Plug-In Development Fundamentals."

This is a requirement for WebLogic Integration Release 2.1 Service Pack 1. If you do not provide a no-argument constructor for classes generated using an earlier release of WebLogic Integration, the classes will be instantiated, but you may receive exceptions if the client and server platforms are incompatible.

The following table describes the plug-in panel classes that you must extend based on the type of plug-in component being defined.

Note: You do not need to implement the plug-in panel interface to display a GUI component for the following plug-in components: functions and message types.

Table 4-6 Plug-In Panel Classes

To define the following plug-in . . .	The plug-in panel class must extend . . .	To define . . .
Any plug-in component	`com.bea.wlpi.common.plugin.PluginPanel`	GUI component to be displayed in the design client. When an action, Start or Event node, or variable type, is being defined, the plug-in panel class defined for it (defined later in this table) extend this class.
Action	`com.bea.wlpi.common.plugin.PluginActionPanel`	GUI component for the plug-in action. This class is used by the Action Plugin dialog box in the Studio, which provides generic support for subactions. **Note:** `PluginActionPanel` extends the `PluginPanel` class defined previously in this table.
Start and Event node	`com.bea.wlpi.common.plugin.PluginTriggerPanel`	Start and/or Event node GUI component to be displayed in the design client. This class is used by the Start Properties and Event Properties dialog boxes in the Studio. **Note:** `PluginTriggerPanel` extends the `PluginPanel` class defined previously in this table.

Table 4-6 Plug-In Panel Classes (Continued)

To define the following plug-in . . .	The plug-in panel class must extend . . .	To define . . .
Variable type	`com.bea.wlpi.common.plugin.PluginVariablePanel`	GUI component to be displayed in the design client so it is available to users for editing the plug-in variable type. This class is used by the Update Variable dialog box in the Studio. **Note:** `PluginVariablePanel` extends the `PluginPanel` class defined previously in this table.
	`com.bea.wlpi.common.plugin.PluginVariableRender`	GUI component to display the value of a plugin-defined variable type in the cell of a `javax.swing.JTable`. This class is used by the Update Variable dialog box in the Studio.

Each plug-in panel class is defined in more detail in the following sections.

Defining the PluginPanel Class

To define the plug-in GUI component displayed in the design client, you must define a class that extends the `com.bea.wlpi.common.plugin.PluginPanel` class.

Note: When defining actions, Start or Event nodes, or variable types, you should extend the corresponding plug-in panel class defined in the table “Plug-In Panel Classes” on page 4-24, which extends the `PluginPanel` class.

The following table describes the class methods defined by the `PluginPanel` class.

Note: You can override any method that is not declared as final.

Table 4-7 PluginPanel Class Methods

Method	Description
`public void exceptionHandlerRenamed(java.lang.String oldName, java.lang.String newName)`	Renames the event handler. The method must update any direct references to the exception handler, and propagate the information to any `com.bea.wlpi.evaluator.Expression` objects owned by the plug-in panel. Subclasses must override this method if they refer to workflow event handlers and propogate updates to those handlers to ensure that the reference is maintained. **Note:** In plug-in nodes, where actions are supported by default, the Plug-in Manager propagates the changes throughout the action lists. The method parameters are defined as follows: ■ *`oldName`*: `java.lang.String` object that specifies the old event handler name. ■ *`newName`*: `java.lang.String` object that specifies the new event handler name.
`public final com.bea.wlpi.common.plugin.PluginPanelContext getContext()`	Gets the parent component in which the plug-in panel is displayed. This method returns a `com.bea.wlpi.common.plugin.PluginPanelContext` object that specifies the parent component. For more information about implementing the `PluginPanelContext`, see "Using Plug-In Run-Time Contexts" on page 4-94.
`public final com.bea.wlpi.common.plugin.PluginData getData()`	Gets the plug-in data. This method returns a `com.bea.wlpi.common.plugin.PluginData` object that specifies the plug-in data. For more information about implementing the `PluginData` object, see "Implementing the PluginData Interface" on page 4-10.

Table 4-7 PluginPanel Class Methods (Continued)

Method	Description
`public java.lang.String getHelpIDString()`	Gets the help topic ID for the plug-in panel. This method returns a `java.lang.String` object that specifies the help topic ID.
`public java.lang.String getString(java.lang.String` *`key`*`)`	Gets a localized display string. The resource bundle name must have been set by a prior call to the `setResourceBundle()` method (described later in this table). The method parameter is defined as follows. *`key`*: `java.lang.String` object that specifies the resource key. This method returns a `java.lang.String` object that specifies the display string.
`public java.lang.String getString(java.lang.String` *`key`*`, java.lang.Object[]` *`args`*`)`	Gets a localized display string. The resource bundle name must have been set by a prior call to the `setResourceBundle()` method (described later in this table). This method uses the object's `ClassLoader` to retrieve the string resource from the nominated resource properties file in its `plugin-ejb.jar` file. The method parameters are defined as follows: ■ *`key`*: `java.lang.String` object that specifies the resource key. ■ *`args`*: `java.lang.Object` object that specifies the arguments to be inserted into the message text. This method returns a `java.lang.String` object that specifies the display string.

Table 4-7 PluginPanel Class Methods (Continued)

Method	Description
`public abstract void load()`	Instructs the plug-in panel to initialize its user interface using the plug-in data. This method calls `getData()` to access the plug-in data, sends the result to the corresponding plug-in class, and calls the appropriate get methods to retrieve the display values. The Plug-in Manager ensures that this method is called exactly once per modal display cycle. **Note:** Plug-ins must not call this method.
`public boolean referencesExceptionHandler(java.lang.String` *`handler`*`)`	Checks whether the plug-in panel references the specified event handler. The method must check by name any direct references that the plug-in panel class holds to the specified exception handler. Subclasses must override this method if they make reference to workflow event handlers to avoid inadvertently deleting a referenced event handler. **Note:** In plug-in nodes, where actions are supported by default, the Plug-in Manager propagates the changes throughout the action lists. The method parameter is defined as follows. *`handler`*: `java.lang.String` object that specifies the event handler name. The method returns a Boolean value: `true` if the plug-in panel references the specified event handler, and `false` if it does not.

Table 4-7 PluginPanel Class Methods (Continued)

Method	Description
`public boolean referencesVariable(java.lang.String` *`variable`*`)`	Checks whether the plug-in panel references the specified variable. The method must check by name any direct references that the plug-in panel class holds to the specified variable. Subclasses must override this method if they make reference to a workflow variable, either directly, by name, or indirectly, via expression, to avoid inadvertently deleting the referenced variables. **Note:** In plug-in nodes, where actions are supported by default, the Plug-in Manager propagates the changes throughout the action lists. The method parameter is defined as follows. *`variable`*: `java.lang.String` object that specifies the variable. The method returns a Boolean value: `true` if the plug-in panel references the specified eventhandler, and `false` if it does not.

Table 4-7 PluginPanel Class Methods (Continued)

Method	Description
`public final void setContext(com.bea.wlpi.common.plugin.PluginPanelContext` *`context`*`, com.bea.wlpi.common.plugin.PluginData` *`data`*`)`	Sets the operating context for the plug-in panel. The Plug-in Manager calls this method before adding the plug-in panel to the design client dialog box. This method stores the owner and data parameters in the corresponding member variables. **Note:** Plug-ins must not call this method. The method parameters are defined as follows: ■ *`context`*: `com.bea.wlpi.common.plugin.PluginPanelContext` object that specifies the design client dialog box context in which the plug-in panel is being displayed. For more information about implementing the `PluginPanelContext` object, see “Using Plug-In Run-Time Contexts” on page 4-94. ■ *`data`*: `com.bea.wlpi.common.plugin.PluginData` object that specifies the plug-in data. For more information about implementing the `PluginData` object, see “Implementing the PluginData Interface” on page 4-10.
`public void setResourceBundle(java.lang.String` *`bundleName`*`)`	Sets the resource bundle to use when localizing strings and messages. The method parameter is defined as follows. *`bundleName`*: `java.lang.String` object that specifies the name of the resource bundle.
`public abstract boolean validateAndSave()`	Instructs the plug-in panel to validate the GUI control values and then save them. This method calls `getData()` to access the plug-in data, sends the result to the corresponding plug-in class, and calls the appropriate set methods to save the display values. This method returns a Boolean value: `true` if the panel was validated and subsequently saved, and `false` if it was not.

Table 4-7 PluginPanel Class Methods (Continued)

Method	Description
`public void variableRenamed(java.lang.String` *`oldName`*`, java.lang.String` *`newName`*`)`	Renames the variables. The method must update any direct references to the variable, and propagate the information to any `com.bea.wlpi.evaluator.Expression` objects owned by the plug-in panel. This can be accomplished by calling the `variableRenamed()` method, followed by the `toString()` method to get the updated expression text. Subclasses must override this method if they make reference to workflow variables, either directly, by name, or indirectly, via expressions, to ensure that the reference is maintained. **Note:** In plug-in nodes, where actions are supported by default, the Plug-in Manager propagates the changes throughout the action lists. The method parameters are defined as follows: ■ *`oldName`*: `java.lang.String` object that specifies the old variable name. ■ *`newName`*: `java.lang.String` object that specifies the new variable name.

The following sections provide code examples showing how the `PluginPanel` class is defined.

Done Node Example

The following code listing shows how to define the `PluginPanel` class for a Done node. The code displays a decision dialog box (yes or no) within the Done Properties dialog box. Notable lines of code are shown in **bold**.

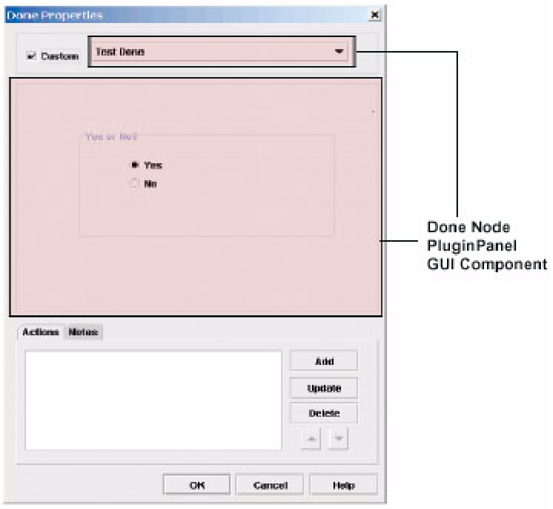
Note: This class is not available as part of the plug-in sample.

Listing 4-9 Defining the PluginPanel Class for a Done Node

```
package com.bea.wlpi.test.plugin;

import java.awt.*;
import javax.swing.*;
import javax.swing.border.TitledBorder;
import javax.swing.border.EtchedBorder;
import java.util.List;
import java.util.Locale;
import com.bea.wlpi.common.plugin.PluginPanel;
import com.bea.wlpi.common.plugin.PluginPanelContext;
import com.bea.wlpi.client.studio.Studio;
import com.bea.wlpi.common.VariableInfo;

public class DoneNodePanel extends PluginPanel
{

    JPanel ButtonPanel = new JPanel();
    ButtonGroup YesNoButtonGroup = new ButtonGroup();
    JRadioButton YesButton = new JRadioButton();
    JRadioButton NoButton = new JRadioButton();
    TitledBorder titledBorder = new TitledBorder(new EtchedBorder());

   public DoneNodePanel()
   {
      super(Locale.getDefault(), "jackolantern");
      setLayout(null);
      setBounds(12,12,420,300);
      setPreferredSize(new Dimension(420,300));
      ButtonPanel.setBorder(titledBorder);
      ButtonPanel.setLayout(null);
      add(ButtonPanel);
      ButtonPanel.setBounds(72,60,300,144);
      YesButton.setText("Yes");
      YesButton.setSelected(true);
      YesNoButtonGroup.add(YesButton);
      ButtonPanel.add(YesButton);
      YesButton.setBounds(60,36,46,23);
      NoButton.setText("No");
      YesNoButtonGroup.add(NoButton);
      ButtonPanel.add(NoButton);
      NoButton.setBounds(60,60,46,23);
      titledBorder.setTitle("Yes or No?");
   }

public void load() {
    DoneNodeData myData = (DoneNodeData)getData();
    if(myData != null) {
        if(myData.getYesOrNo() != null &&
myData.getYesOrNo().equals(TestPluginConstants.DONE_NO)) {
            NoButton.setSelected(true);
        } else {
            YesButton.setSelected(true);
        }
    }
}

   public boolean validateAndSave()
   {
    DoneNodeData myData = (DoneNodeData)getData();
    if(myData != null) {
        if(YesButton.isSelected()) {
            myData.setYesOrNo(TestPluginConstants.DONE_YES);
        } else {
            myData.setYesOrNo(TestPluginConstants.DONE_NO);
        }
    }

      return true;
   }
             }
```

The following figure illustrates the resulting `PluginPanel` GUI component.

Figure 4-1 PluginPanel GUI Component for a Done Node



Refer to the following related example listings:

- "Implementing the PluginObject Interface for a Done Node" on page 4-5 shows how to read the plug-in data in XML format.
- "Implementing the PluginData Interface for a Done Node" on page 4-13 shows how to read and save the plug-in data in XML format. This class extends the `PluginObject` class.
- "Defining the Run-Time Component Class for a Done Node" on page 4-73 shows how to define the execution information for the plug-in.

Workflow Template Properties Example

The following code listing shows how to define the `PluginPanel` class for workflow template properties. The code displays a decision dialog box (yes or no) within the Workflow Template Properties dialog box. Notable lines of code are shown in **bold**.

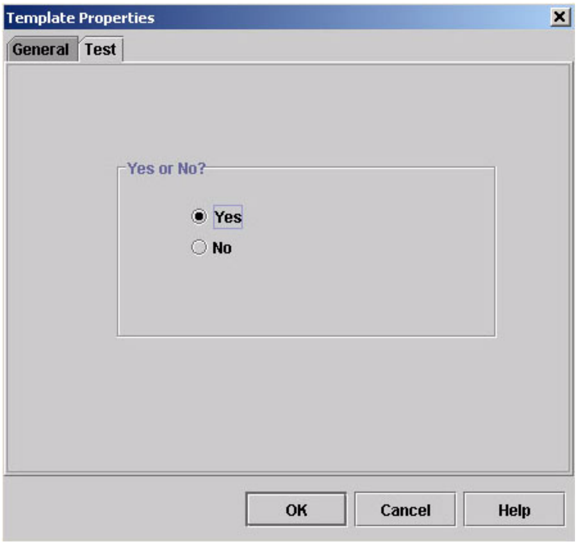
Note: This class is not available as part of the plug-in sample.

Listing 4-10 Defining the PluginPanel Class for Workflow Template Properties

```
package com.bea.wlpi.test.plugin;

import java.awt.*;
import javax.swing.*;
import javax.swing.border.TitledBorder;
import javax.swing.border.EtchedBorder;
import java.util.List;
import java.util.Locale;
import com.bea.wlpi.common.plugin.PluginPanel;
import com.bea.wlpi.common.plugin.PluginPanelContext;
import com.bea.wlpi.client.studio.Studio;
import com.bea.wlpi.common.VariableInfo;

public class TemplatePropertiesPanel extends PluginPanel
{

    JPanel ButtonPanel = new JPanel();
    ButtonGroup YesNoButtonGroup = new ButtonGroup();
    JRadioButton YesButton = new JRadioButton();
    JRadioButton NoButton = new JRadioButton();
    TitledBorder titledBorder = new TitledBorder(new EtchedBorder());

   public TemplatePropertiesPanel()
   {
      super(Locale.getDefault(), "stpatty");
      setLayout(null);
      setBounds(12,12,420,300);
      ButtonPanel.setBorder(titledBorder);
      ButtonPanel.setLayout(null);
      add(ButtonPanel);
      ButtonPanel.setBounds(72,60,300,144);
      YesButton.setText("Yes");
      YesButton.setSelected(true);
      YesNoButtonGroup.add(YesButton);
      ButtonPanel.add(YesButton);
      YesButton.setBounds(60,36,46,23);
      NoButton.setText("No");
      YesNoButtonGroup.add(NoButton);
      ButtonPanel.add(NoButton);
      NoButton.setBounds(60,60,46,23);
      titledBorder.setTitle("Yes or No?");
   }

public void load() {

    TemplatePropertiesData myData = (TemplatePropertiesData)getData();
    if(myData != null) {
        if(myData.getYesOrNo() != null &&
myData.getYesOrNo().equals(TestPluginConstants.DONE_NO)) {
            NoButton.setSelected(true);
        } else {
            YesButton.setSelected(true);
        }
    }
}

   public boolean validateAndSave()
   {
    TemplatePropertiesData myData = (TemplatePropertiesData)getData();
    if(myData != null) {
        if(YesButton.isSelected()) {
            myData.setYesOrNo(TestPluginConstants.DONE_YES);
        } else {
            myData.setYesOrNo(TestPluginConstants.DONE_NO);
        }
    }

      return true;
   }
```

The following figure illustrates the resulting `PluginPanel` GUI component.

Figure 4-2 PluginPanel GUI Component for Workflow Template Properties



Refer to the following related example listings:

- "Implementing the PluginObject Interface for a Done Node" on page 4-5 shows how to define the `PluginObject` class to read the plug-in data in XML format.
- "Implementing the PluginData Interface for Workflow Template Properties" on page 4-18 shows how to read and save the plug-in data in XML format. This class extends the `PluginObject` class.

Workflow Template Definition Properties Example

The following code listing shows how to define the `PluginPanel` class for workflow template definition properties. The code displays a decision dialog box (yes or no) within the Workflow Template Definition Properties dialog box. Notable lines of code are shown in **`bold`**.

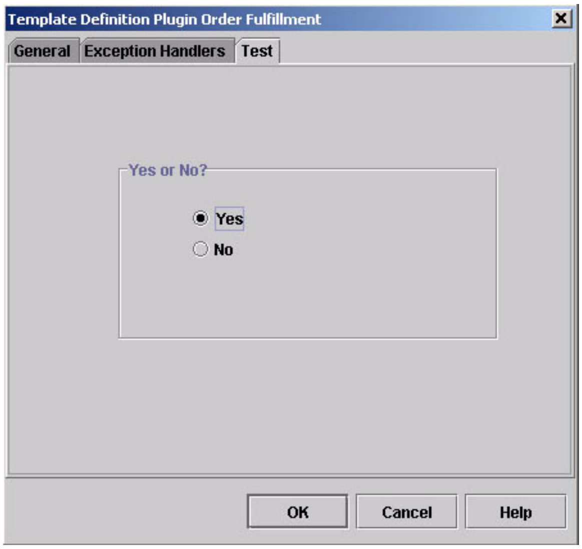
Note: This class is not available as part of the plug-in sample.

Listing 4-11 Defining the PluginPanel Class for Workflow Template Definition Properties

```
package com.bea.wlpi.test.plugin;

import java.awt.*;
import javax.swing.*;
import javax.swing.border.TitledBorder;
import javax.swing.border.EtchedBorder;
import java.util.List;
import java.util.Locale;
import com.bea.wlpi.common.plugin.PluginPanel;
import com.bea.wlpi.common.plugin.PluginPanelContext;
import com.bea.wlpi.client.studio.Studio;
import com.bea.wlpi.common.VariableInfo;

public class TemplateDefinitionPropertiesPanel extends PluginPanel
{

    JPanel ButtonPanel = new JPanel();
    ButtonGroup YesNoButtonGroup = new ButtonGroup();
    JRadioButton YesButton = new JRadioButton();
    JRadioButton NoButton = new JRadioButton();
    TitledBorder titledBorder = new TitledBorder(new EtchedBorder());

   public TemplateDefinitionPropertiesPanel()
   {
      super(Locale.getDefault(), "valentine");
      setLayout(null);
      setBounds(12,12,420,300);
      ButtonPanel.setBorder(titledBorder);
      ButtonPanel.setLayout(null);
      add(ButtonPanel);
      ButtonPanel.setBounds(72,60,300,144);
      YesButton.setText("Yes");
      YesButton.setSelected(true);
      YesNoButtonGroup.add(YesButton);
      ButtonPanel.add(YesButton);
      YesButton.setBounds(60,36,46,23);
      NoButton.setText("No");
      YesNoButtonGroup.add(NoButton);
      ButtonPanel.add(NoButton);
      NoButton.setBounds(60,60,46,23);
      titledBorder.setTitle("Yes or No?");
   }

public void load() {
    TemplateDefinitionPropertiesData myData =
(TemplateDefinitionPropertiesData)getData();
    if(myData != null) {
        if(myData.getYesOrNo() != null &&
myData.getYesOrNo().equals(TestPluginConstants.DONE_NO)) {
            NoButton.setSelected(true);
        } else {
            YesButton.setSelected(true);
        }
    }
}

   public boolean validateAndSave()
   {
    TemplateDefinitionPropertiesData myData =
(TemplateDefinitionPropertiesData)getData();
    if(myData != null) {
        if(YesButton.isSelected()) {
            myData.setYesOrNo(TestPluginConstants.DONE_YES);
        } else {
            myData.setYesOrNo(TestPluginConstants.DONE_NO);
        }
    }

      return true;
}
```

The following figure illustrates the resulting `PluginPanel` GUI component.

Figure 4-3 PluginPanel GUI Component for Workflow Template Definition Properties



Refer to the following related example listings:

- "Implementing the PluginObject Interface for a Done Node" on page 4-5 shows how to define the `PluginObject` class to read the plug-in data in XML format.
- "Implementing the PluginData Interface for Workflow Template Definition Properties" on page 4-19 shows how to read and save the plug-in data in XML format. This class extends the `PluginObject` class.

Defining the PluginActionPanel Class

To define the GUI component displayed in the design client when defining a plug-in action, you must define a class that extends the `com.bea.wlpi.common.plugin.PluginActionPanel` class. In the Studio, the `PluginActionPanel` class is used by the Action Plugin dialog box, which provides generic support for subactions.

The `PluginActionPanel` class defines no additional methods.

Note: The `PluginActionPanel` class extends the `PluginPanel` class. For more information about the `PluginPanel` class methods, see the table "PluginPanel Class Methods" on page 4-26.

The following code listing is an excerpt from the plug-in sample that shows how to define the `PluginActionPanel` class. This excerpt is taken from the `CheckInventoryActionPanel.java` file in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory. Notable lines of code are shown in **bold**.

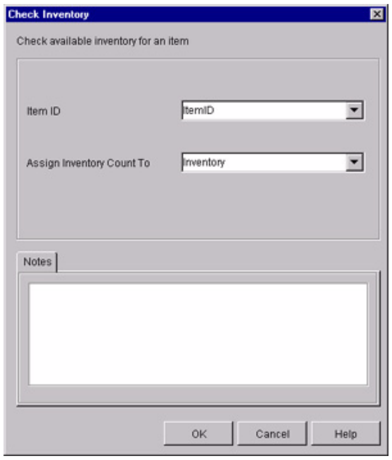
Note: Refer to the `SendConfirmationActionPanel.java` file in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory for another example of how to define a `PluginActionPanel` class.

Listing 4-12 Defining the PluginActionPanel Class

```
package com.bea.wlpi.tour.po.plugin;

import java.awt.*;
import javax.swing.*;
import java.util.List;
import java.util.Locale;
import com.bea.wlpi.common.VariableInfo;
import com.bea.wlpi.common.plugin.PluginActionPanel;
import com.bea.wlpi.common.plugin.PluginPanelContext;

public class CheckInventoryActionPanel extends PluginActionPanel {
    private JLabel inputLabel = new JLabel();
    private JLabel outputLabel = new JLabel();
    private JComboBox inputComboBox = new JComboBox();
    private JComboBox outputComboBox = new JComboBox();
    private List variables = null;

    public CheckInventoryActionPanel() {
        this(Locale.getDefault());
    }

    public CheckInventoryActionPanel(Locale lc) {

        super(lc, "checkinventory");

        setLayout(null);
        setBounds(12, 12, 420, 210);
        setPreferredSize(new Dimension(420, 210));
        add(inputLabel);
        inputLabel.setBounds(12, 48, 96, 24);
        add(outputLabel);
        outputLabel.setBounds(12, 108, 166, 24);
        add(inputComboBox);
        inputComboBox.setBounds(190, 48, 212, 24);
        inputComboBox.setEditable(true);
        add(outputComboBox);
        outputComboBox.setBounds(190, 108, 212, 24);
        outputComboBox.setEditable(true);
    }

    public void load() {

        setResourceBundle("com.bea.wlpi.tour.po.plugin.SamplePlugin");
        inputLabel.setText(getString("inputLabel"));
        outputLabel.setText(getString("outputLabel"));

        CheckInventoryActionData myData = (CheckInventoryActionData)getData();

        variables = getContext().getVariableList(VariableInfo.TYPE_INT);

        // load is called before displaying this panel each time.  Make sure to
        // remove items from the combo box before filling with currently
        // defined variables.
        inputComboBox.removeAllItems();

        String inputVar = myData.getInputVariableName();
        int n = variables == null ? 0 : variables.size();

        for (int i = 0; i < n; i++) {
            VariableInfo varInfo = (VariableInfo)variables.get(i);

            inputComboBox.addItem(varInfo.getName());

            if (inputVar != null && inputVar.equals(varInfo.getName())) {
                inputComboBox.setSelectedIndex(i);
            }
        }

        if (inputVar == null && n > 0)
            inputComboBox.setSelectedIndex(0);

        outputComboBox.removeAllItems();

        String outputVar = myData.getOutputVariableName();

        for (int i = 0; i < n; i++) {
            VariableInfo varInfo = (VariableInfo)variables.get(i);

            outputComboBox.addItem(varInfo.getName());

            if (outputVar != null && outputVar.equals(varInfo.getName())) {
                outputComboBox.setSelectedIndex(i);
            }
        }

        if (outputVar == null && n > 0)
            outputComboBox.setSelectedIndex(0);
    }

    public boolean validateAndSave() {

        CheckInventoryActionData myData = (CheckInventoryActionData)getData();
        String input = (String)inputComboBox.getEditor().getItem();

        try {
            VariableInfo varInfo = getContext().checkVariable(input,
                                           new String[]{ VariableInfo.TYPE_INT });

            if (varInfo == null)
                return false;

            if (!(varInfo.getType().equals(VariableInfo.TYPE_INT))) {

JOptionPane.showMessageDialog(SwingUtilities.windowForComponent(this),
                                             getString("Message_100"),
                                             getString("variableErrorTitle"),
                                             JOptionPane.ERROR_MESSAGE);

                return false;
            }

            input = varInfo.getName();
        } catch (Exception e) {
          JOptionPane.showMessageDialog(SwingUtilities.windowForComponent(this),
                                         e.getLocalizedMessage(),
                                         getString("variableErrorTitle"),
                                         JOptionPane.ERROR_MESSAGE);

            return false;
        }

        String output = (String)outputComboBox.getEditor().getItem();

        try {
            VariableInfo varInfo = getContext().checkVariable(output,
                                       new String[]{ VariableInfo.TYPE_INT });

            if (varInfo == null)
                return false;

            if (!(varInfo.getType().equals(VariableInfo.TYPE_INT))) {

JOptionPane.showMessageDialog(SwingUtilities.windowForComponent(this),
                                              getString("Message_100"),
                                              getString("variableErrorTitle"),
                                              JOptionPane.ERROR_MESSAGE);

                return false;
            }

            output = varInfo.getName();
        } catch (Exception e) {
          JOptionPane.showMessageDialog(SwingUtilities.windowForComponent(this),
                                          e.getLocalizedMessage(),
                                          getString("variableErrorTitle"),
                                          JOptionPane.ERROR_MESSAGE);

            return false;
        }

        if (input == null || output == null) {
            JOptionPane.showMessageDialog(null, getString("Message_101"),
                                          getString("invalidDataTitle"),
                                          JOptionPane.ERROR_MESSAGE);

            return false;
        }

        myData.setInputVariableName(input);
        myData.setOutputVariableName(output);

        return true;
    }
}
```

The following figure illustrates the resulting `PluginActionPanel` GUI component.

Figure 4-4   PluginActionPanel GUI Component


Check Inventory
Check available inventory for an item
Item ID
ItemID
Assign Inventory Count To
Inventory
Notes
OK
Cancel
Help

Refer to the following related example listings:

- `CheckInventoryActionObject.java` in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory shows how to read the plug-in data in XML format.
- "Implementing the PluginActionData Interface" on page 4-21 shows how to read and save the plug-in data in XML format. This class extends the `CheckInventoryActionObject` class.
- "Defining the Run-Time Component Class for an Action" on page 4-61 shows how to define the execution information for the plug-in.
- "Customizing an Action Tree" on page 4-70 shows how to customize the actions and/or action categories listed in the action trees that are displayed in various dialog boxes within the Studio.

For more information about the plug-in sample, see "BPM Plug-In Sample" on page 10-1.

Defining the PluginTriggerPanel Class

To define the GUI component to be displayed in the design client when defining a plug-in Start or Event node, you must defne a class that extends the `com.bea.wlpi.common.plugin.PluginTriggerPanel` class. In the Studio, the Start and Event node `PluginTriggerPanel` classes are used by the Start Properties and Event Properties dialog box, respectively.

The following table describes the class methods that are defined by the `PluginTriggerPanel` class.

Note: The `PluginTriggerPanel` class extends the `PluginPanel` class. For more information about the `PluginPanel` class methods, see the table "PluginPanel Class Methods" on page 4-26.

Table 4-8 PluginTriggerPanel Class Methods

Method	Description
`public java.lang.String getEventDescriptor()`	Gets a string that characterizes the plug-in event descriptor. The event descriptor defines the data for which the plug-in node is watching. The expression evaluator passes the event descriptor to any instances of the associated `com.bea.wlpi.common.plugin.PluginField` implementation class specified by the `com.bea.wlpi.common.plugin.FieldInfo` object. The `FieldInfo` object is supplied by the parent `com.bea.wlpi.common.plugin.StartInfo` or `com.bea.wlpi.common.plugin.EventInfo` object. The method returns a `java.lang.String` object that specifies the event descriptor. For the default implementation this method returns null. For information about defining the plug-in field to access the plug-in-specific external event, see “Defining the Run-Time Component Class for a Message Type” on page 4-84.

Table 4-8 PluginTriggerPanel Class Methods (Continued)

Method	Description
`public java.lang.String[] getFields()`	Gets a list of field names associated with the event (if known). If this list is not null, the associated Start Properties or Event Properties dialog box passes the list to the Expression Builder. The method returns an array of `java.lang.String` objects that specify the field names. For the default implementation this method returns null. For information about defining the plug-in field to access a plug-in-specific external event, see "Defining the Run-Time Component Class for a Message Type" on page 4-84.

The following sections provide code examples showing how the `PluginTriggerPanel` class is defined.

Start Node Example

The following code listing is an excerpt from the plug-in sample that shows how to define the `PluginTriggerPanel` class for a Start node. This excerpt is taken from the `StartNodePanel.java` file in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory. Notable lines of code are shown in **bold**.

Listing 4-13 Defining the PluginTriggerPanel Class for a Start Node

```
package com.bea.wlpi.tour.po.plugin;

import java.awt.*;
import javax.swing.*;
import javax.swing.border.TitledBorder;
import javax.swing.border.EtchedBorder;
import java.util.List;
import java.util.Locale;
import com.bea.wlpi.common.plugin.PluginTriggerPanel;
import com.bea.wlpi.common.plugin.PluginPanelContext;
import com.bea.wlpi.common.VariableInfo;

public class StartNodePanel extends PluginTriggerPanel {
    private JLabel StartOrderLabel = new JLabel();
    private JTextArea StartOrderText = new JTextArea();

    public StartNodePanel() {
        this(Locale.getDefault());
    }

    public StartNodePanel(Locale lc) {

        super(lc, "startorder");

        setLayout(null);
        setBounds(12, 12, 420, 240);
        setPreferredSize(new Dimension(420, 240));
        add(StartOrderLabel);
        StartOrderLabel.setFont(new Font("Dialog", Font.BOLD, 16));
        StartOrderLabel.setBounds(120, 12, 156, 24);
        StartOrderText.setLineWrap(true);
        StartOrderText.setWrapStyleWord(true);
        StartOrderText.setEditable(false);
        add(StartOrderText);
        StartOrderText.setBounds(30, 48, 348, 144);
    }

    public void load() {

        setResourceBundle("com.bea.wlpi.tour.po.plugin.SamplePlugin");
        StartOrderLabel.setText(getString("startOrderLabel"));
        StartOrderText.setText(getString("startOrderText"));
    }

    public boolean validateAndSave() {
        return true;
    }

    public String[] getFields() {
        return SamplePluginConstants.ORDER_FIELDS;
    }

    public String getEventDescriptor() {
        return SamplePluginConstants.START_ORDER_EVENT;
    }
}
```

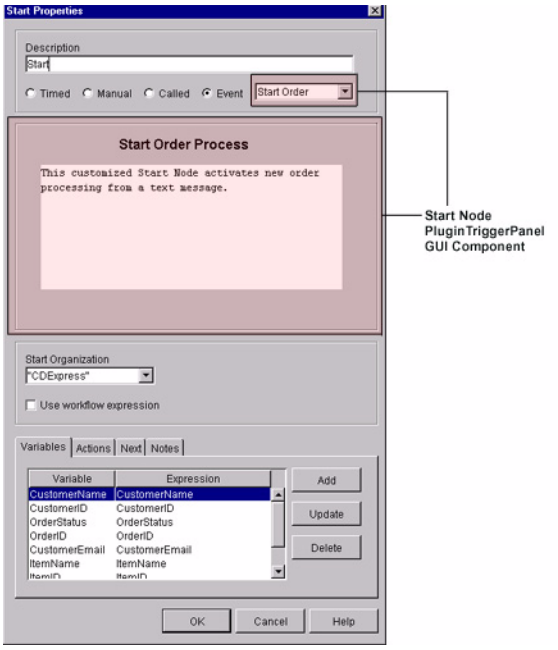
The `START_ORDER_EVENT` and `ORDER_FIELDS` field element values are included within the `SamplePluginConstants.java` class file. They define the plug-in Start node event descriptor and field element values as follows:

```
final static String START_ORDER_EVENT = "startOrder";
final static String[] ORDER_FIELDS = {
       "CustomerName", "CustomerID", "OrderStatus", "OrderID",
       "CustomerEmail", "ItemName", "ItemID", "ItemQuantity",
       "CustomerState"
};
```

For more information about defining the plug-in field to access a plug-in-specific external event, see “Defining the Run-Time Component Class for a Message Type” on page 4-84.

The following figure illustrates the resulting `PluginTriggerPanel` GUI component.

Figure 4-5 PluginTriggerPanel GUI Component for a Start Node



Description: Start

Timed Manual Called [✓ Event] Start Order

Start Order Process

This customized Start Node activates new order processing from a text message.

Start Organization: "CDExpress"

Use workflow expression

Variable	Expression
CustomerName	CustomerName
CustomerID	CustomerID
OrderStatus	OrderStatus
OrderID	OrderID
CustomerEmail	CustomerEmail
ItemName	ItemName
ItemID	ItemID

Refer to the following related example listings:

- "Implementing the PluginObject Interface for a Start Node" on page 4-8 shows how to read the plug-in data in XML format.
- "Implementing the PluginData Interface for a Start Node" on page 4-16 shows how to read and save the plug-in data in XML format.
- "Defining the Run-Time Component Class for a Start Node" on page 4-91 shows how to define the execution information for the plug-in.
- "Using Plug-In Run-Time Contexts" on page 4-94 shows how to define the plug-in fields that can be referenced from an evaluator expression.

For more information about the plug-in sample, see "BPM Plug-In Sample" on page 10-1.

Event Node Example

The following code listing is an excerpt from the plug-in sample that shows how to define the `PluginTriggerPanel` class for an Event node. This excerpt is taken from the `EventNodePanel.java` file in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory. Notable lines of code are shown in **bold**.

Listing 4-14 Defining the PluginTriggerPanel Class for an Event Node

```
package com.bea.wlpi.tour.po.plugin;

import java.awt.*;
import javax.swing.*;
import java.util.Locale;
import com.bea.wlpi.common.plugin.PluginTriggerPanel;
import com.bea.wlpi.common.plugin.PluginPanelContext;
import com.bea.wlpi.common.VariableInfo;

public class EventNodePanel extends PluginTriggerPanel {
    private JLabel confirmOrderLabel = new JLabel();
    private JTextArea confirmOrderText = new JTextArea();

    /**
     * Create a new EventNodePanel.
     */
    public EventNodePanel() {
        this(Locale.getDefault());
    }

    public EventNodePanel(Locale lc) {

        super(lc, "confirmevent");

        setLayout(null);
        setBounds(12, 12, 420, 240);
        setPreferredSize(new Dimension(420, 240));
        add(confirmOrderLabel);
        confirmOrderLabel.setFont(new Font("Dialog", Font.BOLD, 16));
        confirmOrderLabel.setBounds(144, 12, 120, 24);
        confirmOrderText.setRequestFocusEnabled(false);
        confirmOrderText.setLineWrap(true);
        confirmOrderText.setWrapStyleWord(true);
        confirmOrderText.setEditable(false);
        add(confirmOrderText);
        confirmOrderText.setBounds(30, 48, 348, 144);
    }

    public void load() {

        setResourceBundle("com.bea.wlpi.tour.po.plugin.SamplePlugin");
        confirmOrderLabel.setText(getString("confirmOrderLabel"));
        confirmOrderText.setText(getString("confirmOrderText"));
    }

    public boolean validateAndSave() {

        // There are no UI controls on this panel which accept user input.
        // Therefore, there is nothing to do in this method.
        return true;
    }

    public String[] getFields() {
        return SamplePluginConstants.CONFIRM_FIELDS;
    }

    public String getEventDescriptor() {
        return SamplePluginConstants.CONFIRM_EVENT;
    }
}
```

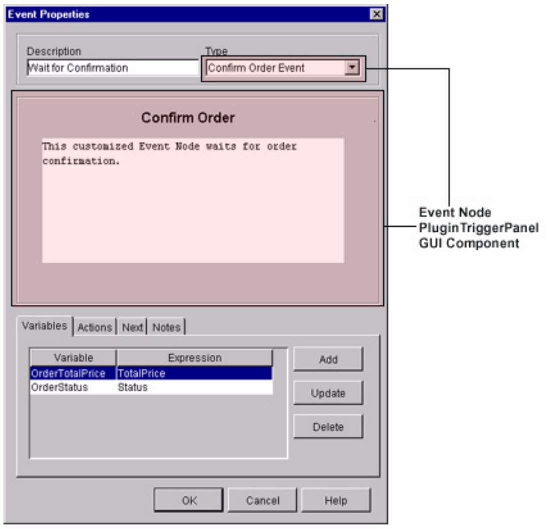
The `CONFIRM_EVENT` and `CONFIRM_FIELD` are included within the `SamplePluginConstants.java` class. They define the plug-in Event node event descriptor and field element values as follows:

```
final static String CONFIRM_EVENT = "confirmOrder";
final static String[] CONFIRM_FIELDS = { "Status", "TotalPrice" };
```

For more information about defining the plug-in field to access a plug-in-specific external event, see "Defining the Run-Time Component Class for a Message Type" on page 4-84.

The following figure illustrates the resulting `PluginTriggerPanel` GUI component.

Figure 4-6 PluginTriggerPanel GUI Component for a Event Node



Description: Wait for Confirmation

Type: Confirm Order Event

This customized Event Node waits for order confirmation.

Variable	Expression
OrderTotalPrice	TotalPrice
OrderStatus	Status

Refer to the following related example listings:

- `EventObject.java` in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory shows how to implement the `PluginObject` interface to read plug-in data.
- "Implementing the PluginData Interface for an Event Node" on page 4-15 shows how to read and save the plug-in data. This class extends the `EventObject` class.

- "Defining the Run-Time Component Class for an Event Node" on page 4-78 shows how to define the execution information for the plug-in.

For more information about the plug-in sample, see "BPM Plug-In Sample" on page 10-1.

Defining the PluginVariablePanel Class

To define the GUI component displayed in the design client when defining a plug-in variable that enables users to edit a plug-in variable type, you must define a class that extends the `com.bea.wlpi.common.plugin.PluginVariablePanel` class. In the Studio, the `PluginVariablePanel` class is used by the Update Variable dialog box.

The following table describes the class methods that are defined by the `PluginVariablePanel` class.

Note: The `PluginVariablePanel` class extends the `PluginPanel` class. For more information about the `PluginPanel` class methods, see the table "PluginPanel Class Methods" on page 4-26.

Table 4-9 PluginVariablePanel Class Methods

Method	Description
`public final java.lang.Object getVariableValue()`	Gets the value of the plug-in variable. The method returns a `java.lang.Object` object that specifies the variable value.

Table 4-9 PluginVariablePanel Class Methods (Continued)

Method	Description
`public final void setContext(java.lang.Object` *`variableValue`*`)`	Sets the operating context for the plug-in panel. The Plug-in Manager calls this method before adding the plug-in variable panel to the Update Variable dialog box. **Note:** Plug-ins must not call this method. The method parameter is defined as follows. *`variableValue`*: `java.lang.Object` object that specifies the variable value. You can obtain the variable value using the `getVariableValue()` method described previously in this table.

The following code listing shows how to define the `PluginVariablePanel` class. Notable lines of code are shown in **bold**.

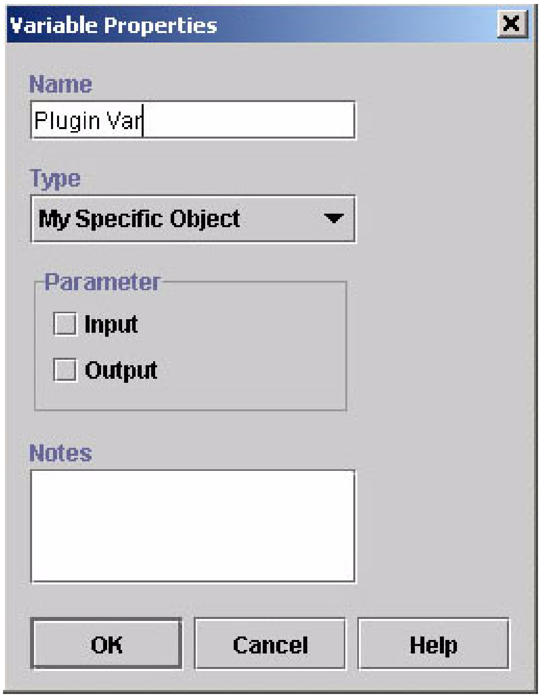
Note: This class is not available as part of the plug-in sample.

Listing 4-15 Defining the PluginVariablePanel Class

```
package com.bea.wlpi.test.plugin;

import java.awt.*;
import javax.swing.*;
import javax.swing.border.TitledBorder;
import javax.swing.border.EtchedBorder;
import java.util.List;
import java.util.Locale;
import com.bea.wlpi.client.studio.Studio;
import com.bea.wlpi.common.VariableInfo;
import com.bea.wlpi.common.plugin.PluginVariablePanel;

public class VariablePanel extends PluginVariablePanel {
    JTextField highField, lowField;
    public VariablePanel() {
        super(Locale.getDefault(), "augustus");
        setLayout(null);
        setBounds(12,12,420,60);
        highField = new JTextField();
        highField.setLocation(20, 10);
        highField.setSize(300, 20);
        add(highField);
        lowField = new JTextField();
        lowField.setLocation(20, 40);
        lowField.setSize(300, 20);
        add(lowField);
    }

    public void load() {
        if (variableValue != null) {
            highField.setText(((MySpecificObject)variableValue).getHigh());
            lowField.setText(((MySpecificObject)variableValue).getLow());
        } else {
            highField.setText("");
            lowField.setText("");
        }
    }

    public boolean validateAndSave() {
        try {
            variableValue = new MySpecificObject(lowField.getText(),
highField.getText());
        } catch (Exception e) {
            return false;
        }
        return true;
    }
}
```

The following figure illustrates the resulting `PluginVariablePanel` GUI component.

Figure 4-7 PluginVariablePanel GUI Component



Refer to the related example listing, "Defining the PluginVariableRenderer Class" on page 4-58, which shows how to display the value of a plugin-defined variable type in the cell of a `javax.swing.JTable`.

Defining the PluginVariableRenderer Class

To display the value of a plugin-defined variable type in the cell of a `javax.swing.JTable`, implement the `com.bea.wlpi.common.plugin.PluginVariableRenderer` interface.

Note: Classes implementing this interface must be subclasses of `java.awt.Component`.

The following table describes the `PluginVariableRenderer` interface method that you must implement.

Table 4-10 PluginVariableRenderer Interface Method

Method	Description
`public void setValue(java.lang.Object value)`	Sets the variable to be displayed. The method parameter is defined as follows. *`value`*: `java.lang.Object` object that specifies the variable value to be displayed. This value can be either null or an instance of the class declared in the corresponding `com.bea.wlpi.common.plugin.VariableTypeInfo` object.

The following code listing shows how to display the value of a plugin-defined variable type in the cell of a `javax.swing.JTable`. Notable lines of code are shown in **bold**.

Note: This class is not available as part of the plug-in sample.

Listing 4-16 Defining the PluginVariableRenderer Class

```
package com.bea.wlpi.test.plugin;

import java.io.Serializable;
import javax.swing.JLabel;
import com.bea.wlpi.common.plugin.PluginVariableRenderer;

public class VariableRenderer extends JLabel implements PluginVariableRenderer,
Serializable {
    public VariableRenderer() {
    }

    public void setValue(Object value) {
        if (value == null)
            setText("null");
        else
            setText(value.toString());
    }
}
```

Refer to "Defining the PluginVariablePanel Class" on page 4-55, which shows how to display the plug-in GUI component in the design client.

Executing the Plug-In

To execute the plug-in, you must define the run-time component class for the plug-in.

The following table describes the plug-in component interfaces that you must implement based on the type of plug-in component being created. To enable the plug-in to read (parse) the incoming data, the run-time component class must implement the `load()` (parsing) method of its parent interface, `com.bea.wlpi.common.plugin.PluginObject`.

Note: The following two plug-in components do not need to define execution information: variable types, workflow template properties, or workflow template definition properties.

Table 4-11 Plug-In Run-Time Component Interfaces

To define the following plug-in . . .	You must implement . . .	To define . . .
Action	`com.bea.wlpi.server.plugin.PluginAction`	Plug-in action execution information. **Note:** To support plug-in actions, you must also customize the actions and/or action categories listed in the action trees that are displayed in various dialog boxes within the Studio.

Table 4-11 Plug-In Run-Time Component Interfaces (Continued)

To define the following plug-in . . .	You must implement . . .	To define . . .
Done node	`com.bea.wlpi.server.plugin.PluginDone`	Plug-in Done node execution information. **Note:** The `PluginDone` interface extends the `com.bea.wlpi.server.plugin.PluginTemplateNode` interface. For more information, see "PluginTemplateNode Interface" on page 4-92.
Event node	`com.bea.wlpi.server.plugin.PluginEvent`	Plug-in event node execution information.
Function	`com.bea.wlpi.common.plugin.PluginFunction`	New evaluator function information.
Message type	`com.bea.wlpi.server.plugin.PluginField`	Plug-in-specific message types.
Start node	`com.bea.wlpi.server.plugin.PluginStart2`	Plug-in Start node execution information. **Note:** The `PluginStart2` interface extends the `com.bea.wlpi.server.plugin.PluginTemplateNode` interface. For more information, see "PluginTemplateNode Interface" on page 4-92.

Note: At run time you can use context interfaces that are passed by the Plug-in Manager to access the run-time context and services for the associated plug-in. For information about the context interfaces, see "Using Plug-In Run-Time Contexts" on page 4-94.

The following sections explain in detail how to define each of the plug-in run-time component classes.

Defining the Run-Time Component Class for an Action

To define the run-time component class for a plug-in action, you must:

- Implement the `com.bea.wlpi.server.plugin.PluginAction` interface to define the plug-in action execution information.
- Customize the actions and/or action categories listed in the action trees that are displayed in various dialog boxes within the Studio.

Defining the Execution Information for a Plug-In Action

To define the execution information for a plug-in action, you must implement the `com.bea.wlpi.server.plugin.PluginAction` interface and its methods, as described in the following table.

Table 4-12 PluginAction Interface Methods

Method	Description
`public int execute(com.bea.wlpi.server.plugin.ActionContext` *`actionContext`*`, com.bea.wlpi.server.common.ExecutionContext` *`execContext`*`) throws com.bea.wlpi.common.WorkflowException`	Executes the plug-in action and its business logic. The method parameters are defined as follows: ■ *`actionContext`*: `com.bea.wlpi.server.plugin.ActionContext` object that specifies the action context. The action context provides action-level services, such as the execution of subactions and subworkflow instantiation. ■ *`execContext`*: `com.bea.wlpi.server.common.ExecutionContext` object that specifies the execution context. The execution context provides access to the run-time context, including the template ID, template definition ID, workflow instance ID, event data, and various services related to the workflow execution. For more information about the action and execution contexts, see "Using Plug-In Run-Time Contexts" on page 4-94. This method returns one of the following `com.bea.wlpi.server.common.ExecutionContext` integer values, indicating the return code, that specifies whether or not processing should continue: ■ `EXIT_CONTINUE`: Exit an error handler and permit processing of subsequent operations. ■ `EXIT_RETRY`: Exit an error handler and request the retry of the failed operation. ■ `EXIT_ROLLBACK`: Exit an error handler and request the rollback of the user transaction. ■ `STOP`: Stop the plug-in execution.

Table 4-12 PluginAction Interface Methods (Continued)

Method	Description
`public void fixup(com.bea.wlpi.evaluator.ExpressionParser` *`parser`*`) throws com.bea.wlpi.common.WorkflowException`	Enables plug-in actions to compile the necessary expressions. The Plug-in Manager calls this method after parsing the template definition and storing it in memory, and prior to starting the workflow. The method parameter is defined as follows. *`parser`*: `com.bea.wlpi.evaluator.ExpressionParser` object that specifies the expression parser.

Table 4-12 PluginAction Interface Methods (Continued)

Method	Description
`public void response(com.bea.wlpi.server.plugin.ActionContext` *`actionContext`*`, com.bea.wlpi.server.common.ExecutionContext` *`execContext`*`, java.lang.Object` *`data`*`) throws com.bea.wlpi.common.WorkflowException`	Processes an asynchronous response directed to this action. Typically, this method is returned as a response to an external request that the action raised in a prior call to its `execute()` method (described earlier in this table). Plug-in actions can use this method to initiate, for example, the asynchronous execution of a subaction list. The Plug-in Manager calls this method when it receives a call to the `com.bea.wlpi.server.worklist.Worklist.response()` method or another overload method. The method parameters are defined as follows: ■ *`actionContext`*: `com.bea.wlpi.server.plugin.ActionContext` object that specifies the action context. The action context provides action-level services such as the execution of subactions and subworkflow instantiation. ■ *`execContext`*: `com.bea.wlpi.server.common.ExecutionContext` object that specifies the execution context. The execution context provides access to the run-time context, including the template ID, template definition ID, workflow instance ID, event data, and various services related to the workflow execution. ■ *`data`*: `java.lang.Object` object that specifies the data object that the plug-in casts to a known type in order to extract the required information. For more information about the action and execution contexts, see “Using Plug-In Run-Time Contexts” on page 4-94.

Table 4-12 PluginAction Interface Methods (Continued)

Method	Description
`public void startedWorkflowDone(com.bea.wlpi.server.plugin.ActionContext` *`actionContext`*`, com.bea.wlpi.server.common.ExecutionContext` *`execContext`*`, com.bea.wlpi.common.VariableInfo[]` *`output`*`) throws com.bea.wlpi.common.WorkflowException`	Notifies the plug-in action that a subworkflow that it had previously started is now complete. The method parameters are defined as follows: ■ *`actionContext`*: `com.bea.wlpi.server.plugin.ActionContext` object that specifies the action context. The action context provides action-level services, such as the execution of subactions and subworkflow instantiation. ■ *`execContext`*: `com.bea.wlpi.common.common.ExecutionContext` object that specifies the execution context. The execution context provides access to the run-time context, including the template ID, template definition ID, workflow instance ID, event data, and various services related to the workflow execution. ■ *`output`*: `com.bea.wlpi.common.VariableInfo` array that specifies the output variables of the called workflow. For more information about the action and execution contexts, see “Using Plug-In Run-Time Contexts” on page 4-94.

The following code listing is an excerpt from the plug-in sample that shows how to define the run-time component class for an action. This excerpt is taken from the `CheckInventoryAction.java` file in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory. Notable lines of code are shown in **bold**.

Note: Refer to the `SendConfirmationAction.java` file in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory for another example of how to define the plug-in action run-time component class.

Listing 4-17 Defining the Run-Time Component Class for an Action

```
package com.bea.wlpi.tour.po.plugin;

import java.io.IOException;
import com.bea.wlpi.server.plugin.PluginAction;
import com.bea.wlpi.common.WorkflowException;
import com.bea.wlpi.common.plugin.PluginException;
import com.bea.wlpi.common.Messages;
import com.bea.wlpi.common.VariableInfo;
import com.bea.wlpi.evaluator.Expression;
import com.bea.wlpi.evaluator.EvaluatorException;
import com.bea.wlpi.server.common.ExecutionContext;
import com.bea.wlpi.evaluator.ExpressionParser;
import com.bea.wlpi.server.plugin.ActionContext;
import org.xml.sax.*;

public class CheckInventoryAction extends CheckInventoryActionObject
        implements PluginAction {
    private Expression inputValueExpression;
    static int[] quantities = {
        250, 120, 5, 75, 0, 300, 550, 25, 16, 630, 3
    };

    public CheckInventoryAction() {
    }

    public void fixup(ExpressionParser parser) {

        System.out.println("SamplePlugin: CheckInventoryAction.fixup called");

        try {
            inputValueExpression =
                inputVariableName != null
                ? new Expression("$" + inputVariableName, parser) : null;
        } catch (EvaluatorException ee) {
            System.out.println("EvaluationException ocurred in
CheckInventoryAction");
        }
    }

    public int execute(ActionContext actionContext, ExecutionContext context)
            throws WorkflowException {

        System.out.println("SamplePlugin: CheckInventoryAction.execute called");

        Object valueObject = inputValueExpression != null
                          ? inputValueExpression.evaluate(context) : null;
        if (valueObject == null)
            throw new PluginException("Sample Plugin", "itemNo is null");

        if (!(valueObject instanceof Long))
            throw new PluginException("Sample Plugin", "itemNo not an integer");

        int itemNo = ((Long)valueObject).intValue();
        int quantity = quantities[itemNo % quantities.length] + itemNo;

        System.out.println("CheckInventoryAction: Output = " + quantity);
        context.setVariableValue(outputVariableName, new Long(quantity));

        return ExecutionContext.CONTINUE;
    }

    public void response(ActionContext actionContext, ExecutionContext
execContext, Object data)
            throws WorkflowException {
    }

    public void startedWorkflowDone(ActionContext actionContext,
                                    ExecutionContext context,
                                    VariableInfo[] output) {
    }
```

Refer to the following related example listings:

- `CheckInventoryActionObject.java` in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory shows how to read the plug-in data in XML format.
- "Implementing the PluginActionData Interface" on page 4-21 shows how to read and save the plug-in data in XML format. This class extends the `CheckInventoryActionObject` class.
- "Defining the PluginActionPanel Class" on page 4-40 shows how to display the plug-in GUI component in the design client.
- "Customizing an Action Tree" on page 4-70 shows how to customize the actions and/or action categories listed in the action trees that are displayed in various dialog boxes within the Studio.

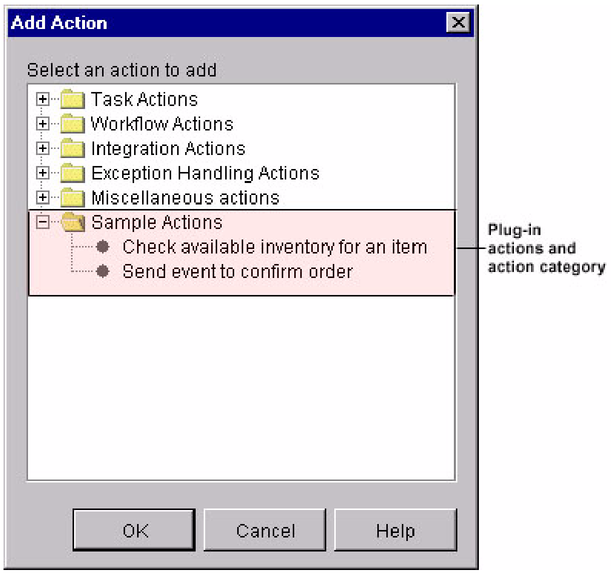
For more information about the plug-in sample, see "BPM Plug-In Sample" on page 10-1.

Customizing the Action Tree

To support plug-in actions, you must customize the actions and/or action categories listed in the action trees that are displayed in various dialog boxes within the Studio.

For example, the following figure illustrates the Add Action dialog box with a customized version of the action tree.

Figure 4-8 Customized Action Tree


Add Action

Select an action to add

- Task Actions
- Workflow Actions
- Integration Actions
- Exception Handling Actions
- Miscellaneous actions
- Sample Actions
  - Check available inventory for an item
  - Send event to confirm order

As shown in the previous figure, the BPM action tree has been customized to add one new action category, Sample Actions, that provides the following plug-in actions:

- Checks available inventory for an item
- Sends Confirm Order Event

You can customize the action tree by performing the following steps:

1. Define a `com.bea.wlpi.common.plugin.CategoryInfo` object that, in turn, defines the custom plug-in actions and/or action categories.

   For information about creating a `CategoryInfo` object, see "Defining Plug-In Value Objects" on page 2-6.

2. Implement the `com.bea.wlpi.server.plugin.Plugin` remote interface `getPluginCapabilitiesInfo()` method to define the `com.bea.wlpi.common.plugin.PluginCapabilitiesInfo` object.

   Use the `CategoryInfo` object defined in step 1 to define the custom action tree characteristics.

When the Plug-in Manager calls the `getPluginCapabilitiesInfo()` method, it must pass the existing action category tree as a `com.bea.wlpi.common.plugin.CategoryInfo` object to enable the plug-in to traverse the tree and determine where to add the custom actions and/or action categories. Once the Plug-in Manager retrieves this valid tree structure, it merges the retrieved tree structure with the existing tree structure and assigns a new `systemID` to each new category.

The Plug-in Manager can call the `getPluginCapabilitiesInfo()` method multiple times, and you must return a newly initialized action tree each time. If you reuse an existing `CategoryInfo` object, the Plug-in Manager raises an `IllegalStateException` when it calls the `setSystemID()` method for the second time.

You can add new actions and/or subcategories to existing action categories at any leve., You cannot, however, remove actions or subcategories from an existing category.

The following code listing is an excerpt from the plug-in sample that shows how to define the following methods:

- `getCategoryInfo()` method, which lets you define a `CategoryInfo` object that, in turn, defines the custom plug-in actions and action categories.
- `getPluginCapabilitiesInfo()` method, which enables you to customize the action tree.

This excerpt is taken from the `SamplePluginBean.java` file in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory. The example defines both the Check Inventory and Send Confirmation actions. Notable lines of code are shown in **bold**.

Listing 4-18 Customizing an Action Tree

```
private CategoryInfo[] getCategoryInfo(SampleBundle bundle) {

    ActionInfo checkInventoryAction =
            new ActionInfo(SamplePluginConstants.PLUGIN_NAME, 1,
                    bundle.getString("checkInventoryName"),
                    bundle.getString("checkInventoryDesc"), ICON_BYTE_ARRAY,
                    ActionCategoryInfo.ID_NEW,
                    ActionInfo.ACTION_STATE_ALL,
                    SamplePluginConstants.CHECKINV_CLASSES);

    ActionInfo sendConfirmAction =
            new ActionInfo(SamplePluginConstants.PLUGIN_NAME, 2,
                    bundle.getString("sendConfirmName"),
                    bundle.getString("sendConfirmDesc"), ICON_BYTE_ARRAY,
                    ActionCategoryInfo.ID_NEW,
                    ActionInfo.ACTION_STATE_ALL,
                    SamplePluginConstants.SENDCONF_CLASSES);

    ActionCategoryInfo[] actions =
            new ActionCategoryInfo[]{ checkInventoryAction, sendConfirmAction};

    CategoryInfo[] catInfo =
          new CategoryInfo[]{ new CategoryInfo(SamplePluginConstants.PLUGIN_NAME,
                    0, bundle.getString("catName"),
                    bundle.getString("catDesc"),
                    ActionCategoryInfo.ID_NEW,
                    actions)};
    return catInfo;
}

public PluginCapabilitiesInfo getPluginCapabilitiesInfo(Locale lc,
        CategoryInfo[] info) {

    PluginInfo pi;
    FieldInfo orderFieldInfo;
    FieldInfo confirmFieldInfo;
    FieldInfo[] fieldInfo;
    FunctionInfo fi;
    FunctionInfo[] functionInfo;
    StartInfo si;
    StartInfo[] startInfo;
    EventInfo ei;
    EventInfo[] eventInfo;
    SampleBundle bundle = new SampleBundle(lc);

    log("getPluginCapabilities called");
    pi = createPluginInfo(lc);

    orderFieldInfo =
            new FieldInfo(SamplePluginConstants.PLUGIN_NAME, 3,
                    bundle.getString("orderFieldName"),
                    bundle.getString("orderFieldDesc"),
                    SamplePluginConstants.ORDER_FIELD_CLASSES, false);

    confirmFieldInfo =
            new FieldInfo(SamplePluginConstants.PLUGIN_NAME, 4,
                    bundle.getString("confirmFieldName"),
                    bundle.getString("confirmFieldDesc"),
                    SamplePluginConstants.CONFIRM_FIELD_CLASSES, false);

    fieldInfo = new FieldInfo[]{ orderFieldInfo, confirmFieldInfo};

    ei = new EventInfo(SamplePluginConstants.PLUGIN_NAME, 6,
                       bundle.getString("confirmOrderName"),
                       bundle.getString("confirmOrderDesc"), ICON_BYTE_ARRAY,
                       SamplePluginConstants.EVENT_CLASSES,
                       confirmFieldInfo);

    eventInfo = new EventInfo[]{ ei};

    fi = new FunctionInfo(SamplePluginConstants.PLUGIN_NAME, 7,
            bundle.getString("calcTotalName"),
            bundle.getString("calcTotalDesc"),
            bundle.getString("calcTotalHint"),
            SamplePluginConstants.FUNCTION_CLASSES, 3, 3);

    functionInfo = new FunctionInfo[]{ fi};

    si = new StartInfo(SamplePluginConstants.PLUGIN_NAME, 5,
            bundle.getString("startOrderName"),
            bundle.getString("startOrderDesc"), ICON_BYTE_ARRAY,
            SamplePluginConstants.START_CLASSES, orderFieldInfo);

    startInfo = new StartInfo[]{ si};

    PluginCapabilitiesInfo pci = new PluginCapabilitiesInfo(pi,
          getCategoryInfo(bundle), eventInfo, fieldInfo, functionInfo, startInfo,
            null, null, null, null, null);

    return pci;
}
```

Refer to the following related example listings:

- `CheckInventoryActionObject.java` in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory shows how to read the plug-in data in XML format.
- "Implementing the PluginActionData Interface" on page 4-21 shows how to read and save the plug-in data in XML format. This class extends the `CheckInventoryActionObject` class.
- "Defining the PluginActionPanel Class" on page 4-40 shows how to display the plug-in GUI component in the design client.
- "Defining the Run-Time Component Class for an Action" on page 4-66 shows how to define the plug-in execution information.

For more information about the plug-in sample, see "BPM Plug-In Sample" on page 10-1.

Defining the Run-Time Component Class for a Done Node

To define the run-time component class for a Done node, implement the `com.bea.wlpi.server.plugin.PluginDone` interface.

Note: The `PluginDone` interface extends `com.bea.wlpi.server.plugin.PluginTemplateNode`. For more information about the `PluginTemplateNode` interface and its methods, see "PluginTemplateNode Interface" on page 4-92.

The `PluginDone` interface adds no other methods.

The following code listing shows how to define the run-time component class for a Done node. Notable lines of code are shown in **bold**.

Note: This class is not available as part of the plug-in sample.

Listing 4-19 Defining the Run-Time Component Class for a Done Node

```
package com.bea.wlpi.test.plugin;

import com.bea.wlpi.common.Messages;
import com.bea.wlpi.common.WorkflowException;
import com.bea.wlpi.evaluator.ExpressionParser;
import com.bea.wlpi.server.common.ExecutionContext;
import com.bea.wlpi.server.plugin.PluginDone;
import java.io.IOException;
import java.util.Map;
import org.xml.sax.*;

public class DoneNode extends DoneObject implements PluginDone {
    public DoneNode() {
    }

    public int activate(ExecutionContext context)
        throws WorkflowException {

        System.out.println("TestPlugin: DoneNode activated");

        // Initialize the plugin instance data.
        Map instanceData =
(Map)context.getPluginInstanceData(TestPluginConstants.PLUGIN_NAME);
        if (instanceData != null) {
            Object started =
instanceData.get(TestPluginConstants.INST_DATA_STARTED);
            System.out.println("instance data = " + started);
        }

        int stopMode;
        if (yesOrNo.equals(TestPluginConstants.DONE_YES)) {
            System.out.println("TestPlugin: DoneNode = YES");
            stopMode = ExecutionContext.CONTINUE;
        } else {
            System.out.println("TestPlugin: DoneNode = NO");
            stopMode = ExecutionContext.STOP;
        }

        return stopMode;
    }

    public void fixup(ExpressionParser parser) {
    }
```

Refer to the following related example listings:

- "Implementing the PluginObject Interface for a Done Node" on page 4-5 shows how to read the plug-in data in XML format.
- "Implementing the PluginData Interface for a Done Node" on page 4-13 shows how to read and save the plug-in data in XML format. This class extends the `PluginObject` class.
- "Defining the PluginPanel Class for a Done Node" on page 4-32 shows how to define plug-in GUI component.

Defining the Run-Time Component Class for an Event Node

To define the run-time component class for an event node, implement the `com.bea.wlpi.server.plugin.PluginEvent` interface.

The following table describes the `PluginEvent` interface methods that you must implement as part of the run-time component class.

Table 4-13 PluginEvent Interface Methods

Method	Description
`public int activate(com.bea.wlpi.server.common.EventContext` *`eventContext`*`, com.bea.wlpi.server.common.ExecutionContext` *`execContext`*`) throws com.bea.wlpi.common.WorkflowException`	Activates the event. The Event Processor calls this method when the matching event is activated by an incoming transition from a predecessor node. Plug-ins then record an event watch to enable the Event Processor to match an incoming event to this particular node and workflow instance. Plug-ins can use the default event watch registration, addressed message handling, and the event matching facility by calling the `activateEvent()` method to the to the `com.bea.wlpi.server.plugin.EventContext`. Plug-ins are responsible for recording the necessary information and providing an event handler to preform the run-time matching based on the defined criteria. For more information about processing plug-in events, see "Processing Plug-In Events" on page 6-1. **Note:** If the plug-in does not rely on the BPM JMS event listener, then it is not required to provide an event handler. The method parameters are defined as follows: ■ *`eventContext`*: `com.bea.wlpi.server.plugin.EventContext` object that specifies the event context. The event context provides access to run-time event-related services, such as event watch registration. ■ *`execContext`*: `com.bea.wlpi.server.common.ExecutionContext` object that specifies the execution context. The execution context provides access to the run-time context, including the template ID, template definition ID, workflow instance ID, event data, and various services related to the workflow execution. For more information about the event and execution contexts, see "Using Plug-In Run-Time Contexts" on page 4-94. This method returns one of the following `com.bea.wlpi.server.common.ExecutionContext` integer values, indicating the return code, that specifies whether or not to processing should continue after the event is activated: ■ `CONTINUE`: Continue processing. ■ `STOP`: Stop processing.

Table 4-13 PluginEvent Interface Methods (Continued)

Method	Description
`public void fixup(com.bea.wlpi.evaluator.ExpressionParser` *`parser`*`) throws com.bea.wlpi.common.WorkflowException`	Enables the plug-in node to compile the necessary expressions. The Plug-in Manager calls this method after parsing the template definition and storing it in memory, and prior to starting the workflow. The method parameter is defined as follows. *`parser`*: `com.bea.wlpi.evaluator.ExpressionParser` object that specifies the expression parser.

Table 4-13 PluginEvent Interface Methods (Continued)

Method	Description
`public int trigger(com.bea.wlpi.server.common.EventContext` *`eventContext`*`, com.bea.wlpi.server.common.ExecutionContext` *`execContext`*`) throws com.bea.wlpi.common.WorkflowException`	Triggers the event. The Event Processor calls this method when it detects an incoming event that matches the criteria for this event node. Plug-ins that use the default event watch registration and matching services must call the `removeEventWatch()` method to the `com.bea.wlpi.server.plugin.EventContext` to place the event in the nonlistening state. Plug-ins that do not use the default event watch registration and matching services must deactivate whatever plug-in-supplied event watch record was established for this node in the prior `activate()` call. The plug-in must provide an event handler if the event arrives via the BPM JMS event listener. Otherwise, it must implement its own event listener service. For more information about defining an event handler, see "Processing Plug-In Events" on page 6-1. The method parameters are defined as follows: ■ *`eventContext`*: `com.bea.wlpi.server.plugin.EventContext` object that specifies the event context. The event context provides access to run-time event-related services such as event watch registration. ■ *`execContext`*: `com.bea.wlpi.server.common.ExecutionContext` object that specifies the execution context. The execution context provides access to the run-time context, including the template ID, template definition ID, workflow instance ID, event data, and various services related to the workflow execution. For more information about the event and execution contexts, see "Using Plug-In Run-Time Contexts" on page 4-94. This method returns one of the following `com.bea.wlpi.server.common.ExecutionContext` integer values indicating the return code that specifies whether or not to processing should continue after the event is triggered: ■ `CONTINUE`: Continue processing. ■ `STOP`: Stop processing.

The following code listing is an excerpt from the plug-in sample that shows how to define the run-time component class for an Event node. This excerpt is taken from the `StartNode.java` file in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory. Notable lines of code are shown in **bold**.

Listing 4-20 Defining the Run-Time Component Class for an Event Node

```
package com.bea.wlpi.tour.po.plugin;

import java.io.IOException;
import com.bea.wlpi.server.plugin.PluginEvent;
import com.bea.wlpi.common.WorkflowException;
import com.bea.wlpi.common.Messages;
import com.bea.wlpi.evaluator.Expression;
import com.bea.wlpi.evaluator.EvaluatorException;
import com.bea.wlpi.server.common.ExecutionContext;
import com.bea.wlpi.server.plugin.EventContext;
import com.bea.wlpi.server.workflow.Workflow;
import com.bea.wlpi.server.workflow.Variable;
import com.bea.wlpi.evaluator.ExpressionParser;
import com.bea.wlpi.server.workflow.TemplateNode;
import org.xml.sax.*;

public class EventNode extends EventObject implements PluginEvent {

    public EventNode() {
    }

    public int activate(EventContext eventContext, ExecutionContext execContext)
            throws WorkflowException {

        System.out.println("SamplePlugin: EventNode activated");
        eventContext.activateEvent(execContext,
                                   SamplePluginConstants.CONTENTTYPE,
                                   eventDesc, null, null);

        return ExecutionContext.CONTINUE;
    }

    public int trigger(EventContext context, ExecutionContext execContext)
            throws WorkflowException {

        System.out.println("SamplePlugin: EventNode triggered");
        context.removeEventWatch(execContext);
        return ExecutionContext.CONTINUE;
    }

    public void fixup(ExpressionParser parser) {
    }
}
```

Refer to the following related example listings:

- `EventObject.java` in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory shows how to implement the `PluginObject` interface to read plug-in data.
- "Implementing the PluginData Interface for an Event Node" on page 4-15 shows how to read and save the plug-in data. This class extends the `EventObject` class.
- "Defining the PluginTriggerPanel Class for an Event Node" on page 4-51 shows how to display the plug-in GUI component in the design client.

For more information about the plug-in sample, see "BPM Plug-In Sample" on page 10-1.

Defining the Run-Time Component Class for a Function

To define the run-time component class for a function, implement the `com.bea.wlpi.common.plugin.PluginFunction` interface. The following table describes the `PluginFunction` interface method that you must implement.

Table 4-14 PluginFunction Interface Method

Method	Description
`public java.lang.Object evaluate(com.bea.wlpi.evaluator.EvaluationContext` *`context`*`, java.lang.Object[]` *`args`* `) throws com.bea.wlpi.common.plugin.PluginException`	Evaluates the function. The expression evaluator calls this method when it needs to evaluate this function call. The plug-in function can calculate its return value from the contextual information supplied via the context parameter. This parameter provides access to event data and the workflow instance state (where appropriate). The method parameters are defined as follows: ■ *`context`*: `com.bea.wlpi.evaluator.EvaluationContext` object that specifies the evaluation context. ■ *`args`*: Array of `java.lang.Object` values that specifies the parameter values. The values are precalculated through the evaluation of the expressions that represent the function arguments in the source expression. This method returns a `java.lang.Object` object that specifies the function evaluation results.

The following code listing is an excerpt from the plug-in sample that shows how to define the run-time component class for a function. This excerpt is taken from the `CalculateTotalPriceFunction.java` file in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory. Notable lines of code are shown in **bold**.

Listing 4-21 Defining a Run-Time Component Class for a Function

```
package com.bea.wlpi.tour.po.plugin;

import com.bea.wlpi.common.Messages;
import com.bea.wlpi.common.plugin.PluginFunction;
import com.bea.wlpi.common.plugin.PluginException;
import com.bea.wlpi.evaluator.*;
import com.bea.wlpi.tour.po.BadStateException;
import java.lang.NumberFormatException;

/**
 * This sample function calculates the total price of an order.  The price is
 * calculated using the itemID, quantity, and the State/Province of the order.
 * The State/Province is used to look up the sales tax rate.
 */
public class CalculateTotalPriceFunction implements PluginFunction {
    static StateTax[] stateTax = {
        new StateTax("AB", 0.07), new StateTax("AK", 0.06),
        new StateTax("AL", 0.06), new StateTax("AR", 0.03),
        new StateTax("AZ", 0.05), new StateTax("BC", 0.05),
        new StateTax("CA", 0.04), new StateTax("CO", 0.08),
        new StateTax("CT", 0.03), new StateTax("DC", 0.05),
        new StateTax("DE", 0.05), new StateTax("FL", 0.00),
        new StateTax("GA", 0.06), new StateTax("HI", 0.07),
        new StateTax("IA", 0.07), new StateTax("ID", 0.08),
        new StateTax("IL", 0.06), new StateTax("IN", 0.03),
        new StateTax("KS", 0.05), new StateTax("KY", 0.07),
        new StateTax("LA", 0.06), new StateTax("MA", 0.05),
        new StateTax("MB", 0.05), new StateTax("MD", 0.04),
        new StateTax("ME", 0.04), new StateTax("MI", 0.03),
        new StateTax("MN", 0.05), new StateTax("MO", 0.06),
        new StateTax("MS", 0.07), new StateTax("MT", 0.07),
        new StateTax("NB", 0.08), new StateTax("NC", 0.07),
        new StateTax("ND", 0.08), new StateTax("NE", 0.03),
        new StateTax("NF", 0.06), new StateTax("NH", 0.09),
        new StateTax("NJ", 0.03), new StateTax("NM", 0.06),
        new StateTax("NV", 0.03), new StateTax("NY", 0.06),
        new StateTax("NS", 0.08), new StateTax("NT", 0.07),
        new StateTax("OH", 0.07), new StateTax("OK", 0.02),
        new StateTax("ON", 0.08), new StateTax("OR", 0.08),
        new StateTax("PA", 0.07), new StateTax("PE", 0.07),
        new StateTax("PQ", 0.05), new StateTax("RI", 0.05),
        new StateTax("SC", 0.05), new StateTax("SD", 0.04),
        new StateTax("SK", 0.04), new StateTax("TN", 0.06),
        new StateTax("TX", 0.06), new StateTax("UT", 0.07),
        new StateTax("VA", 0.07), new StateTax("VT", 0.08),
        new StateTax("WA", 0.07), new StateTax("WI", 0.07),
        new StateTax("WV", 0.08), new StateTax("WY", 0.05),
        new StateTax("YT", 0.07)
    };
    static double[] prices = {
        29.95, 524.79, 33.21, 9.99, 12.28, 152.50, 43.55, 32.90, 328.55, 72.50,
        87.50
    };

    public CalculateTotalPriceFunction() throws EvaluatorException {
        System.out.println("CalculateTotalPriceFunction: constructor called");
    }

    public Object evaluate(EvaluationContext context, Object[] args)
            throws PluginException {

        int itemNo;
        int quantity;
        String state;

        System.out.println("CalculateTotalPriceFunction: evaluate called");

        try {
            itemNo = ((Long)args[0]).intValue();
        } catch (Exception e) {
            throw new PluginException(SamplePluginConstants.PLUGIN_NAME,
                                      "Invalid ItemID argument");
        }

        try {
            quantity = ((Long)args[1]).intValue();
        } catch (Exception e2) {
            e2.printStackTrace();

            throw new PluginException(SamplePluginConstants.PLUGIN_NAME,
                                      "Invalid Quantity argument");
        }

        if (!(args[2] instanceof String)) {
            throw new PluginException(SamplePluginConstants.PLUGIN_NAME,
                                      "Invalid State argument");
        }

        state = (String)args[2];

        int i;

        // Find the state/province in the stateTax array
        for (i = 0; i < stateTax.length; ++i) {
            if (stateTax[i].equals(state))
```

```
                break;
        }

        if (i == stateTax.length)
            throw new PluginException(new BadStateException("Invalid state 
abbreviation: "
                    + state));

        double total = (prices[itemNo % prices.length] + itemNo / prices.length)
                       * quantity * (1 + stateTax[i].getTax());

        return new Double(total);
    }
}

class StateTax {
    String abbrev;
    double tax;

    public boolean equals(String abbrev) {
        return this.abbrev.equalsIgnoreCase(abbrev);
    }

    public double getTax() {
        return tax;
    }

    public StateTax(String abbrev, double tax) {
        this.abbrev = abbrev;
        this.tax = tax;
    }
}
```

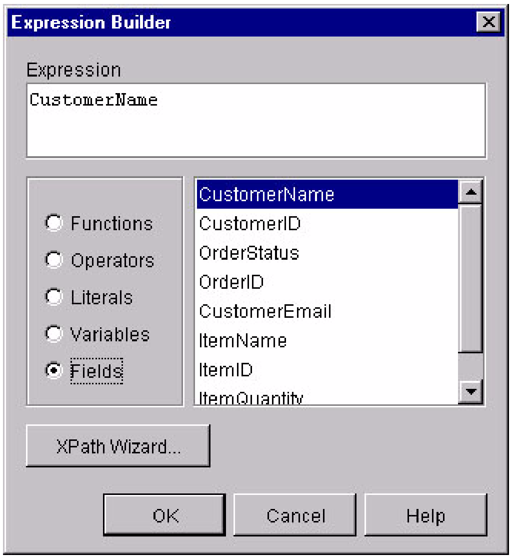
For more information about the plug-in sample, see "BPM Plug-In Sample" on page 10-1.

Defining the Run-Time Component Class for a Message Type

To define the run-time component class for a message type, you must implement the `com.bea.wlpi.common.plugin.PluginField` interface to define a plug-in field. A plug-in field enables you to parse custom plug-in data that is associated with an external event received by a Start or Event node. The data can then be referenced from an evaluator expression. The plug-in determines the content type of the external event by accessing the associated event descriptor.

The plug-in framework uses the plug-in field data to populate the Expression Builder dialog box. For example, the following figure shows an Expression Builder dialog box in which the plug-in Fields category is selected, resulting in the display of a list of valid field elements.

Figure 4-9 Plug-In Fields Displayed in an Expression Builder Dialog Box



Expression: CustomerName

( ) Functions
( ) Operators
( ) Literals
( ) Variables
(•) Fields

CustomerName
CustomerID
OrderStatus
OrderID
CustomerEmail
ItemName
ItemID
ItemQuantity

To define a plug-in field, implement the `com.bea.wlpi.common.plugin.PluginField` interface. The following table describes the `PluginField` interface methods that you must implement.

Table 4-15 PluginField Interface Methods

Method	Description
`public java.lang.Object evaluate(com.bea.wlpi.evaluator.EvaluationContext` *`context`*`) throws com.bea.wlpi.common.plugin.PluginException`	Evaluates the field. The expression evaluator calls this method when it needs to retrieve the value of the field referenced by this object from the event contained in the evaluation context parameter. When calculating its value, the plug-in field must take into account its field qualifiers. The method parameter is defined as follows. *`context`*: `com.bea.wlpi.evaluator.EvaluationContext` object that specifies the evaluation context. This method returns a `java.lang.Object` object that specifies the field evaluation results.
`public void init(java.lang.String` *`name`*`, java.lang.String` *`eventDescriptor`*`) throws com.bea.wlpi.common.plugin.PluginException`	Initializes the values for the field name and event descriptor. The Plug-in Manager uses the name of the field whose value is requested in the context of the event descriptor. A field type can support one or more message types. If multiple message types are supported, the field type uses the event descriptor to distinguish between types. The method parameters are defined as follows: ■ *`name`*: `java.lang.String` object that specifies the field name. ■ *`eventDescriptor`*: `java.lang.String` object that specifies the field event descriptor.

Table 4-15 PluginField Interface Methods (Continued)

Method	Description
`public void setQualifier(com.bea.wlpi.server.plugin.PluginField` *`qualifier`*`) throws com.bea.wlpi.common.plugin.PluginException`	Sets the field qualifier. The Plug-in Manager calls this method when the expression parser encounters a qualified field reference. This method provides the plug-in fields with an opportunity to access external data dictionary information and retrieve column details to be saved as part of the compiled expression. The method parameter is defined as follows. *`qualifier`*: `com.bea.wlpi.common.plugin.PluginField` object that specifies the field qualifier of the same class as `this` object.

The following code listing is an excerpt from the plug-in sample that shows how to define a run-time component class for a message type. The plug-in field processes a single message type containing data associated with a customer order. The data is in `String` format, and the individual data elements are separated by semicolons. The plug-in field expects the external event data to be provided as a string buffer containing values that are delimited by semicolons. In the `execute()` method, the class performs some simple validation of the event data, and returns the `String` object for the requested field name. This excerpt is taken from the `OrderField.java` file in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory. Notable lines of code are shown in **bold**.

Listing 4-22 Defining the Run-Time Component Class for a Message Type

```
package com.bea.wlpi.tour.po.plugin;

import com.bea.wlpi.common.plugin.PluginException;
import com.bea.wlpi.common.plugin.PluginField;
import com.bea.wlpi.evaluator.EvaluationContext;
import com.bea.wlpi.server.eventprocessor.EventData;
import java.util.StringTokenizer;
```

```
/*
 * This sample field type expects a string buffer with semicolon delimited
 * field values.  The fields are in a fixed order.
 */

public final class OrderField implements PluginField {
    private String docType;
    private String name;

    public void init(String name, String eventDescriptor)
            throws PluginException {
        this.name = name;
        docType = eventDescriptor;
    }

    public void setQualifier(PluginField qualifier) throws PluginException {

        System.out.println("OrderField.setQualifier(" + qualifier + ')');

        throw new PluginException(SamplePluginConstants.PLUGIN_NAME,
                                  "Qualifiers are not supported");
    }

    public Object evaluate(EvaluationContext context) throws PluginException {

        // Get the event data and check that it is a String object.
        EventData eventData = context.getEventData();

        if (eventData == null)
            throw new PluginException(SamplePluginConstants.PLUGIN_NAME,
                                      "The event data is null.");

        docType = eventData.getEventDescriptor();

        if (!docType.equals(SamplePluginConstants.START_ORDER_EVENT))
            throw new PluginException(SamplePluginConstants.PLUGIN_NAME,
                                      "The event descriptor is invalid.");

        Object object = eventData.getContent();

        if (!(object instanceof String))
            throw new PluginException(SamplePluginConstants.PLUGIN_NAME,
                                      "The event data is invalid.");

        // Check to make sure the field name is valid
        int i;

        for (i = 0; i < SamplePluginConstants.ORDER_FIELDS.length; i++) {
            if (SamplePluginConstants.ORDER_FIELDS[i].equals(name))
```

```
                break;
        }

        // Was the field name found in the list of valid field names?
        if (i == SamplePluginConstants.ORDER_FIELDS.length)
            throw new PluginException(SamplePluginConstants.PLUGIN_NAME,
                                      "The field name " + name
                                      + " is invalid.");

        String data = (String)object;
        StringTokenizer st = new StringTokenizer(data, ";");
        String token = null;

        while (st.hasMoreTokens() && i >= 0) {
            token = st.nextToken();

            i--;
        }

        // Did the data ran out of fields prior to finding the one required?
        if (i >= 0) {
            throw new PluginException(SamplePluginConstants.PLUGIN_NAME,
                                      "The event data is invalid.");
        }

        String value = token;

        System.out.println("OrderField: name = " + name + ", value = " + value);

        // Return the text value.
        return value;
    }
}
```

The `ORDER_FIELDS` value is defined within the `SamplePluginConstants.java` class file as follows:

```
final static String[] ORDER_FIELDS = {
     "CustomerName", "CustomerID", "OrderStatus", "OrderID",
     "CustomerEmail", "ItemName", "ItemID", "ItemQuantity",
     "CustomerState"
};
```

The figure “Plug-In Fields Displayed in an Expression Builder Dialog Box” on page 4-84 shows the field elements populated within the Expression Builder dialog box.

For more information about the plug-in sample, see "BPM Plug-In Sample" on page 10-1.

Defining the Run-Time Component Class for a Start Node

To define the run-time component class for a Start node, implement the `com.bea.wlpi.server.plugin.PluginStart2` interface. The following table describes the `PluginStart2` interface method that you must implement.

Note: The `PluginStart2` interface inherits methods from the `com.bea.wlpi.server.plugin.PluginTemplateNode` interface. For more information, see "PluginTemplateNode Interface" on page 4-92.

Table 4-16 PluginStart2 Interface Method

Method	Description
`public void setTrigger(com.bea.wlpi.server.plugin.EventContext` *`context`*`, java.lang.String` *`orgExpr`*`, boolean` *`orgIsExpression`* `) throws com.bea.wlpi.common.WorkflowException`	Sets the event watch for this Start node. The template definition calls this method when the template definition is activated and saved. Plug-ins use this method to record an event watch. Using recorded event watches, the Event Processor can match an incoming event to this particular node and template definition. Plug-ins can use the default event watch registration, addressed message handling, and event matching facility by calling the `com.bea.wlpi.server.plugin.EventContext.postStartWatch()` method. Plug-ins must provide an event handler to perform run-time matching. **Note:** If a plug-in does not rely on the BPM JMS event listener, it is not required to provide an event handler. The method parameters are defined as follows: ■ *`eventContext`*: `com.bea.wlpi.server.plugin.EventContext` object that specifies the Start node event context. The event context provides access to run-time event-related services, such as event watch registration. ■ *`orgExpr`*:: `java.lang.String` object that specifies the expression used to generate the ID of the organization in which to instantiate the workflow. ■ *`orgIsExpression`*:: Boolean value that indicates whether the specified organization is an expression (`true`) or not (`false`). For more information about defining an event handler to process plug-in events, see “Processing Plug-In Events” on page 6-1.

The following code listing is an excerpt from the plug-in sample that shows how to define the run-time component class for a Start node. This excerpt is taken from the `StartNode.java` file in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory. Notable lines of code are shown in **bold**.

Listing 4-23 Defining the Run-Time Component Class for a Start Node

```
package com.bea.wlpi.tour.po.plugin;

import java.io.IOException;
import com.bea.wlpi.server.plugin.PluginStart2;
.
.
.
public class StartNode extends StartObject implements PluginStart2 {

    public StartNode() {
    }
.
.
.
    public void setTrigger(EventContext context, String orgExpr,
                           boolean orgIsExpr)
            throws WorkflowException {

        System.out.println("SamplePlugin: StartNode - setTrigger called");
        context.postStartWatch(SamplePluginConstants.CONTENTTYPE, eventDesc,
                               null, null);
    }

    public void fixup(ExpressionParser parser) {
    }
}
```

Refer to the following related example listings:

- "Implementing the PluginObject Interface for a Start Node" on page 4-8 shows how to read the plug-in data in XML format.
- "Implementing the PluginData Interface for a Start Node" on page 4-16 shows how to read and save plug-in data in XML format. This example extends the `StartObject` class.

- "Defining the PluginTriggerPanel Class for a Start Node" on page 4-48 shows how to display the plug-in GUI component in the design client.
- "Using Plug-In Run-Time Contexts" on page 4-94 shows how to define the plug-in fields that can be referenced from an evaluator expression.

For more information about the plug-in sample, see "BPM Plug-In Sample" on page 10-1.

PluginTemplateNode Interface

The `com.bea.wlpi.server.plugin.PluginTemplateNode` interface provides methods for activating Done and Start nodes, and compiling their expressions.

The `PluginTemplateNode` interface is extended by the following interfaces:

- `com.bea.wlpi.server.plugin.PluginDone`
- `com.bea.wlpi.server.plugin.PluginStart2`

The following table describes the `PluginTemplateNode` interface methods that you must implement as part of the run-time component class when defining a Done or Start node.

Table 4-17 PluginTemplateNode Interface Methods

Method	Description
`public void activate(com.bea.wlpi.server.common.ExecutionContext` *`context`*`) throws com.bea.wlpi.common.WorkflowException`	Activates the node. The plug-in framework calls this method when the matching node is activated by an incoming transition from a predecessor node. The method parameter is defined as follows. *`execContext`*: `com.bea.wlpi.server.common.ExecutionContext` object that specifies the execution context. The execution context provides access to the run-time context, including the template ID, template definition ID, workflow instance ID, event data, and various services related to the workflow execution. For more information about the execution context, see "Using Plug-In Run-Time Contexts" on page 4-94.
`public void fixup(com.bea.wlpi.evaluator.ExpressionParser` *`parser`*`) throws com.bea.wlpi.common.WorkflowException`	Enables the plug-in node to compile the necessary expressions. The Plug-in Manager calls this method after parsing the template definition and storing it in memory, and prior to starting the workflow. The plug-in Start and Done nodes should perform all expensive initialization steps at this time. The method parameter is defined as follows. *`parser`*: `com.bea.wlpi.evaluator.ExpressionParser` object that specifies the expression parser.

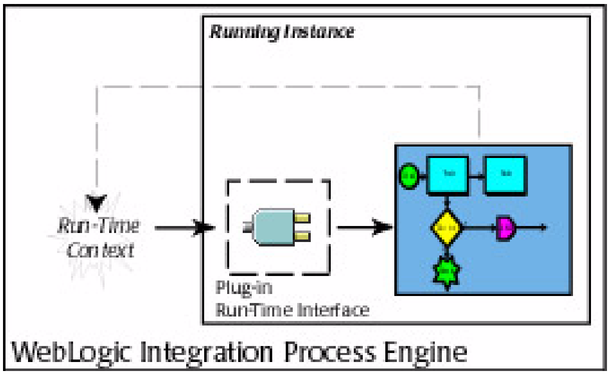
For more information about the `PluginTemplateNode` interface, see the `com.bea.wlpi.server.plugin.PluginTemplateNode` Javadoc.

Using Plug-In Run-Time Contexts

To define the run-time execution information for the plug-in, you must implement a run-time interface for the plug-in component, as defined in “Executing the Plug-In” on page 4-59. At run time, the plug-in communicates with the process engine using a process called *context passing*: the Plug-in Manager obtains an instance of the plug-in component run-time interface and passes the context to it.

The following figure illustrates context passing.

Figure 4-10 Context Passing


Running Instance
Run-Time Context
Plug-in Run-Time Interface
WebLogic Integration Process Engine

Each *context interface* provides restricted access to the Plug-in Manager functionality, enabling the plug-in to execute and manage its own application logic, and introduce the plug-in instance data into the BPM run-time environment.

The following table describes the plug-in run-time context interfaces.

Table 4-18 Plug-In Run-Time Context Interfaces

The following context . . .	Provides . . .
`com.bea.wlpi.server.plugin.ActionContext`	Run-time context and services associated with an action.
`com.bea.wlpi.evaluator.EvaluationContext`	Run-time evaluation parameters for elements in an expression.

Table 4-18 Plug-In Run-Time Context Interfaces (Continued)

The following context . . .	Provides . . .
`com.bea.wlpi.server.plugin.EventContext`	Run-time context and services associated with an event.
`com.bea.wlpi.server.common.ExecutionContext`	Execution context of a running workflow instance.
`com.bea.wlpi.common.plugin.PluginPanelContext`	Client-side context for the BPM design client.

The following section describes the context interfaces in more detail.

Action Context

The `com.bea.wlpi.server.plugin.ActionContext` interface provides the run-time context and services for plug-in actions. This context is passed via the `com.bea.wlpi.server.plugin.PluginAction` interface `execute()` method.

The following table describes the `ActionContext` interface methods that you can use to access information about the action context.

Table 4-19 ActionContext Interface Methods

Method	Description
`public int executeSubActionList(int` *`index`*`, com.bea.wlpi.server.common.ExecutionContext` *`context`*`) throws com.bea.wlpi.common.WorkflowException`	Executes each subaction on a list. The method parameters are defined as follows: ■ *`index`*: Integer value that specifies the index of the subaction list to execute. This value equates to the index of the `classNames` array that corresponds to the `com.bea.wlpi.common.plugin.ActionInfo` object. ■ *`execContext`*: `com.bea.wlpi.server.common.ExecutionContext` object that specifies the execution context that is passed by the caller. The execution context provides access to the run-time context, including the template ID, template definition ID, workflow instance ID, event data, and various services related to the workflow execution. This method returns one of the following `com.bea.wlpi.server.common.ExecutionContext` integer values, indicating the return code, that specifies whether or not processing should continue: ■ `CONTINUE`: Continue processing. ■ `STOP`: Stop processing. In this case, the caller must return immediately and pass this return value. For more information about the execution context, see "Execution Context" on page 4-106.

Table 4-19 ActionContext Interface Methods (Continued)

Method	Description
`public java.lang.String getActionId() throws com.bea.wlpi.common.WorkflowException`	Gets the ID that uniquely defines this action. This ID is required to support asynchronously executed callbacks to plug-in actions. The ID must be passed to the `com.bea.wlpi.server.worklist.Worklist` interface `response()` method. This method returns a `java.lang.String` object that specifies the action ID.

Table 4-19 ActionContext Interface Methods (Continued)

Method	Description
`public java.lang.String instantiateWorkflow(com.bea.wlpi.server.common.ExecutionContext` *`context`*`, java.lang.String` *`orgID`*`, java.lang.String` *`templateID`*`, com.bea.wlpi.common.VariableInfo[]` *`initialValues`*`, java.util.Map` *`pluginData`*`) throws com.bea.wlpi.common.WorkflowException`	Creates a new workflow instance. The method parameters are defined as follows: ■ *`context`*: `com.bea.wlpi.server.common.ExecutionContext` object that specifies the execution context that is passed by the caller. The execution context provides access to the run-time context, including the template ID, template definition ID, workflow instance ID, event data, and various services related to the workflow execution. For more information about the execution context, see “Execution Context” on page 4-106. ■ *`orgID`*: `java.lang.String` object that specifies the ID of the organization in which the workflow should be instantiated. ■ *`templateID`*: `java.lang.String` object that specifies the ID of the workflow template to instantiate. ■ *`intialValues`*: Array of `com.bea.wlpi.common.VariableInfo` objects that specifies the ID of the workflow template to be instantiated. ■ *`pluginData`*: `java.util.Map` object that specifies a map of all plug-in instance data, keyed on the plug-in name. This method returns an XML document that is compliant with the Client Request DTD, `ClientReq.dtd`, as described in “DTD Formats” in *Programming BPM Client Applications*. The XML document contains information about the running instance, including the instance ID and template definition ID. It can be accessed by parsing the document using an XML parser, such as a SAX (Simple API for XML) parser.

Evaluation Context

The `com.bea.wlpi.evaluator.EvaluationContext` interface provides the run-time evaluation parameters for the elements in an expression. This context is passed via the `evaluate()` method to the `com.bea.wlpi.server.plugin.PluginField` interface and the `com.bea.wlpi.server.plugin.PluginFunction` interface.

The following table describes the `EvaluationContext` interface methods that you can use to access information about the evaluation context.

Table 4-20 EvaluationContext Interface Methods

Method	Description
`public final int getCalendarType()`	Gets the type of calendar to use when performing date arithmetic. This method returns an integer value that specifies one of the following calendar types, as defined by the `com.bea.wlpi.evaluator.ExecutionContext` interface: ■ `CALTYPE_ASSIGNEE` (1): Task assignee calendar. ■ `CALTYPE_GREGORIAN` (3): Gregorian calendar. ■ `CALTYPE_ORG` (0): Instance organization calendar. ■ `CALTYPE_SPECIFIC` (2): Specific calendar.
`public final com.bea.wlpi.server.eventprocessor.EventData getEventData()`	Gets the data for the current event. This method returns a `com.bea.wlpi.server.eventprocessor.EventData` object that specifies the current event data.

Table 4-20 EvaluationContext Interface Methods (Continued)

Method	Description
`public final com.bea.wlpi.server.common.ExecutionContext getExecutionContext()`	Gets the execution context within which to evaluate the expression. This method returns an `com.bea.wlpi.server.common.ExecutionContext` object that specifies the execution context. For more information about the execution context, see "Execution Context" on page 4-106.
`public final java.lang.String getTaskID()`	Gets the user ID of the current task, if any. This method returns a `java.lang.String` object that specifies the task ID.
`public final java.lang.String getUserID()`	Gets the user ID that made the current top-level API call. This method returns a `java.lang.String` object that specifies the user ID.

Table 4-20 EvaluationContext Interface Methods (Continued)

Method	Description
`public java.lang.String instantiateWorkflow(com.bea.wlpi.server .common.ExecutionContext` *`context`*`, java.lang.String` *`orgID`*`, java.lang.String` *`templateID`*`, com.bea.wlpi.common.VariableInfo[]` *`initialValues`*`, java.util.Map` *`pluginData`*`) throws com.bea.wlpi.common.WorkflowException`	Creates a new workflow instance. The method parameters are defined as follows: ■ *`context`*: `com.bea.wlpi.server.common.ExecutionContext` object that specifies the execution context that is passed by the caller. The execution context provides access to the run-time context, including the template ID, template definition ID, workflow instance ID, event data, and various services related to the workflow execution. For more information about the execution context, see “Execution Context” on page 4-106. ■ *`orgID`*: `java.lang.String` object that specifies the ID of the organization in which the workflow should be instantiated. ■ *`templateID`*: `java.lang.String` object that specifies the ID of the workflow template to be instantiated. ■ *`intialValues`*: Array of `com.bea.wlpi.common.VariableInfo` objects that specifies the ID of the workflow template to instantiate. ■ *`pluginData`*: `java.util.Map` object that specifies a map of all plug-in instance data, keyed on the plug-in name. This method returns an XML document that is compliant with the Client Request DTD, `ClientReq.dtd`, as described in “DTD Formats” in *Programming BPM Client Applications*. The XML document contains information about the running instance, including the instance ID and template definition ID. It can be accessed by parsing the document using an XML parser, such as a SAX (Simple API for XML) parser.

Event Context

The `com.bea.wlpi.server.plugin.EventContext` interface provides the run-time context and services to plug-in events. This context is passed via the following methods:

- `activate()` and `trigger()` methods to the `com.bea.wlpi.server.plugin.PluginEvent` interface
- `settrigger()` method to the `com.bea.wlpi.server.plugin.PluginStart2` interface

The following table describes the `EventContext` interface methods that you can use to access information about the event context.

Table 4-21 Event Context Interface Methods

Method	Description
`public void activateEvent(com.bea.wlpi.server.common.ExecutionContext` *`context`*`, java.lang.String` *`contentType`*`, java.lang.String` *`eventDescriptor`*`, java.lang.String` *`keyValue`*`, java.lang.String` *`condition`*`) throws com.bea.wlpi.common.WorkflowException`	Performs default event activation. Checks whether the Event Processor has received and persisted a message addressed to this workflow instance or template. If no message exists, an event watch record is posted. If a message exists, the matching event is consumed and the event node is triggered if the following criteria are met: ■ Message contains the required content type and event descriptor. ■ Key value matches the key value specified by the caller (if specified). ■ Conditional expression evaluates to `true` (if specified). If passing a content type other than `text/xml`, it is the plug-in's responsibility to ensure that a matching event key expression is registered in the event key table. This can be accomplished using the `addEventKey()` method to the `com.bea.wlpi.server.admin.Admin` interface. The plug-in should also provide a plug-in field to evaluate a key value from incoming data of this content type and format. For more information about defining a plug-in field, see "Defining the Run-Time Component Class for a Message Type" on page 4-84. The method parameters are defined as follows: ■ *`context`*: `com.bea.wlpi.server.common.ExecutionContext` object that specifies the execution context that is passed by the caller. For more information about the execution context, see "Execution Context" on page 4-106. ■ *`contentType`*: `java.lang.String` object that specifies the MIME content type that describes the basic data type of the message. ■ *`eventDescriptor`*: `java.lang.String` object that specifies the event descriptor in a format that is compatible with the `contentType` value, or null. ■ *`keyValue`*: `java.lang.String` object that specifies the key value required to trigger this call, or null. ■ *`condition`*: `java.lang.String` object that specifies the conditional expression to evaluate against the event, or null. This condition must evaluate to true before the event can be triggered.

Table 4-21 Event Context Interface Methods (Continued)

Method	Description
`public java.lang.String getNodeId()`	Gets the ID of the Event or Start node. This ID is required to support asynchronously executed callbacks to plug-in actions. The ID must be passed to the `response()` method to the `com.bea.wlpi.server.worklist.Worklist` interface. This method returns a `java.lang.String` object that specifies the action ID.
`public java.lang.String getTemplateDefinitionID()`	Gets the ID of the workflow template definition. This method returns a `java.lang.String` object that specifies the template definition ID.
`public java.lang.String getTemplateID()`	Gets the ID of the workflow template. This method returns a `java.lang.String` object that specifies the template ID.
`public void postStartWatch(java.lang.String` *`contentType`*`, java.lang.String` *`eventDescriptor`*`, java.lang.String` *`keyValue`*`, java.lang.String` *`condition`*`)`	Registers an Event Processor watch record for the specified message. **Note:** This method can only be called by a Start node. If passing a content type other than `text/xml`, it is the plug-in's responsibility to ensure that a matching event key expression is registered in the event key table. This can be accomplished using the `addEventKey()` method to the `com.bea.wlpi.server.admin.Admin` interface. The plug-in should also provide a plug-in field to evaluate a key value from incoming data of this content type and format. For more information about defining a plug-in field, see "Defining the Run-Time Component Class for a Message Type" on page 4-84. The method parameters are defined as follows: ■ *`contentType`*: `java.lang.String` object that specifies the MIME content type that describes the basic data type of the message. ■ *`eventDescriptor`*: `java.lang.String` object that specifies the event descriptor in a format that is compatible with the `contentType` value, or null. ■ *`keyValue`*: `java.lang.String` object that specifies the key value required to trigger this call, or null. ■ *`condition`*: `java.lang.String` object that specifies the conditional expression to evaluate against the event, or null. This condition must evaluate to true before the event can be triggered.

Table 4-21 Event Context Interface Methods (Continued)

Method	Description
`public void removeEventWatch(com.bea.wlpi.server.common.ExecutionContext` *`context`*`)`	Unregisters the Event Processor watch record for the specified message. **Note:** This message should be called only by an Event node. The method parameter is defined as follows. *`context`*: `com.bea.wlpi.server.common.ExecutionContext` object that specifies the execution context that is passed to the caller. For more information about the execution context, see "Execution Context" on page 4-106.
`public void removeStartWatch()`	Unregisters the Event Processor watch record for the specified message. **Note:** This message should be called only by a Start node.
`public java.lang.String instantiateWorkflow(com.bea.wlpi.server.common.ExecutionContext` *`context`*`, java.lang.String` *`orgID`*`, java.lang.String` *`templateID`*`, com.bea.wlpi.common.VariableInfo[]` *`initialValues`*`, java.util.Map` *`pluginData`*`) throws com.bea.wlpi.common.WorkflowException`	Creates a new workflow instance. The method parameters are defined as follows: ■ *`context`*: `com.bea.wlpi.server.common.ExecutionContext` object that specifies the execution context that is passed by the caller. The execution context provides access to the run-time context, including the template ID, template definition ID, workflow instance ID, event data, and various services related to the workflow execution. ■ *`orgID`*: `java.lang.String` object that specifies the ID of the organization in which the workflow should be instantiated. ■ *`templateID`*: `java.lang.String` object that specifies the ID of the workflow template to be instantiated. ■ *`intialValues`*: Array of `com.bea.wlpi.common.VariableInfo` objects that specifies the ID of the workflow template to be instantiated. ■ *`pluginData`*: `java.util.Map` object that specifies a map of all plug-in instance data, keyed on the plug-in name. This method returns an XML document that is compliant with the Client Request DTD, `ClientReq.dtd`, as described in "DTD Formats" in *Programming BPM Client Applications*. The XML document contains information about the running instance, including the instance ID and template definition ID. It can be accessed by parsing the document using an XML parser, such as a SAX (Simple API for XML) parser.

Execution Context

The `com.bea.wlpi.server.common.ExecutionContext` interface provides the run-time context and services for a running workflow instance. This context is passed via the following methods:

- `com.bea.wlpi.server.plugin.PluginAction` interface `execute()`, `response()`, and `startedWorkflowDone()` methods
- `com.bea.wlpi.server.plugin.PluginEvent` interface `activate()` and `trigger()` methods
- `com.bea.wlpi.server.plugin.PluginTemplateNode` interface `activate()` method (used by the Start and Done nodes)

The following table describes the `ExecutionContext` interface methods that you can use to access information about the action context.

Table 4-22 ExecutionContext Interface Methods

Method	Description
`public void addClientResponse(java.lang.String xml)`	Appends an XML document to the API method return value. The method parameter is defined as follows. *`xml`*: `java.lang.String` object that specifies the XML document to be appended.
`public java.lang.String getErrorHandler() throws com.bea.wlpi.common.WorkflowException`	Gets the name of the current error handler. This method returns a `java.lang.String` object that specifies the current error handler.
`public com.bea.wlpi.server.eventprocessor.EventData getEventData()`	Gets the data associated with the current event, if any. This method returns a `com.bea.wlpi.server.eventprocessor.EventData` object that specifies the event data, or an empty string that specifies the system eventhandler.
`public int getExceptionNumber()`	Gets the message number of the error being handled by the eventhandler. This method returns an integer value that specifies the message number.

Table 4-22 ExecutionContext Interface Methods (Continued)

Method	Description
`public java.lang.Exception getExceptionObject()`	Gets the exception object being handled by the event handler. This method returns a `java.lang.Exception` object that specifies the exception object.
`public int getExceptionSeverity()`	Gets the exception severity code of the error being handled by the event handler. This method returns an integer value that specifies one of the following `com.bea.wlpi.common.WorkflowException` severity codes: ■ `ERROR_CUSTOM`: Custom error raised either by an application or by a workflow executing the `Invoke Error handler` action. ■ `ERROR_SYSTEM`: Fatal exception occurred while a user request was being processed. ■ `ERROR_UNKNOWN`: Unknown error type (internal use only). ■ `ERROR_WORKFLOW`: Fatal, illegal condition, such as inconsistent workflow state. ■ `WARNING_WORKFLOW`: Nonfatal workflow condition that the user can rectify manually.
`public java.lang.String getExceptionText()`	Gets the message text of the exception being handled by the event handler. This method returns a `java.lang.String` object that specifies the message text.
`public java.lang.String getExceptionType()`	Gets the message type of the exception being handled by the event handler. This method returns a `java.lang.String` object that specifies the message text.
`public java.lang.String getInstanceID()`	Gets the ID of the current workflow instance. This method returns a `java.lang.String` object that specifies the ID.

Table 4-22 ExecutionContext Interface Methods (Continued)

Method	Description
`public java.lang.String getOrg() throws com.bea.wlpi.common.WorkflowException`	Gets the ID of the organization in which the current instance is running. This method returns a `java.lang.String` object that specifies the message text.
`public java.lang.Object getPluginInstanceData(java.lang.String` *`pluginName`*`) throws com.bea.wlpi.common.WorkflowException`	Gets the workflow instance data provided by the named plug-in. The method parameter is defined as follows. *`pluginName`*: `java.lang.String` object that specifies the plug-in name. This method returns a `java.lang.Object` object that specifies the plug-in instance data.
`public java.lang.String getRequestor()`	Gets the ID of the user that made the current API request. This method returns a `java.lang.String` object that specifies the requestor ID.
`public boolean getRollbackOnly()`	Determines whether the current user transaction has been marked for rollback only. This method returns `true` if the transaction is set for rollback only, and `false` otherwise.
`public java.lang.String getTemplateDefinitionID()`	Gets the ID of the current template definition. This method returns a `java.lang.String` object that specifies the template definition ID.
`public com.bea.wlpi.common.plugin.PluginObject getTemplateDefintionPluginData(java.lang.String` *`pluginName`*`)`	Gets the template definition data for the specified plug-in. The method parameter is defined as follows. *`pluginName`*: `java.lang.String` object that specifies the plug-in name. This method returns a `com.bea.wlpi.common.plugin.PluginObject` object that specifies the template definition data.
`public java.lang.String getTemplateID()`	Gets the ID of the current template. This method returns a `java.lang.String` object that specifies the template ID.

Table 4-22 ExecutionContext Interface Methods (Continued)

Method	Description
`public com.bea.wlpi.common.plugin.PluginObject getTemplatePluginData(java.lang.String` *`pluginName`*`)`	Gets the template data for the specified plug-in. The method parameter is defined as follows. *`pluginName`*: `java.lang.String` object that specifies the plug-in name. This method returns a `com.bea.wlpi.common.plugin.PluginObject` object that specifies the template data.
`public com.bea.wlpi.common.VariableInfo getVariableInfo(java.lang.String` *`name`*`)`	Gets information about the specified plug-in variable. The method parameter is defined as follows. *`name`*: `java.lang.String` object that specifies the variable name. This method returns a `com.bea.wlpi.common.VariableInfo` object that specifies the variable information.
`public java.lang.Object getVariableValue(java.lang.String` *`name`*`) throws com.bea.wlpi.common.WorkflowException`	Gets the variable value for the specified plug-in variable. The method parameter is defined as follows. *`name`*: `java.lang.String` object that specifies the variable name. This method returns a `java.lang.Object` object that specifies the variable information. Also refer to the `getInstanceVariable()` method to the `com.bea.wlpi.server.admin.Admin` interface, as described in "Monitoring Run-Time Variables" in *Programming BPM Client Applications*.

Table 4-22 ExecutionContext Interface Methods (Continued)

Method	Description
`public java.lang.String instantiate(java.lang.String` *`orgID`*`, java.lang.String` *`initialNode`*`, java.lang.String` *`parentTemplateDefinitionID`*`, java.lang.String` *`parentID`*`, java.lang.String` *`parentNodeID`*`, com.bea.wlpi.server.eventprocessor.EventData` *`eventData`*`, java.util.List` *`lVariableValues`*`, java.util.Map` *`pluginData`*`) throws com.bea.wlpi.common.WorkflowException`	Creates a new workflow instance The method parameters are defined as follows: ■ *`orgID`*: `java.lang.String` object that specifies the organization ID that will be associated with the new instance. ■ *`initialNode`*: `java.lang.String` object that specifies the ID of the Start node to be activated. ■ *`parentTemplateDefinitionID`*: `java.lang.String` object that specifies the ID of the parent template definition (if a subworkflow is being instantiated). ■ *`parentID`*: `java.lang.String` object that specifies the ID of the parent workflow instance (if a subworkflow is being instantiated). ■ *`parentNodeID`*: `java.lang.String` object that specifies the ID of the node in the parent workflow to be notified of events in the lifecycle of the workflow (if a subworkflow is being instantiated). ■ *`eventData`*: `com.bea.wlpi.server.eventprocessor.EventData` object that specifies the event data to pass to the called Start nodes in the workflow. (This parameter provides an alternative to using the *`lVariableValues`* value to set variable values explicitly.) ■ *`lVariableValues`*: `java.util.List` object that specifies a list of `com.bea.wlpi.common.VariableInfo` objects that initialize the workflow instance variables. Note that non-null initial values for all mandatory input variables must be passed through this parameter. ■ *`pluginData`*: `java.util.Map` object that specifies a map of all plug-in instance data, keyed on the plug-in name. This method returns a `java.lang.String` object that specifies the ID of the new workflow instance. Also refer to the `instantiateWorkflow()` method to the `com.bea.wlpi.server.worklist.Worklist` interface, as described in "Manually Starting Workflows" in *Programming BPM Client Applications*.

Table 4-22 ExecutionContext Interface Methods (Continued)

Method	Description
`public int invokeErrorHandler(java.lang.String` *`handlerName`*`, java.lang.Exception` *`e`*`)`	Invokes the specified event handler. The method parametes are defined as follows: ■ *`handlerName`*: `java.lang.String` object that specifies the name of the event handler to be invoked. ■ *`e`*: `java.lang.Exception` object that specifies the event value to be thrown. This method returns an integer value that specifies the status of the call.
`public boolean isAuditEnabled()`	Determines whether or not auditing is enabled for the current workflow. This method returns `true` if auditing is enabled, and `false` otherwise.
`public void setErrorHandler(java.lang.String` *`handlerName`*`) throws com.bea.wlpi.common.WorkflowException`	Sets the current event handler. The method parameter is defined as follows. *`handlerName`*: `java.lang.String` object that specifies the one of the following: name of the event handler to set; null, to restore the previous event handler; or an empty string, to set the system event handler.
`public void setPluginInstanceData(java.lang.String` *`pluginName`*`, java.lang.Object` *`data`*`) throws com.bea.wlpi.common.WorkflowException`	Sets the workflow instance data for the specified plug-in. The method parameters are defined as follows: ■ *`pluginName`*: `java.lang.String` object that specifies the plug-in name. ■ *`data`*: `java.lang.Object` object that specifies the plug-in data. Also refer to the `setInstanceVariable()` method to the `com.bea.wlpi.server.admin.Admin` interface, as described in "Monitoring Run-Time Variables" in *Programming BPM Client Applications*.
`public void setRollbackOnly()`	Sets the user transaction for rollback only.

Table 4-22 ExecutionContext Interface Methods (Continued)

Method	Description
`public void setVariableValue(java.lang.String` *`name`*`, java.lang.Object` *`value`*`) throws com.bea.wlpi.common.WorkflowException`	Sets the value of a variable. The method parameters are defined as follows: ■ *`orgID`*: `java.lang.String` object that specifies the organization ID that will be associated with the new instance. ■ *`initialNode`*: `java.lang.String` object that specifies the ID of the Start node to be activated. Also refer to the `setInstanceVariable()` method to the `com.bea.wlpi.server.admin.Admin` interface, as described in "Monitoring Run-Time Variables" in *Programming BPM Client Applications*.

Table 4-22 ExecutionContext Interface Methods (Continued)

Method	Description
`public java.lang.String taskAssign(java.lang.String` *`instanceID`*`, java.lang.String` *`taskID`*`, java.lang.String` *`assigneeID`*`, boolean` *`bRole`*`, boolean` *`bLoadBalance`*`) throws com.bea.wlpi.common.WorkflowException`	Assigns a workflow task to a participant. The method parameters are defined as follows: ■ *`instanceID`*: `java.lang.String` object that specifies the workflow instance ID. ■ *`taskID`*: `java.lang.String` object that specifies the task ID. ■ *`assigneeID`*: `java.lang.String` object that specifies the ID of the assignee (user or role). ■ *`bRole`*: Boolean value that indicates whether the specified assignee is a role (`true`) or user (`false`). ■ *`bLoadBalance`*: Boolean value that specifies whether or not load balancing should be applied across the members of a role. This parameter is ignored if *`bRole`* is set to `false`. This method returns an XML document that is compliant with the Client Request DTD, `ClientReq.dtd`, as described in "DTD Formats" in *Programming BPM Client Applications*. The XML document contains information about the running instance, including the instance ID and template definition ID. It can be accessed by parsing the document using an XML parser, such as a SAX (Simple API for XML) parser. Also refer to the `taskAssign()` method to the `com.bea.wlpi.server.worklist.Worklist` interface, as described in "Managing Run-Time Tasks" in *Programming BPM Client Applications*.

Table 4-22 ExecutionContext Interface Methods (Continued)

Method	Description
`public java.lang.String taskDoIt(java.lang.String` *`instanceID`*`, java.lang.String` *`taskID`*`) throws com.bea.wlpi.common.WorkflowException`	Executes a workflow task. The method parameters are defined as follows: ■ *`instanceID`*: `java.lang.String` object that specifies the workflow instance ID. ■ *`taskID`*: `java.lang.String` object that specifies the task ID. This method returns an XML document that is compliant with the Client Request DTD, `ClientReq.dtd`, as described in "DTD Formats" in *Programming BPM Client Applications*. The XML document contains information about the running instance, including the instance ID and template definition ID. It can be accessed by parsing the document using an XML parser, such as a SAX (Simple API for XML) parser. Also refer to the `taskExecute()` method to the `com.bea.wlpi.server.worklist.Worklist` interface, as described in "Managing Run-Time Tasks" in *Programming BPM Client Applications*.
`public java.lang.String taskMarkDone(java.lang.String` *`instanceID`*`, java.lang.String` *`taskID`*`) throws com.bea.wlpi.common.WorkflowException`	Marks a workflow task complete. The method parameters are defined as follows: ■ *`instanceID`*: `java.lang.String` object that specifies the workflow instance ID. ■ *`taskID`*: `java.lang.String` object that specifies the task ID. This method returns an XML document that is compliant with the Client Request DTD, `ClientReq.dtd`, as described in "DTD Formats" in *Programming BPM Client Applications*.The XML document contains information about the running instance, including the instance ID and template definition ID. It can be accessed by parsing the document using an XML parser, such as a SAX (Simple API for XML) parser. Also refer to the `taskMarkDone()` method to the `com.bea.wlpi.server.worklist.Worklist` interface, as described in "Managing Run-Time Tasks" in *Programming BPM Client Applications*.

Table 4-22 ExecutionContext Interface Methods (Continued)

Method	Description
`public java.lang.String taskSetProperties(java.lang.String` *`instanceID`*`, java.lang.String` *`taskID`*`, int` *`priority`*`, boolean` *`doneWithoutExecute`*`, boolean` *`executeIfDone`*`, boolean` *`unmarkDone`*`, boolean` *`modifiable`*`, boolean` *`reassignment`*`) throws com.bea.wlpi.common.WorkflowException`	Sets the properties for a workflow task. The method parameters are defined as follows: ■ *`instanceID`*: `java.lang.String` object that specifies the workflow instance ID. ■ *`taskID`*: `java.lang.String` object that specifies the task ID. ■ *`priority`*: Integer value that specifies the task priority: `0` (low), `1` (medium), or `2` (high). ■ *`doneWithoutExecute`*: Boolean value that specifies whether or not a user has permission to mark a task as complete using the `taskMarkDone()` method, described previously in this table. ■ *`executeIfDone`*: Boolean value that specifies whether or not a user has permission to execute a task using the `taskDoIt()` method, described previously in this table. ■ *`unmarkDone`*: Boolean value that specifies whether or not a user has permission to mark a task as incomplete using the `taskUnmarkDone()` method, described later in this table. ■ *`modifiable`*: Boolean value that specifies whether or not a user has permission to set properties using this method. ■ *`reassignable`*: Boolean value that specifies whether or not a user has permission to assign a task using the `taskAssign()` method, described previously in this table. This method returns an XML document that is compliant with the Client Request DTD, `ClientReq.dtd`, as described in "DTD Formats" in *Programming BPM Client Applications*. The XML document contains information about the running instance, including the instance ID and template definition ID. It can be accessed by parsing the document using an XML parser, such as a SAX (Simple API for XML) parser. Also refer to the `taskSetProperties()` method to the `com.bea.wlpi.server.worklist.Worklist` interface, as described in described in "Managing Run-Time Tasks" in *Programming BPM Client Applications*.

Table 4-22 ExecutionContext Interface Methods (Continued)

Method	Description
`public java.lang.String taskUnassign(java.lang.String` *`instanceID`*`, java.lang.String` *`taskID`*`) throws com.bea.wlpi.common.WorkflowException`	Unassigns a workflow task. The method parameters are defined as follows: ■ *`instanceID`*: `java.lang.String` object that specifies the workflow instance ID. ■ *`taskID`*: `java.lang.String` object that specifies the task ID. This method returns an XML document that is compliant with the Client Request DTD, `ClientReq.dtd`, as described in "DTD Formats" in *Programming BPM Client Applications*. The XML document contains information about the running instance, including the instance ID and template definition ID. It can be accessed by parsing the document using an XML parser, such as a SAX (Simple API for XML) parser. Also refer to the `taskUnassign()` method to the `com.bea.wlpi.server.worklist.Worklist` interface, as described in "Managing Run-Time Tasks" in *Programming BPM Client Applications*.
`public java.lang.String taskUnmarkDone(java.lang.String` *`instanceID`*`, java.lang.String` *`taskID`*`) throws com.bea.wlpi.common.WorkflowException`	Marks a workflow task as incomplete. The method parameters are defined as follows: ■ *`instanceID`*: `java.lang.String` object that specifies the workflow instance ID. ■ *`taskID`*: `java.lang.String` object that specifies the task ID. This method returns an XML document that is compliant with the Client Request DTD, `ClientReq.dtd`, as described in "DTD Formats" in *Programming BPM Client Applications*. The XML document contains information about the running instance, including the instance ID and template definition ID. It can be accessed by parsing the document using an XML parser, such as a SAX (Simple API for XML) parser. Also refer to the `taskUnmarkdone()` method to the `com.bea.wlpi.server.worklist.Worklist` interface, as described in "Managing Run-Time Tasks" in *Programming BPM Client Applications*.

For more information, see the `com.bea.wlpi.server.common.ExecutionContext` Javadoc.

PluginPanelContext

The `com.bea.wlpi.common.plugin.PluginPanelContext` interface provides the run-time context and services for the design client (for example, the Studio), including:

- Access to plug-in template and template definition data
- Services to launch the Expression Builder, validate and manipulate expressions, and invoke the Add Variable dialog box.

Note: Not all methods are applicable in all dialog box contexts. If the plug-in invokes a method in an invalid context, a `java.lang.UnsupportedOperationException` exception is thrown.

You can access the plug-in panel context using the `com.bea.wlpi.common.plugin.PluginPanel` get and set methods defined in the table "PluginPanel Class Methods" on page 4-26.

The following table describes the `PluginPanelContext` interface methods that you can use to access information about the action context.

Table 4-23 PluginPanelContext Interface Methods

Method	Description
`public com.bea.wlpi.common.VariableInfo checkVariable(java.lang.String` *`name`*`, java.lang.String[]` *`validTypes`*`) throws com.bea.wlpi.common.WorkflowException`	Determines whether a variable exists. If not variable exists, this method invokes the Add Variable dialog box to enable a user to define a new workflow variable of a valid type specified by the caller. The method parameters are defined as follows: ■ *`name`*: `java.lang.String` object that specifies the variable name. ■ *`validTypes`*: Array of `java.lang.String` object that specifies either valid types for the specified variable, or null. For a list of valid types, see the `com.bea.wlpi.common.VariableInfo` Javadoc. This method returns a `com.bea.wlpi.common.VariableInfo` object that specifies the existing or new variable information, or null.
`public javax.naming.Context getInitialContext()`	Gets the design client JNDI context. This method returns a `javax.naming.Context` object that specifies the JNDI context. This context contains the same security context used by the design client. **Note:** The caller must not close this context.
`public com.bea.wlpi.common.PluginData getPluginTemplateData(java.lang.String` *`pluginName`*`)`	Gets the plug-in data for the template associated with the specified template. The method parameter is defined as follows. *`pluginName`*: `java.lang.String` object that specifies the plug-in name. This method returns a `com.bea.wlpi.common.plugin.PluginData` object that specifies the plug-in template.

Table 4-23 PluginPanelContext Interface Methods (Continued)

Method	Description
`public com.bea.wlpi.common.PluginData getPluginTemplateDefinitionData(java.lang.String` *`pluginName`*`)`	Gets the plug-in data for the template definition associated with the specified template. The method parameter is defined as follows. *`pluginName`*: `java.lang.String` object that specifies the plug-in name. This method returns a `com.bea.wlpi.common.plugin.PluginData` object that specifies the plug-in template definition.
`public int getTemplateDefinitionID()`	Gets the ID of the owner of the template definition. This method returns a `java.lang.String` object that specifies the template definition ID.
`public int getTemplateID()`	Gets the ID of the owner of the template. This method returns a `java.lang.String` object that specifies the template ID.
`public java.util.List getVariableList()`	Gets a list of variables and their types. This method returns a `java.util.List` object containing a list of `com.bea.wlpi.common.VariableInfo` objects that describe the variables.
`public java.util.List getVariableList(java.lang.String` *`type`*`)`	Gets a list of variables of the specified type. This method returns a `java.util.List` object containing a list of `com.bea.wlpi.common.VariableInfo` objects that describe the variables of the specified type.
`public com.bea.wlpi.common.VariableInfo invokeAddVariableDialog() throws com.bea.wlpi.common.WorkflowException`	Invokes the Add Variable dialog box to enable a user to define a new workflow variable. This method returns a `com.bea.wlpi.common.VariableInfo` object that specifies the new variable information, or null if the OK button was not selected to create the variable.

Table 4-23 PluginPanelContext Interface Methods (Continued)

Method	Description
`public com.bea.wlpi.common.VariableInfo invokeAddVariableDialog(java.lang.String` *`name`*`, java.lang.String[]` *`validTypes`*`) throws com.bea.wlpi.common.WorkflowException`	Invokes the Add Variable dialog box to enable a user to define a new workflow variable of a valid type specified by the caller. The method parameters are defined as follows: ■ *`name`*: `java.lang.String` object that specifies the variable name. ■ *`validTypes`*: Array of `java.lang.String` object that specifies valid types for the specified variable, or null. For a list of valid types, see the `com.bea.wlpi.common.VariableInfo` Javadoc. This method returns a `com.bea.wlpi.common.VariableInfo` object that specifies the new variable information, or null if the OK button was not selected to create the variable.

Table 4-23 PluginPanelContext Interface Methods (Continued)

Method	Description
`public void invokeExpressionBuilder(javax.swing.text.JTextComponent` *`txtInput`*`, boolean` *`condition`*`, com.bea.wlpi.common.plugin.FieldInfo` *`fieldInfo`*`, java.lang.String[]` *`fields`*`, java.lang.String` *`eventDescriptor`*`)`	Invokes the Expression Builder dialog box. The current expression must be displayed in a `javax.swing.text.JTextComponent` object, or an associated subclass, and the client initializes the Expression Builder from this text component. When the user selects OK in the Expression Builder dialog box, the client validates the expression, closes the dialog box, and updates the text component with the modified expression. The method parameters are defined as follows: ■ *`txtInput`*: `javax.swing.text.JTextComponent` object that specifies the text input component containing the expression. ■ *`condition`*: Boolean value that specifies whether or not to build a conditional expression. ■ *`fieldInfo`*: `com.bea.wlpi.common.plugin.FieldInfo` object that specifies any plug-in field information. This parameter is required if the expression is permitted to reference fields. ■ *`fields`*: Array of `java.lang.String` objects that represent valid plug-in field names that match those available to the component specified by the event descriptor. Field types that require qualifiers are not well suited to being specified with this parameter. ■ *`eventDescriptor`*: `java.lang.String` object that specifies the event descriptor.

Table 4-23 PluginPanelContext Interface Methods (Continued)

Method	Description
`public boolean isVariableInExpression(java.lang.String expr, java.lang.String var) throws com.bea.wlpi.evaluator.EvaluatorExpression`	Determines whether or not an expression references the specified variable. The method parameters are defined as follows: ■ *`expr`*: `java.lang.String` object that specifies the expression. ■ *`var`*: `java.lang.String` object that specifies the variable name. This method returns `true` if the variable is referenced and `false` otherwise.
`public java.lang.String renameVariableInExpression(java.lang.String expr, java.lang.String oldName, java.lang.String newName) throws com.bea.wlpi.evaluator.EvaluatorExpression`	Updates the expression references to a renamed variable. The method parameters are defined as follows: ■ *`expr:`* `java.lang.String` object that specifies the expression. ■ *`oldName`*: `java.lang.String` object that specifies the old variable name. ■ *`newName`*: `java.lang.String` object that specifies the new variable name. This method returns a `java.lang.String` object that specifies the updated expression text.

Table 4-23 PluginPanelContext Interface Methods (Continued)

Method	Description
`public java.lang.String validateExpression(java.lang.String` *`expression`*`, boolean` *`allowVariables`*`, com.bea.wlpi.common.plugin.FieldInfo` *`fieldInfo`*`, java.lang.String` *`eventDescriptor`*`) throws com.bea.wlpi.evaluator.EvaluatorExpression`	Updates the expression references to a renamed variable. The method parameters are defined as follows: ■ *`expression`*: `java.lang.String` object that specifies the text of the expression to be validated. ■ *`allowVariables`*: Boolean value that specifies whether or not the expression allows variables. ■ *`fieldInfo`*: `com.bea.wlpi.common.plugin.FieldInfo` object that specifies the plug-in field type. ■ *`eventDescriptor`*: `java.lang.String` object that specifies the event descriptor. This method returns a `java.lang.String` object that specifies the updated expression text.

For more information, see the `com.bea.wlpi.server.plugin.PluginPanelContext` Javadoc.

Defining the Plug-In Component Value Objects

The final step consists of defining the value object for the plug-in component to further define the component data. To define the plug-in component value object, use the associated constructor. Each of the plug-in value objects described in the table "Plug-In Value Objects" on page 2-5, provides one or more constructors for creating object data. The constructors for creating value objects are described in "Plug-In Value Object Summary" on page B-1.

You must pass the plug-in value objects for each of the plug-in components when defining the `com.bea.wlpi.common.plugin.PluginCapabilitiesInfo` object, as described in the `getPluginCapabilities()` method description, in the table "Remote Interface Plug-In Information Methods" on page 3-10.

The following code listing is an excerpt from the plug-in sample that shows how to implement the `getPluginCapabilitiesInfo()` method. This excerpt is taken from the `SamplePluginBean.java` file in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory. Notable lines of code are shown in **bold**.

Listing 4-24 Implementing the getPluginCapabilitiesInfo() Method

```
public PluginCapabilitiesInfo getPluginCapabilitiesInfo(Locale lc,
        CategoryInfo[] info) {

    PluginInfo pi;
    FieldInfo orderFieldInfo;
    FieldInfo confirmFieldInfo;
    FieldInfo[] fieldInfo;
    FunctionInfo fi;
    FunctionInfo[] functionInfo;
    StartInfo si;
    StartInfo[] startInfo;
    EventInfo ei;
    EventInfo[] eventInfo;
    SampleBundle bundle = new SampleBundle(lc);

    log("getPluginCapabilities called");
    pi = createPluginInfo(lc);
    orderFieldInfo =
        new FieldInfo(SamplePluginConstants.PLUGIN_NAME, 3,
                      bundle.getString("orderFieldName"),
                      bundle.getString("orderFieldDesc"),
                      SamplePluginConstants.ORDER_FIELD_CLASSES, false);
    confirmFieldInfo =
        new FieldInfo(SamplePluginConstants.PLUGIN_NAME, 4,
                      bundle.getString("confirmFieldName"),
                      bundle.getString("confirmFieldDesc"),
                      SamplePluginConstants.CONFIRM_FIELD_CLASSES, false);
    fieldInfo = new FieldInfo[]{ orderFieldInfo, confirmFieldInfo };
    ei = new EventInfo(SamplePluginConstants.PLUGIN_NAME, 6,
                       bundle.getString("confirmOrderName"),
                      bundle.getString("confirmOrderDesc"), ICON_BYTE_ARRAY,
                       SamplePluginConstants.EVENT_CLASSES,
                       confirmFieldInfo);
    eventInfo = new EventInfo[]{ ei };
    fi = new FunctionInfo(SamplePluginConstants.PLUGIN_NAME, 7,
                          bundle.getString("calcTotalName"),
                          bundle.getString("calcTotalDesc"),
                          bundle.getString("calcTotalHint"),
                          SamplePluginConstants.FUNCTION_CLASSES, 3, 3);
    functionInfo = new FunctionInfo[]{ fi };
    si = new StartInfo(SamplePluginConstants.PLUGIN_NAME, 5,
                       bundle.getString("startOrderName"),
                    bundle.getString("startOrderDesc"), ICICON_BYTE_ARRAYON,
                       SamplePluginConstants.START_CLASSES, orderFieldInfo);
    startInfo = new StartInfo[]{ si };

    PluginCapabilitiesInfo pci = new PluginCapabilitiesInfo(pi,
                                     getCategoryInfo(bundle), eventInfo,
                                     fieldInfo, functionInfo, startInfo,
                                     null, null, null, null, null);

    return pci;
}
```

CHAPTER

5 Using Plug-In Notifications

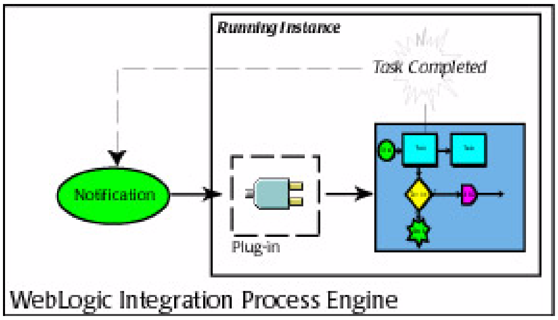
This section explains how to use plug-in notifications. It includes the following topics:

- Overview
- Registering the Plug-In as a Notification Listener
- Getting Information About a Received Notification
- Unregistering the Plug-In as a Notification Listener

Overview

BPM communicates with plug-ins by broadcasting *notifications*. A notification is a message that the WebLogic Integration process engine sends to a plug-in, indicating that an event has occurred, as illustrated in the following figure.

Figure 5-1 Notification Sent by WebLogic Integration Process Engine to Plug-In



You can register a plug-in as a notification listener to receive up to four types of notifications, provided as part of the `com.bea.wlpi.server.plugin` package.

Note: To minimize the impact on performance, only register a plug-in for notifications that it absolutely needs to receive. In particular, use caution when registering to receive run-time instance and task notifications.

The following table defines:

- Four types of notifications
- Related events that cause a notification to be sent
- Integer values in the `com.bea.wlpi.common.plugin.PluginConstants` interface that correspond to each event. You can specify these values when registering the plug-in as a notification listener, as described in “Registering the Plug-In as a Notification Listener” on page 5-4.

Table 5-1 Notification Types and Related Events

Notification Type	Description	Related Events	PluginConstants Value
`InstanceNotification`	Workflow instance notification	Workflow instance is aborted	`INSTANCE_ABORTED`
		Workflow instance is completed	`INSTANCE_COMPLETED`
		Workflow instance is created	`INSTANCE_CREATED`
		Workflow instance is deleted	`INSTANCE_DELETED`
		Workflow instance is updated	`INSTANCE_UPDATED`
`TaskNotification`	Task notification	Task is assigned	`TASK_ASSIGNED`
		Task is completed	`TASK_COMPLETED`
		Task is executed	`TASK_EXECUTED`
		Task is overdue	`TASK_OVERDUE`
		Task is started	`TASK_STARTED`
		Task is unassigned	`TASK_UNASSIGNED`
		Task is unmarked as done	`TASK_UNMARKED_DONE`
		Task is updated	`TASK_UPDATED`
`TemplateDefinitionNotification`	Template definition notification	Template definition is created	`DEFINITION_CREATED`
		Template definition is deleted	`DEFINITION_DELETED`
		Template definition is updated	`DEFINITION_UPDATED`

Table 5-1 Notification Types and Related Events (Continued)

Notification Type	Description	Related Events	PluginConstants Value
`TemplateNotification`	Template notification	Template is created	`TEMPLATE_CREATED`
		Template is deleted	`TEMPLATE_DELETED`
		Template is updated	`TEMPLATE_UPDATED`

The following sections describe how to register and unregister the plug-in as a notification listener, and get information about a received notification.

Registering the Plug-In as a Notification Listener

Typically, the plug-in should register itself as a notification listener at load time. For example, when implementing the `load()` method, as described in "Lifecycle Management Methods" on page 3-7, you should register the plug-in as a notification listener, if necessary.

To register the plug-in as a notification listener, use one (or more) of the `com.bea.wlpi.server.plugin.PluginManagerCfg` methods defined in the following table, based on the type of notifications that you want the plug-in to receive.

Note: For information about connecting to the Plug-in Manager, see "Connecting to the Plug-In Manager" on page 2-2.

Table 5-2 Plug-In Notification Registration Methods

To register the plug-in as . . .	Use the following method . . .
`InstanceNotification` listener	`public void addInstanceListener( com.bea.wlpi.server.plugin.Plugin` *`plugin`*`, int` *`mask`*`) throws java.rmi.RemoteException`

Table 5-2 Plug-In Notification Registration Methods (Continued)

To register the plug-in as . . .	Use the following method . . .
`TaskNotification` listener	`public void addTaskListener( com.bea.wlpi.server.plugin.Plugin` *`plugin`*`, int` *`mask`*`) throws java.rmi.RemoteException`
`TemplateDefinitionNotification` listener	`public void addTemplateDefinitionListener( com.bea.wlpi.server.plugin.Plugin` *`plugin`*`, int` *`mask`*`) throws java.rmi.RemoteException`
`TemplateNotification` listener	`public void addTemplateListener( com.bea.wlpi.server.plugin.Plugin` *`plugin`*`, int` *`mask`*`) throws java.rmi.RemoteException`

The following table defines the parameters for which you must specify values for the plug-in notification registration method.

Table 5-3 Plug-In Notification Registration Method Parameters

Parameter	Description
`plugin`	EJBObject implementing the plug-in remote interface.
`mask`	Integer bitmask specifying the events for which you want to register. For a list of valid values, see the notification `com.bea.wlpi.common.plugin.PluginConstants` values defined in the table “Notification Types and Related Events” on page 5-3. The `mask` value can also be set to `EVENT_NOTIFICATION_ALL` or `EVENT_NOTIFICATION_NONE` to globally register or unregister, respectively, all events in a particular category. This value is formed by performing a bitwise OR of the required event constants.

For more information about notification listener registration methods, see the `com.bea.wlpi.server.plugin.PluginManagerCfg` Javadoc.

The following code listing is an excerpt from the plug-in sample that shows how to register the plug-in as a notification listener for *all notification types* and *all of their related events*. This excerpt is taken from the `SamplePluginBean.java` file in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory. Notable lines of code are shown in **bold**.

Listing 5-1 Registering the Plug-In as a Notification Listener

```
      .
      .
      .
public void load(PluginObject pluginData) throws PluginException {
      .
      .
      .
   pm.addInstanceListener(plugin, PluginConstants.EVENT_NOTIFICATION_ALL);
   pm.addTaskListener(plugin, PluginConstants.EVENT_NOTIFICATION_ALL);
   pm.addTemplateDefinitionListener(plugin,
      PluginConstants.EVENT_NOTIFICATION_ALL);
   pm.addTemplateListener(plugin, PluginConstants.EVENT_NOTIFICATION_ALL);
      .
      .
      .
    }
```

For more information about the plug-in sample, see “BPM Plug-In Sample” on page 10-1.

Getting Information About a Received Notification

Once a notification is received, you can get information about it, including its source.

The following table defines methods available for getting information about any received notification. These methods can be accessed via the `com.bea.wlpi.server.plugin.PluginNotification` class which all notification types extend.

Table 5-4 General Plug-In Notification Information Methods

Method	Description
`public int getEventType()`	Gets the event type. This method returns a `com.bea.wlpi.common.plugin.PluginConstants` interface integer value corresponding to the related event, as defined in the table “Notification Types and Related Events” on page 5-3.
`public java.lang.Object getSource()`	Gets the workflow entity object that has changed. This method returns a `java.lang.Object` object specifying the workflow entity that has changed, and can be any of the following `com.bea.wlpi.common.plugin` value objects: `InstanceInfo`, `TaskInfo`, `TemplateDefinitionInfo`, or `TemplateInfo`. For information about the methods available for getting information about each value object, see “Plug-In Value Object Summary” on page B-1.

For more information about these methods, see the `com.bea.wlpi.server.plugin.PluginNotification` Javadoc.

In addition, the following table defines the methods that are available to each notification type to obtain addition information about the source component that has changed.

Table 5-5 Plug-In Notification Information Methods Based on Notification Type

Notification Type	Method	Description
`InstanceNotification`	`public com.bea.wlpi.common.InstanceInfo getInstance()`	Gets information about the workflow instance that has changed. This method returns a `com.bea.wlpi.common.InstanceInfo` object. To access information about the workflow instance, use the `InstanceInfo` object methods described in "InstanceInfo Object" in "Value Object Summary" in *Programming BPM Client Applications*.
`TaskNotification`	`public com.bea.wlpi.common.TaskInfo getTask()`	Gets information about the task that has changed. This method returns a `com.bea.wlpi.common.TaskInfo` object. To access information about the task, use the `TaskInfo` object methods described in "TaskInfo Object" in "Value Object Summary" in *Programming BPM Client Applications*.
`TemplateDefinitionNotification`	`public com.bea.wlpi.common.TemplateDefinitionInfo getTemplateDefinition()`	Gets information about the template definition that has changed. This method returns a `com.bea.wlpi.common.TemplateDefinitionInfo` object. To access information about the template definition, use the `TemplateDefinitionInfo` object methods described in "TemplateDefinitionInfo Object" in "Value Object Summary" in *Programming BPM Client Applications*.

Table 5-5 Plug-In Notification Information Methods Based on Notification Type (Continued)

Notification Type	Method	Description
`TemplateNotification`	`public com.bea.wlpi.common.TemplateInfo getTemplate()`	Gets information about the template that has changed. This method returns a `com.bea.wlpi.common.TemplateInfo` object. To access information about the template, use the `TemplateInfo` object methods described in "TemplateInfo Object" in "Value Object Summary" in *Programming BPM Client Applications*.

Unregistering the Plug-In as a Notification Listener

Typically, the plug-in should unregister itself as a notification listener at unload time. For example, when implementing the `unload()` method, as described in "Lifecycle Management Methods" on page 3-7, you should unload the plug-in as a notification listener, if necessary.

To unregister the plug-in as a notification listener, use one (or more) of the `com.bea.wlpi.server.plugin.PluginManagerCfg` methods defined in the following table, based on the type of notification listener that you want to remove.

Note: For information about connecting to the Plug-in Manager, see "Connecting to the Plug-In Manager" on page 2-2.

Table 5-6 Plug-In Notification Unregistration Methods

To unregister the plug-in as . . .	Use the following method . . .
`InstanceNotification` listener	`public void removeInstanceListener( com.bea.wlpi.server.plugin.Plugin plugin) throws java.rmi.RemoteException`
`TaskNotification` listener	`public void removeTaskListener( com.bea.wlpi.server.plugin.Plugin plugin) throws java.rmi.RemoteException`
`TemplateDefinitionNotification` listener	`public void removeTemplateDefinitionListener( com.bea.wlpi.server.plugin.Plugin plugin) throws java.rmi.RemoteException`
`TemplateNotification` listener	`public void removeTemplateListener( com.bea.wlpi.server.plugin.Plugin plugin) throws java.rmi.RemoteException`

In each case, you must specify a `com.bea.wlpi.server.plugin.Plugin` object that identifies the plug-in that you are unregistering as a notification listener.

The following code listing shows how to unregister the plug-in as a notification listener for *all notification types* and *all of their related events*. Notable lines of code are shown in **bold**.

Listing 5-2 Unregistering the Plug-In as a Notification Listener

```
public void unload() throws PluginException {
      .
      .
      .
   pm.removeInstanceListener(plugin);
   pm.removeTaskListener(plugin);
   pm.removeTemplateDefinitionListener(plugin);
   pm.removeTemplateListener(plugin);
      .
      .
    }
```

CHAPTER

6 Processing Plug-In Events

This section explains how to process plug-in events. It includes the following topics:

- Overview of Plug-In Events
- EventData Class
- Defining the Plug-In Event Handler
- Defining Plug-In Message Types
- Defining an Event Watch Entry
- Sending an Event to the Plug-In Event Handler

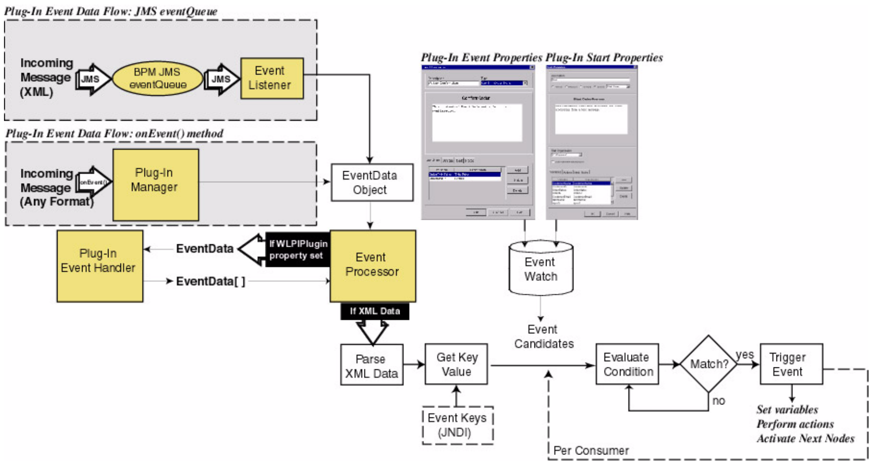
Overview of Plug-In Events

An *event* is an asynchronous notification from another workflow or from an external source, such as another application. You can define an event to start a workflow, initialize variables, activate a node in the workflow, or execute an action. You define events and event properties when defining Start and Event nodes using the WebLogic Integration Studio. For more information about defining events and event properties, refer to *Using the WebLogic Integration Studio*.

The BPM framework supports event messages in XML format that are delivered via JMS. To support event messages in both XML and non-XML format, you must define a *plug-in event*.

The following figure illustrates the data flow for plug-in events.

Figure 6-1 Plug-In Event Data Flow



As shown in the previous figure, the plug-in event messages are passed to the system in one of the following two ways:

- JMS eventQueue provided by BPM.

  At run time, the event listener listens for events (via the JMS eventQueue) which are received in XML format and delivered using JMS.

  For more information about setting up JMS event listeners for the eventQueue, see “Establishing JMS Connections” in *Programming BPM Client Applications*.

- `onEvent()` method of the `com.bea.wlpi.server.plugin.PluginManager` EJB.

  At run time, the Plug-in Manager `onEvent()` method accepts any data format specified by the plug-in.

The WebLogic Integration process engine stores an incoming plug-in event message as a `com.bea.wlpi.server.eventprocessor.EventData` object, and passes that object to the Event Processor. For more information about the `EventData` object, see “EventData Class” on page 6-5.

Upon receipt of an `EventData` object, the Event Processor first checks to see if the `WLPIPlugin` property is defined as part of the message.

The `WLPIPlugin` message property can be defined in either of the following ways:

- By defining the *`properties`* constructor parameter to the `com.bea.wlpi.server.eventprocessor.EventData` object to include a property named `WLPIPlugin`, as described in the table "EventData Object Constructors" on page 6-5.
- By defining a JMS message property named `WLPIPlugin`, as described in "Setting Message Property Fields" in "Developing a WebLogic JMS Application" in *Programming WebLogic JMS*, at the following URL:

  `http://e-docs.bea.com/wls/docs61/jms/implement.html`

If the `WLPIPlugin` property is defined, the Event Processor pre-processes the plug-in event data by passing the `EventData` object to the plug-in event handler. The pre-processed event data is returned to the Event Processor as an array of `EventData` objects. For more information about defining the plug-in event handler, see "Defining the Plug-In Event Handler" on page 6-11.

Next, the Event Processor checks to see if the resulting data is in XML format, and, if so, performs the following steps:

1. Parses the event data.

   At this time, the Event Processor retrieves the content type and event descriptor supplied by the `EventData` object.

   The content type refers to the MIME (Multi-Purpose Internet Mail Exensions) content type, and describes the basic data type of the message. The content type defaults to `text/xml`, indicating that the data is in text format and obeys the rules of XML.

   The event descriptor provides a precise definition of the data format, and is interpreted in the context of the content type.

   For example, if the content type is set to `text/xml`, the event descriptor is the XML document type. If the DOCTYPE tag is set, the XML document type is the public ID or system ID, if defined, or the document element name. If, on the other hand, the content type is set to `application/x-java-object` (specifying a serialized Java object), the event descriptor is the fully-qualified Java class name.

2. Retrieves the event key value, if specified, and compares it to the pre-defined event keys in JNDI.

An event key specifies the incoming data field that should be treated as the primary key, and is associated with a particular content type and event descriptor. An event key provides a filtering mechanism for incoming event data to improve performance. For more information about defining event keys, see *Using the WebLogic Integration Studio*.

Note: If a plug-in Event or Start node uses the `activateEvent()` method or the `postStartWatch()` method, respectively, to the `com.bea.wlpi.server.plugin.EventContext` interface, and passes a content type other than `text/xml`, it is the plug-in's responsibility to ensure that a matching event key expression is registered in the event key table. This can be accomplished using the `addEventKey()` method to the `com.bea.wlpi.server.admin.Admin` interface. The plug-in should also provide a plug-in field to evaluate a key value from incoming data of this content type and format. For more information about defining a plug-in field, see "Defining the Run-Time Component Class for a Message Type" on page 4-84.

3. Searches the event watch table for possible candidates to receive the event content based on the content type, event descriptor, and event key value.

   For more information about how plug-in events are added as entries to the event watch table, see "Defining an Event Watch Entry" on page 6-14.

4. Evaluates the condition, if specified, and determines if there is a successful match.

   A condition consists of an expression that evaluates to true or false. A condition enables users to define multiple workflows for the same event data that they want to process differently, based on the content of the message. For example, the following provides a valid condition if the event data contains a field called Order Amount: `"Order Amount > $500.00"`. For more information about defining conditions, see *Using the WebLogic Integration Studio*.

5. In the event of a match, triggers the event to either start a new instance of a workflow or resume processing of an existing workflow instance.

Each of these tasks are performed for *each* consumer subscribed to the event.

To support plug-in events, you must perform the following steps:

1. Define the plug-in event handler to handle plug-in event data.

2. Define the custom message types, or plug-in fields, to parse the event data returned by the event handler.
3. Define the event watch entry.

The following sections describe each of these steps in detail, and explain how to send a message to the plug-in event handler.

Before you can create an event handler, you need to understand how to create and access information about the `EventData` container class, which stores the incoming plug-in event message.

EventData Class

As shown in the figure "Plug-In Event Data Flow" on page 6-2, the `com.bea.wlpi.server.eventprocessor.EventData` object provides the container class for incoming data messages.

The following table defines the constructors that can be used to create an `EventData` object.

Table 6-1 EventData Object Constructors

Constructor	Description
`public EventData(org.w3c.dom.Document` *`document`*`)`	Constructs an `EventData` object from a pre-parsed `DOM` object containing XML, and sets the following values: ■ Content type to `CONTENT_TYPE_DOM.` ■ Event descriptor to the `DOCTYPE` public or system ID, if defined, or the document element name. The constructor parameter is defined as follows. *`document`*: `org.w3c.dom.Document` object that specifies the XML content.

Table 6-1 EventData Object Constructors (Continued)

Constructor	Description
`public EventData(java.lang.String` *`xml`*`) throws com.bea.wlpi.common.WorkflowException`	Constructs an `EventData` object from a String object containing XML, and sets the following values: ■ Content type to `CONTENT_TYPE_XML`. ■ Event descriptor to the `DOCTYPE` public or system ID, if defined, or the document element name. The constructor parameter is defined as follows. *`xml`*: `java.lang.String` object that specifies the XML content.

Table 6-1 EventData Object Constructors (Continued)

Constructor	Description
`public EventData( java.lang.Object` *`content`*`, java.lang.String` *`contentType`*`, java.lang.String` *`eventDescriptor`*`, long` *`expiration`*`, java.lang.String[]` *`templateNames`*`, java.lang.String[]` *`instanceIDs`*`, java.util.Map` *`properties`*`)`	Constructs an `EventData` object containing an object of any type. The constructor parameters are defined as follows: ■ *`content`*: `java.lang.Object` object that specifies the content. ■ *`contentType`*: `java.lang.String` object that specifies the content type. ■ *`eventDescriptor`*: `java.lang.String` object that specifies the event descriptor. ■ *`expiration`*: Long value that specifies the expiration. ■ *`templateNames`*: Array of `java.lang.String` objects that specify the names to which the event data should be delivered, or null. ■ *`instanceIDs`*: `java.lang.String` objects that specify the IDs of the workflow instances to which the event data should be delivered, or null. ■ *`properties`*: `java.util.Map` object that specifies a map of custom message properties. For example, you may want to define any of the following custom message properties: - `WLPIContentType`: Content type. - `WLPIEventDescriptor`: Event descriptor. - `WLPIPlugin`: Plug-in name. **Note:** The `EventData` object also provides the `CONTENT_TYPE`, `EVENT_DESCRIPTOR`, and `PLUGIN` constant values that define the content type, event descriptor, and plug-in name, respectively. If *`templateNames`* or *`instanceIDs`* are non-null, the event data is considered to be addressed.

The following table describes the `EventData` object information, the constructor parameters used to define the information, and the methods that can be used to access that information after the object is defined, if applicable.

Table 6-2 EventData Object Information

Object Information	Constructor Parameter	Get Method
`org.w3c.dom.Document` document (DOM) containing the XML content.	*`document`*	`public final boolean isDOM()`
String document containing the XML content.	*`xml`*	`public final boolean isXML()`
`java.lang.Object` object containing the event data content.	*`content`*	`public final java.lang.Object getContent()`
Content type.	*`contentType`*	`public final java.lang.String getContentType()`
Event descriptor.	*`eventDescriptor`*	`public final java.lang.String getEventDescriptor()`
Expiration.	*`expiration`*	`public final long getExpiration()`
Array of template names to which the event data is addressed.	*`templateNames`*	`public final String[] getTemplateNames()`
Array of workflow instance IDs to which the event data is addressed.	*`instanceIDs`*	`public final java.lang.String[] getInstanceIDs()`

Table 6-2 EventData Object Information (Continued)

Object Information	Constructor Parameter	Get Method

Plug-in message properties.	*properties*	`public java.lang.Object getProperty(java.lang.String propertyName) throws com.bea.wlpi.common.WorkflowException`
		
		`public java.util.Set getPropertyNames() throws com.bea.wlpi.common.WorkflowException`

Unique message identifier assigned by the Event Processor. **Note:** This method only applies to addressed messages. If the event message data is not addressed, this method returns null.	*N/A*	`public final java.lang.String getMessageID()`
Key value of the event data. The Event Processor sets the key value if a key value expression is defined for the content type and event descriptor combination.	*N/A*	`public final java.lang.String getKeyValue()`

The following table lists the methods that can be used to parse the XML data.

Table 6-3 XML Parsing Methods

Method	Description
`public final void parseXML(boolean` *`validate`*`, javax.xml.parsers.DocumentBuilder` *`docBuild`*`) throws com.bea.wlpi.common.WorkflowException`	Parses contents of the event data into an `org.w3c.dom.Document` document (DOM) if the content is XML, and sets the following values: ■ Content type to `#CONTENT_TYPE_DOM.` The method parameters are defined as follows. ■ *`validate`*: True if the XML should be validated against the referenced DTD. ■ *`docBuild`*:: `javax.xml.parsers.DocumentBuilder` object that specifies the document builder to use to parse the XML.
`public final void parseXML(boolean` *`validate`*`) throws com.bea.wlpi.common.WorkflowException`	Parses contents of the event data into an `org.w3c.dom.Document` document (DOM) if the content is XML, and sets the following values: ■ Content type to `#CONTENT_TYPE_DOM.` ■ Event descriptor to the `DOCTYPE` public or system ID, if defined, or the document element name. The method parameter is defined as follows. *`validate`*: True if the XML should be validated against the referenced DTD.

Table 6-3 XML Parsing Methods (Continued)

Method	Description
`public final void parseXML() throws com.bea.wlpi.common.WorkflowException`	Parses contents of the event data into an `org.w3c.dom.Document` object if the content is XML, and sets the following values: ■ Content type to `#CONTENT_TYPE_DOM.` ■ Event descriptor to the `DOCTYPE` public or system ID, if defined, or the document element name. The XML is not validated.

The following section explains how to define the plug-in event handler to pre-process the incoming `EventData` object.

Defining the Plug-In Event Handler

As shown in the figure "Plug-In Event Data Flow" on page 6-2, you must define the plug-in event handler to pre-process plug-in event data, and enable the Event Processor to parse the data.

To define the plug-in event handler, you must perform the following tasks.

1. Define the execution handler component class.
2. Define a `com.bea.wlpi.server.plugin.EventHandlerInfo` value object, passing the name of the event handler object defined in step 1 as an argument to the constructor.
3. Register the event handler by passing the `EventHandlerInfo` object defined in step 2 as a constructor parameter to the plug-in `PluginCapabilitiesInfo` object.

The following sections describe each of these steps in more detail.

Defining the Event Handler Component Class

To define a plug-in event handler component class, implement the `com.bea.wlpi.server.plugin.EventHandler` interface.

The following table defines the `EventHandler` interface method that you must implement to pre-process the incoming plug-in event data.

Table 6-4 EventHandler Interface Method

Method	Description
`public com.bea.wlpi.server.eventprocessor.EventData[] onEvent(com.bea.wlpi.server.eventprocessor.EventData` *`eventData`*`) throws com.bea.wlpi.common.plugin.PluginException`	Pre-processes incoming plug-in event data. The Event Processor calls this method upon receipt of an event message (usually via JMS) that is addressed to a specific plug-in. An event handler can translate a single incoming event into multiple outgoing events to be handled sequentially by the Event Processor. The method parameter is defined as follows. *`eventData`*: `com.bea.wlpi.server.eventprocessor.EventData` object that specifies the plug-in event data. This method returns one of the following values: ■ One or more `EventData` objects containing the converted event data, with the appropriate content type and attributes set. ■ Null, if the event handler fully processes the incoming plug-in event data, and no further processing is required by the Event Processor.

Creating an Event Handler Value Object

To create an event handler value object, `com.bea.wlpi.common.plugin.EventHandlerInfo`, use the constructor defined in "EventHandlerInfo Object" on page B-17. You must pass the name of the event handler component class, defined in the previous section, as the *`classNames`* constructor parameter value.

For example, the following code excerpt creates a new event handler value object, passing the name of the event handler class, `sample.MyEventHandler`. The event handler class implements the `EventHandler` `com.bea.wlpi.server.plugin.EventHandler` interface, as described in "Defining the Event Handler Component Class" on page 6-12.

```
eh = new EventHandlerInfo(SamplePluginConstants.PLUGIN_NAME,
        bundle.getString("startOrderName"),
        bundle.getString("startOrderDesc")
        sample.MyEventHandler);
eventHandler = new EventHandlerInfo[]{ eh };
```

For more information about the `EventHandlerInfo` object, see "EventHandlerInfo Object" on page B-17.

Registering an Event Handler

To register the event handler, create a `com.bea.wlpi.common.plugin.PluginCapabilitiesInfo` value object, as defined in "PluginCapabilitiesInfo Object" on page B-29, and pass the `EventHandlerInfo` object (defined in the previous section) as the *`eventHandler`* constructor parameter value.

For example, the following code excerpt creates a new `PluginCapabilitiesInfo` value object, passing the name of the `com.bea.wlpi.common.plugin.EventHandlerInfo` object as an argument.

```
PluginCapabilitiesInfo pci = new PluginCapabilitiesInfo(pi,
      getCategoryInfo(bundle), eventInfo, fieldInfo, functionInfo,
      startInfo, null, null, null, null, eventHandler);
```

For more information about the `PluginCapabilitiesInfo` object, see "PluginCapabilitiesInfo Object" on page B-29.

Defining Plug-In Message Types

You must define plug-in message types, or *plug-in fields*, to enable the process engine to extract information from the plug-in `com.bea.wlpi.server.eventprocessor.EventData` objects. The extracted values can be assigned to workflow variables or used as part of an expression, such as in key value or conditional expressions.

For more information about defining plug-in message types, see "Defining the Run-Time Component Class for a Message Type" on page 4-84.

Defining an Event Watch Entry

An event watch entry enables the Event Processor to match an incoming event to the plug-in node.

The following table describes how the process engine adds plug-in event entries to the event watch table for each plug-in node type.

Table 6-5 How Event Watch Entries Are Added Based on Plug-In Node Type

Node Type	Description
Event node	When a workflow processor activates the node, the Plug-in Framework calls the `activate()` method on the `com.bea.wlpi.server.plugin.PluginEvent` object. The `activate()` method records an entry in the event watch table by calling the `activateEvent()` method on the `com.bea.wlpi.server.plugin.EventContext`. For more information about implementing the `activate()` method, see the table "PluginEvent Interface Methods" on page 4-75. For more information about the `EventContext`, see "Event Context" on page 4-102.

Table 6-5 How Event Watch Entries Are Added Based on Plug-In Node Type (Continued)

Node Type	Description
Start node	When a user sets a template definition to active, the Plug-in Framework calls the `activate()` method on the `com.bea.wlpi.server.plugin.PluginStart2` object, which in turn calls the `setTrigger()` method. The `setTrigger()` method records an entry in the event watch table by calling the `postStartWatch()` method on the `com.bea.wlpi.server.plugin.EventContext`. For more information about implementing the `setTrigger()` method, see the table “PluginStart2 Interface Method” on page 4-90. For more information about the `EventContext`, see “Event Context” on page 4-102.

Sending an Event to the Plug-In Event Handler

To send an event to the plug-in event handler, the message sender must set the `com.bea.wlpi.server.eventprocessor.EventData` object `WLPIPlugin` string property to the event handler name.

For example, the following code segment demonstrates how to build a properties map, setting the `WLPIPlugin` property to the name of the event handler, `SamplePlugin`, and passing this information to the `EventData` constructor.

```
Map props = new HashMap();
props.put("WLPIPlugin",“SamplePlugin”);
EventData eventData = new EventData(data,
  “text/x-appplication/sample”,
  “Order”, 0, null, null, props);
```

CHAPTER

7 Managing Plug-Ins

This section explains how to manage plug-ins. It includes the following topics:

- Viewing Plug-Ins
- Loading Plug-Ins
- Configuring Plug-Ins
- Refreshing the List of Plug-Ins
- Using the Studio to Manage Plug-ins

Viewing Plug-Ins

To view installed plug-ins, use the `com.bea.wlpi.server.plugin.PluginManager` interface methods defined in the following table.

Table 7-1 PluginManager Interface Methods for Viewing Installed Plug-Ins

Method	Description
`public com.bea.wlpi.common.plugin.PluginInfo getPlugin(java.lang.String` *`pluginName`*`, java.util.Locale` *`lc`*`) throws java.rmi.RemoteException, com.bea.wlpi.common.WorkflowException`	Gets the localized, basic information for the specified plug-in. The method parameters are defined as follows. ■ *`pluginName`*: `java.lang.String` object that specifies the plug-in name. ■ *`lc`*: `java.util.Locale` object that specifies a locale in which to localize display strings. This method returns a `com.bea.wlpi.common.plugin.PluginInfo` object that specifies the basic plug-in information. For more information, see "PlugInfo Object" on page B-33.
`public com.bea.wlpi.common.plugin.PluginInfo[] getPlugins(java.util.Locale` *`lc`*`) throws java.rmi.RemoteException, com.bea.wlpi.common.WorkflowException`	Gets a list of installed plug-ins for the specified locale. The method parameter is defined as follows. *`lc`*: `java.util.Locale` object that specifies a locale in which to localize display strings. This method returns an array of `com.bea.wlpi.common.plugin.PluginInfo` objects that specify the installed plug-ins. For more information, see "PlugInfo Object" on page B-33.

For example, the following code gets a list of installed plug-ins for the specified locale, `lc`, and saves the list to the `plugins[]` array. In this example, `pm` represents the `EJBObject` reference to the `PluginManager` EJB.

```
plugins[]=pm.getPlugins(lc);
```

For more information about the `getPlugin()` and `getPlugins()` methods, see the `com.bea.wlpi.server.plugin.PluginManager` Javadoc.

Loading Plug-Ins

To load an installed plug-in, use the `com.bea.wlpi.server.plugin.PluginManagerCfg` interface method defined in the following table.

Table 7-2 PluginManager Interface Method for Loading an Installed Plug-In

Method	Description
`public void loadPlugin(java.lang.String` *`pluginName`*`, com.bea.wlpi.commonVersionInfo` *`version`*`) throws java.rmi.RemoteException, com.bea.wlpi.common.WorkflowException`	Loads and initializes the specified plug-in. The Plug-in Manager merges the capabilities of the plug-in with those of the currently loaded plug-ins. The method parameters are defined as follows. ■ *`pluginName`*: `java.lang.String` object that specifies the plug-in name. ■ *`version`*: `com.bea.wlpi.common.VersionInfo` object that specifies the Plug-in Manager version. For information about accessing the Plug-in Manager version, see "Getting the Plug-In Framework Version" on page 2-4.

For example, the following code loads the plug-in, `MyPlugin`, for the specified Plug-in Manager version, `version`. In this example, `pmCfg` represents the `EJBObject` reference to the `PluginManager` EJB.

```
pmCfg.getPlugins(MyPlugin, version);
```

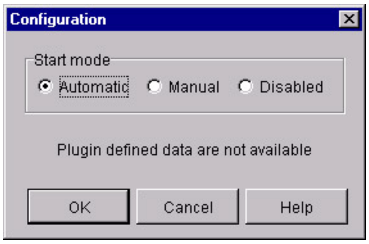
For more information about the `loadPlugin()` method, see the `com.bea.wlpi.server.plugin.PluginManagerCfg` Javadoc.

Configuring Plug-Ins

By default, when configuring the plug-in you can specify the plug-in start mode.

For example, the following figure shows the default plug-in configuration dialog in the WebLogic Intergration Studio.

Figure 7-1 Default Plug-In Configuration Dialog in the Studio



Start mode: [✓ Automatic] Manual Disabled

Plugin defined data are not available

The start mode can be set to one of the following values:

- `Automatic`: started automatically at system startup. This is the default.
- `Manual`: must be started manually by a user.
- `Disabled`: disabled and cannot be started.

For information about accessing the plug-in configuration dialog, see "Configuring Plug-Ins" in "Configuring Workflow Resources" in *Using the WebLogic Integration Studio*.

If required, you can customize the plug-in configuration requirements. The customized plug-in configuration requirements appear in place of the text `Plugin defined data are not available` shown in the previous figure.

The following sections explain how to customize plug-in configuration requirements and edit configuration values using the plug-in API, including the following topics:

- Customize the Plug-In Configuration Requirements
- Setting the Plug-In Configuration Values
- Getting the Plug-In Configuration Values
- Deleting the Plug-In Configuration Values

Customize the Plug-In Configuration Requirements

The following sections describe the steps required to customize the plug-in configuration requirements, including the following topics:

- Implementing the PluginData Interface
- Defining the PluginPanel Class
- Defining the ConfigurationInfo Value Object

Implementing the PluginData Interface

To read (parse) and save plug-in data in XML format, implement the plug-in data interface, as described in "Implementing the PluginData Interface" on page 4-10.

The following code listing shows how to define a class that implements the `PluginData` interface for the plug-in configuration. Notable lines of code are shown in **bold**.

Note: This class is not available as part of the plug-in sample.

Listing 7-1 Implementing the PluginData Interface—Plug-In Configuration

```
package com.bea.wlpi.test.plugin;

import java.io.IOException;
import com.bea.wlpi.common.plugin.PluginData;
import com.bea.wlpi.common.XMLWriter;
import java.util.List;
import java.util.Map;
import org.xml.sax.*;

public class ConfigData implements PluginData {
    private static final String YESORNO_TAG = "yesorno";

    private String yesOrNo;
    private transient String lastValue;

    public ConfigData() {
        this.yesOrNo = TestPluginConstants.CONFIG_NO;
    }
    public ConfigData(String yesOrNo) {
        this.yesOrNo = yesOrNo;
    }

    public void load(XMLReader parser) {
    }

    void setYesOrNo(String decision) {
        yesOrNo = decision;
    }

    String getYesOrNo() {
        return yesOrNo;
    }

    public void setDocumentLocator(Locator locator) {
    }

    public void startDocument()
        throws SAXException {
    }

    public void endDocument()
        throws SAXException {
    }

    public void startPrefixMapping(String prefix, String uri)
        throws SAXException {
    }

    public void endPrefixMapping(String prefix)
        throws SAXException {
    }

   public void startElement(String namespaceURI, String localName, String qName,
Attributes atts)
        throws SAXException {
        lastValue = null;
    }

    public void endElement(String namespaceURI, String localName, String name)
        throws SAXException {
        if (name.equals(YESORNO_TAG))
            yesOrNo = lastValue;
    }

    public void characters(char[] ch, int start, int length)
        throws SAXException {
        String value = new String(ch, start, length);
        if (lastValue == null)
            lastValue = value;
        else
            lastValue = lastValue + value;
    }

    public void ignorableWhitespace(char[] ch, int start, int length)
        throws SAXException {
    }

    public void processingInstruction(String target, String data)
        throws SAXException {
    }

    public void skippedEntity(String name)
        throws SAXException {
    }

    public void save(XMLWriter writer, int indent) throws IOException {
        writer.saveElement(indent, YESORNO_TAG, yesOrNo);
    }

    // TODO:
    public List getReferencedPublishables(Map publishables) {
        return null;
    }

    public String getPrintableData() {
        return toString();
    }

    public String toString() {
        return "ConfigData[yesOrNo=" + yesOrNo + ']';
    }

    public Object clone() {
        return new ConfigData(yesOrNo);
    }
}
```

Defining the PluginPanel Class

To display the plug-in GUI component within the design client, define a class that extends the plug-in panel class, as described in “Defining the PluginPanel Class” on page 4-25.

The following code listing shows how to define a class for the plug-in configuration that extends the `PluginPanel` class. Notable lines of code are shown in **bold**.

Note: This class is not available as part of the plug-in sample.

Listing 7-2 Defining the PluginPanel Class—Plug-In Configuration

```
package com.bea.wlpi.test.plugin;

import java.awt.*;
import javax.swing.*;
import javax.swing.border.TitledBorder;
import javax.swing.border.EtchedBorder;
import java.util.List;
import java.util.Locale;
import com.bea.wlpi.common.plugin.PluginPanel;
import com.bea.wlpi.common.plugin.PluginPanelContext;
import com.bea.wlpi.client.studio.Studio;
import com.bea.wlpi.common.VariableInfo;

public class ConfigPanel extends PluginPanel {

    JPanel ButtonPanel = new JPanel();
    ButtonGroup YesNoButtonGroup = new ButtonGroup();
    JRadioButton YesButton = new JRadioButton();
    JRadioButton NoButton = new JRadioButton();
    TitledBorder titledBorder = new TitledBorder(new EtchedBorder());

    public ConfigPanel() {
        super(Locale.getDefault(), "halloween");
//        super(Locale.getDefault(), "pgconfig");
        //TODO: Create resource bundle for strings
        setLayout(null);
        setBounds(12,12,420,300);
        ButtonPanel.setBorder(titledBorder);
        ButtonPanel.setLayout(null);
        add(ButtonPanel);
        ButtonPanel.setBounds(72,60,300,144);
        YesButton.setText("JavaHelp");
        YesButton.setSelected(true);
        YesNoButtonGroup.add(YesButton);
        ButtonPanel.add(YesButton);
        YesButton.setBounds(60,36,100,23);
        NoButton.setText("HTML Help");
        YesNoButtonGroup.add(NoButton);
        ButtonPanel.add(NoButton);
        NoButton.setBounds(60,60,100,23);
        titledBorder.setTitle("Online Help");
    }

    public void load() {
        ConfigData myData = (ConfigData)getData();
        if (myData != null) {
            if (myData.getYesOrNo().equals(TestPluginConstants.CONFIG_NO)) {
                NoButton.setSelected(true);
            } else {
                YesButton.setSelected(true);
            }
        }
    }

    public boolean validateAndSave() {
        ConfigData myData = (ConfigData)getData();
        if (myData != null) {
            myData.setYesOrNo(YesButton.isSelected()
                                  ? TestPluginConstants.CONFIG_YES
                                  : TestPluginConstants.CONFIG_NO);
        }
        return true;
    }
}
```

Defining the ConfigurationInfo Value Object

To further define the plug-in component data, define a `com.bea.wlpi.common.plugin.ConfigurationInfo` value object for the plug-in configuration. You can then pass the `ConfigurationInfo` object using the *`config`* parameter to the `com.bea.wlpi.common.plugin.PluginInfo` value object, when defining the basic plug-in information.

If you set the *`config`* parameter to null when defining the `PluginInfo` value object, no plug-in specific configuration is defined. In this case, the plug-in configuration dialog in the Studio will appear as in the figure “Default Plug-In Configuration Dialog in the Studio” on page 7-4. For more information about defining the `PluginInfo` value object, see “PlugInfo Object” on page B-33.

For example, the following code listing shows how to define a `ConfigurationInfo` value object.

```
ci = new ConfigurationInfo(TestPluginConstants.PLUGIN_NAME, 12,
     "test plugin configuration",
     TestPluginConstants.CONFIG_CLASSES);
```

The `CONFIG_CLASSES` field element value is listed within the `TestPluginConstants.java` class file, and defines the classes as follows:

```
final static String CONFIG_DATA =
    "com.bea.wlpi.test.plugin.ConfigData";

final static String CONFIG_PANEL =
    "com.bea.wlpi.test.plugin.ConfigPanel";
final static String[] CONFIG_CLASSES = {
    CONFIG_DATA, CONFIG_PANEL
};
```

For more information about the `ConfigurationInfo`, see “ConfigurationInfo Object” on page B-14.

Setting the Plug-In Configuration Values

To set the plug-in configuration values, use the `com.bea.wlpi.server.plugin.PluginManagerCfg` interface method defined in the following table.

Table 7-3 PluginManagerCfg Interface Method for Setting Configuration Values

Method	Description
`public void setPluginConfiguration(java.lang.String` *`pluginName`*`, com.bea.wlpi.common.VersionInfo` *`version`*`, int` *`startMode`*`, java.lang.String` *`config`*`) throws java.rmi.RemoteException, com.bea.wlpi.common.WorkflowException`	Sets the plug-in configuration information. The method parameters are defined as follows. ■ *`pluginName`*: `java.lang.String` object that specifies the plug-in name. ■ *`version:`* `com.bea.wlpi.common.VersionInfo` object that specifies the Plug-in Manager version. For information about accessing the Plug-in Manager version, see "Getting the Plug-In Framework Version" on page 2-4. ■ *`startMode`*: Integer value that specifies when the Plug-in Manager starts the plug-in. This value can be set to one of the following `com.bea.wlpi.common.plugin.PluginConstants` values: `MODE_AUTOMATIC`: plug-in is started automatically at system startup. `MODE_DISABLED`: plug-in is disabled and cannot be started. `MODE_MANUAL`: plug-in must be started manually by a user. ■ *`config`*: `java.lang.String` object that specifies the plug-in configuration information as an XML document.

For example, the following code sets the plug-in configuration information using the `pconfig.xml` file for `MyPlugin` plug-in, the specified Plug-in Manager version, `version`, and the `MODE_AUTOMATIC` start mode. In this example, `pmCfg` represents the `EJBObject` reference to the `PluginManagerCfg` EJB.

```
pmCfg.setPluginConfiguration(MyPlugin, version, MODE_AUTOMATIC,
  pconfig.xml);
```

For more information about the `setPluginConfiguration()` method, see the `com.bea.wlpi.server.plugin.PluginManagerCfg` Javadoc.

Getting the Plug-In Configuration Values

To get the plug-in configuration values, use the `com.bea.wlpi.server.plugin.PluginManager` interface method defined in the following table.

Table 7-4 PluginManager Interface Method for Getting Configuration Values

Method	Description
`public com.bea.wlpi.common.plugin.ConfigurationData getPluginConfiguration(java.lang.String` *`pluginName`*`, com.bea.wlpi.common.VersionInfo` *`version`*`) throws java.rmi.RemoteException, com.bea.wlpi.common.WorkflowException`	Gets the plug-in configuration information. The method parameters are defined as follows. ■ *`pluginName`*: `java.lang.String` object that specifies the plug-in name. ■ *`version:`* `com.bea.wlpi.common.VersionInfo` object that specifies the Plug-in Manager version. For information about accessing the Plug-in Manager version, see “Getting the Plug-In Framework Version” on page 2-4. This method returns a `com.bea.wlpi.common.plugin.ConfigurationData` object that specifies the configuration information. For more information about the methods that you can use to access specific configuration information, see “ConfigurationData Object” on page B-12.

For example, the following code gets the plug-in configuration information for `MyPlugin` and the specified Plug-in Manager version, `version`. In this example, `pm` represents the `EJBObject` reference to the `PluginManager` EJB.

```
configData=pm.setPluginConfiguration(MyPlugin, version);
```

For more information about the `getPluginConfiguration()` method, see the `com.bea.wlpi.server.plugin.PluginManager` Javadoc.

Deleting the Plug-In Configuration Values

You can delete a configuration for the plug-in if you no longer need the configuration. When you delete a configuration, you do not delete the plug-in itself, just its registered configuration.

To delete the plug-in configuration values, use the `com.bea.wlpi.server.plugin.PluginManagerCfg` interface method defined in the following table.

Table 7-5 PluginManagerCfg Interface Methods for Deleting Configuration Values

Method	Description
`public void deletePluginConfiguration( java.lang.String` *`pluginName`*`, com.bea.wlpi.common.VersionInfo` *`version`*`) throws java.rmi.RemoteException, com.bea.wlpi.common.WorkflowException`	Deletes the plug-in configuration information. The method parameters are defined as follows. ■ *`pluginName`*: `java.lang.String` object that specifies the plug-in name. ■ *`version:`* `com.bea.wlpi.common.VersionInfo` object that specifies the Plug-in Manager version. For information about accessing the Plug-in Manager version, see "Getting the Plug-In Framework Version" on page 2-4.

For example, the following code deletes the plug-in configuration values for `MyPlugin` and the specified Plug-in Manager version, `version`. In this example, `pmCfg` represents the `EJBObject` reference to the `PluginManagerCfg` EJB.

```
pmCfg.deletePluginConfiguration(MyPlugin, version);
```

For more information about the `deletePluginConfiguration()` method, see the `com.bea.wlpi.server.plugin.PluginManagerCfg` Javadoc.

Refreshing the List of Plug-Ins

To refresh the list of plug-ins, use the `com.bea.wlpi.server.plugin.PluginManagerCfg` interface method defined in the following table.

Table 7-6 PluginManagerCfg Interface Method for Refreshing the List of Plug-Ins

Method	Description
`public void refresh() throws java.rmi.RemoteException, com.bea.wlpi.common.WorkflowException`	Refreshes plug-in information in the cache. **Note:** Due to the amount of resources required to execute this method, its use should be limited. This method causes the Plug-in Manager to re-query all loaded plug-ins and rebuild its internal plug-in capabilities cache. Plug-ins can call this method if their capabilities are dynamically configured.

For example, the following code refreshes all loaded plug-ins. In this example, `pmCfg` represents the `EJBObject` reference to the `PluginManager` EJB.

```
pmCfg.refresh();
```

For more information about the `refresh()` method, see the `com.bea.wlpi.server.plugin.PluginManagerCfg` Javadoc.

Using the Studio to Manage Plug-ins

You can view, load, and configure plug-ins from within the Studio design client interface. For more information, see "Configuring Workflow Resources" in *Using the WebLogic Integration Studio*.

CHAPTER

8 Defining Plug-In Online Help

To define plug-in online help, you must perform the following steps:

1. Define the plug-in online help (JavaHelp or HTML) files.

   For example, see the HTML files in the `WLI_HOME/samples/bpm_api/htmlhelp/Sample` directory.

2. Define the `com.bea.wlpi.common.plugin.HelpSetInfo` value object, using the constructor defined in "HelpSetInfo Object" on page B-25.

   You must specify whether your are defining a JavaHelp or HTML help set using the *`helpType`* constructor parameter, and pass the following information using the *`helpNames`* constructor parameter, based on whether you are defining a JavaHelp or HTML help set, respectively:

   - JavaHelp help set (`.hs`) filename and JavaHelp help key
   - HTML help files root directory and HTML filename for the main index page or table of contents

   For more information, see "HelpSetInfo Object" on page B-25.

3. Define the `com.bea.wlpi.common.plugin.PluginInfo` value object, using the constructor defined in "PlugInfo Object" on page B-33. You must pass the name of the `HelpSetInfo` value object, defined in the previous step, as the *`helpSet`* constructor parameter value.

The following code listing is an excerpt from the plug-in sample that shows how to define plug-in HTML online help. The code defines a method, `createPluginInfo()`, that defines a `HelpSetInfo` value object and, subsequently, a `PluginInfo` object, passing this `HelpSetInfo` object. This excerpt is taken from the

`SamplePluginBean.java` file in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory. Notable lines of code are shown in **bold**.

Listing 8-1 Defining Plug-In HTML Online Help

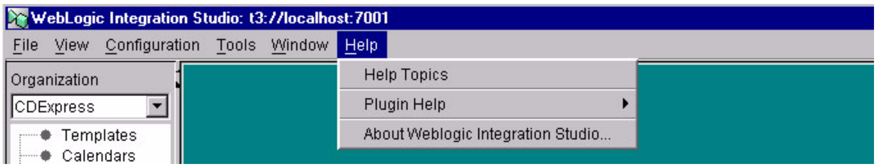
```
private PluginInfo createPluginInfo(Locale lc) {
   HelpSetInfo helpSet;
   PluginInfo pi;
   SampleBundle bundle = new SampleBundle(lc);
   String name = bundle.getString("pluginName");
   String desc = bundle.getString("pluginDesc");
   String helpName = bundle.getString("helpName");
   String helpDesc = bundle.getString("helpDesc");
   helpSet = new HelpSetInfo(
         SamplePluginConstants.PLUGIN_NAME, helpName, helpDesc,
         new String[]{ "htmlhelp/Sample", "index" },
         HelpSetInfo.HELP_HTML);
   pi = new PluginInfo(SamplePluginConstants.PLUGIN_NAME, name,
      desc, lc, SamplePluginConstants.VENDOR_NAME,
      SamplePluginConstants.VENDOR_URL,
      SamplePluginConstants.PLUGIN_VERSION,
      SamplePluginConstants.PLUGIN_FRAMEWORK_VERSION,
      null, null, helpSet);
   return pi;
}
```

In this example:

- The HTML files, along with any other supporting files must be packaged in a WAR file, for example, `sampleplugin.war`. The WAR file must be deployed as a `WebAppComponent` with a `Name` attribute value of `com.bea.wlpi.SamplePlugin`. For information about other deployment issues, see “Updating the Configuration File” on page 9-8.
- The plug-in framework builds a URL to the help set using the `htmlhelp/Sample` path, which is relative to the value of *`pluginName`*. For example: `http://localhost:7001/com.bea.wlpi.SamplePlugin/htmlhelp/Sample/index.htm`
- The default help page is `index.htm`.

The following figure illustrates the WebLogic Integration Studio Help menu that includes access to the sample plug-in HTML help set.

Figure 8-1 Plug-In Help Set



The following code listing shows an example of how to define a `HelpSetInfo` value object for the plug-in JavaHelp help set.

Listing 8-2 Defining Plug-In JavaHelp Online Help

```
javaHelpSet = new HelpSetInfo(SamplePluginConstants.PLUGIN_NAME,
        "Sample Plugin JavaHelp", "Plugin-provided help set",
        new String[] {"javahelp/HolidayHistory", "hol_intro"},
        HelpSetInfo.HELP_JAVA_HELP);
```

In this example:

- The plug-in framework would build a URL to the help set supplied by this plug-in using the `javahelp/HolidayHistory` path, which is relative to the value of *`pluginName`*. For example:
  `http://localhost:7001/com.bea.wlpi.SamplePlugin/javahelp/HolidayHistory.hs`
- The default help topic is `hol_intro`.

For information about deploying the plug-in online help, see "Deploying the Plug-In" on page 9-1.

CHAPTER

9 Deploying the Plug-In

A plug-in is a stateless session EJB. It is deployed like any other EJB. This section explains how to deploy the plug-in. It includes the following topics:

- Defining the Plug-In Deployment Descriptor Files
- Packaging the Plug-In
- Updating the Configuration File

Note: For your convenience, the plug-in sample has been deployed for you. The plug-in sample JAR, WAR, and deployment descriptor files are copied to the appropriate directories upon installation.

Defining the Plug-In Deployment Descriptor Files

To deploy the plug-in, you must define the plug-in deployment descriptor files that define the EJB and/or online help deployment properties, as described in the following sections.

Defining the Plug-In EJB Deployment Descriptor Files

The following table lists the deployment descriptor files that you may need to define to deploy the plug-in EJB.

Table 9-1 Plug-In EJB Deployment Descriptor Files

Define the following file . . .	To specify . . .
`ejb-jar.xml`	Basic EJB structure, internal dependencies, and application assembly information.
`weblogic-ejb-jar.xml`	WebLogic Server caching, clustering, and performance information, and WebLogic Server resource mappings, including security, JDBC pool, JMS connection factory, and other deployed EJB resources.
`weblogic-cmp-rdbms-jar.xml`	WebLogic Server container-managed persistence services.

For more information about the EJB deployment descriptor files, see "Deploying EJBs in the EJB Container" in *Programming WebLogic Enterprise Java Beans* at the following URL:

`http://e-docs.bea.com/wls/docs61/ejb/deploy.html`

The following code listings provide excerpts from the plug-in sample showing how to define the `ejb-jar.xml` and `weblogic-ejb-jar.xml` deployment descriptor files.

Note: Plug-ins must support container-managed transaction demarcation. Therefore, the `trans-attr` element of the plug-in notification listener method must have a value of `Required`, `Supports`, or `Mandatory`.

Listing 9-1 Plug-In Sample ejb-jar.xml EJB Deployment Descriptor

```
<?xml version="1.0"?>

<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
2.0//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_2_0.dtd">

<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>SamplePlugin</ejb-name>
      <home>com.bea.wlpi.server.plugin.PluginHome</home>
      <remote>com.bea.wlpi.server.plugin.Plugin</remote>
      <ejb-class>com.bea.wlpi.tour.po.plugin.SamplePluginBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
      <ejb-ref>
```

```
        <ejb-ref-name>ejb/PluginManagerCfg</ejb-ref-name>
        <ejb-ref-type>Session</ejb-ref-type>
        <home>com.bea.wlpi.server.plugin.PluginManagerCfgHome</home>
        <remote>com.bea.wlpi.server.plugin.PluginManagerCfg</remote>
        <ejb-link>PluginManagerCfg</ejb-link>
      </ejb-ref>
    </session>
  </enterprise-beans>
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>SamplePlugin</ejb-name>
        <method-intf>Remote</method-intf>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

Listing 9-2 Plug-In Sample weblogic-ejb-jar.xml EJB Deployment Descriptor

```
<?xml version="1.0"?>

<!DOCTYPE weblogic-ejb-jar PUBLIC "-//BEA Systems, Inc.//DTD WebLogic 6.0.0
EJB//EN" "http://www.bea.com/servers/wls600/dtd/weblogic-ejb-jar.dtd">

<weblogic-ejb-jar>
  <weblogic-enterprise-bean>
    <ejb-name>SamplePlugin</ejb-name>
    <stateless-session-descriptor>
      <pool>
        <max-beans-in-free-pool>100</max-beans-in-free-pool>
        <initial-beans-in-free-pool>0</initial-beans-in-free-pool>
      </pool>
      <stateless-clustering>
        <stateless-bean-is-clusterable>True</stateless-bean-is-clusterable>

<stateless-bean-methods-are-idempotent>True</stateless-bean-methods-are-idempot
ent>
      </stateless-clustering>
    </stateless-session-descriptor>
    <reference-descriptor>
      <ejb-reference-description>
        <ejb-ref-name>ejb/PluginManagerCfg</ejb-ref-name>
        <jndi-name>com.bea.wlpi.PluginManagerCfg</jndi-name>
      </ejb-reference-description>
    </reference-descriptor>
    <jndi-name>com.bea.wlpi.tour.po.plugin.SamplePlugin</jndi-name>
  </weblogic-enterprise-bean>
</weblogic-ejb-jar>
```

Defining the Plug-In Online Help Deployment Descriptor Files

The following table lists the deployment descriptor files that you may need to define to deploy the plug-in online help.

Table 9-2 Plug-In Online Help Deployment Descriptor Files

Define the following file . . .	To specify . . .
`web.xml`	Web application configuration information.
`weblogic.xml`	Resource mapping information for named resources in the `web.xml` file and resources residing elsewhere in WebLogic Server; and JSP and HTTP session attributes.

For more information about the online help (web application) deployment descriptor files, see "Writing WebApplication Deployment Descriptors" in *Assembling and Configuring Web Applications* at the following URL:

`http://e-docs/wls/docs61/webapp/webappdeployment.html`

The following code listings provide excerpts from the plug-in sample showing how to define the `web.xml` and `weblogic.xml` deployment descriptor files.

Listing 9-3 Plug-In Sample Online Help web.xml Deployment Descriptor

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
Application 2.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
```

```
    <display-name>Sample Plugin Online Help</display-name>
    <description>
        This Web Application serves up HTML Help for the
        WebLogic Process Integrator Sample Plugin.
    </description>
    <welcome-file-list>
    <welcome-file>
        com/bea/wlpi/tour/po/plugin/htmlhelp/index.htm
    </welcome-file>
    </welcome-file-list>
</web-app>
```

Listing 9-4 Plug-In Sample Online Help weblogic.xml Deployment Descriptor

```
<!DOCTYPE weblogic-web-app PUBLIC "-//BEA Systems, Inc.//DTD Web
Application 6.0//EN"
"http://www.bea.com/servers/wls600/dtd/weblogic-web-jar.dtd">
<weblogic-web-app>
    <description>
        This Web Application serves up HTML Help for the
        WebLogic Process Integrator Sample Plugin.
    </description>
</weblogic-web-app>
```

Packaging the Plug-In

To package the plug-in in a JAR file that will be deployed to the WebLogic Server, perform the following steps:

1. Create the build directory and compile the source files into this directory using `javac`.

   For the plug-in sample, the source files are located in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory and are compiled into the `samples/bpm_api/plugin` directory.

2. Copy the deployment descriptors (`ejb-jar.xml`, `weblogic-ejb-jar.xml`), created in the previous section, into a subdirectory called `META-INF` in the build directory.

   For example, copy the deployment descriptor files to the `WLI_HOME/samples/bpm_api/plugin/META-INF` directory.

3. Create a JAR file of the build directory, including the compiled source files and deployment descriptors.

   For the plug-in sample, the resulting JAR file is stored as `WLI_HOME/lib/sampleplugin-ejb.jar`.

4. Run `weblogic.ejbc` on the JAR file to generate the WebLogic Server container classes.

   For the plug-in sample, the output of this utility is stored in the `samples/bpm_api/plugin/ejbcgen` directory.

5. Optionally, create a WAR file containing the plug-in online help and deployment descriptor files.

   For the plug-in sample, the plug-in online help files are stored in the `WLI_HOME/samples/bpm_api/plugin/htmlhelp` directory. The resulting WAR file is stored as `WLI_HOME/lib/sampleplugin.war`.

The following code listing is an excerpt from the plug-in sample that shows how to define a build script, `build.cmd`, to package the plug-in. This file is located in the `WLI_HOME/samples/bpm_api/plugin` directory. Notable lines of code are shown in **bold**.

Note: Before running the `build.cmd` script, you must update and run the `setEnv.cmd` script to set the environment. This script is located in the `WLI_HOME/samples/bpm_api/plugin` directory.

Listing 9-5 Plug-In Sample Build Script

```
@rem Copyright (c) 2001 BEA Systems, Inc. All rights reserved.
@rem build.cmd - compile and create the sampleplugin deployable jar file.
@echo off


@rem Compile classes
setlocal
```

```
set JAVAC_ARGS=-d . -g -deprecation
echo Compiling Sample Plugin classes
"%JAVA_HOME%\bin\javac" %JAVAC_ARGS% source\*.java
endlocal

@rem Create jar
echo Building Sample Plugin EJB jar for bean classes
erase /f _sampleplugin-ejb.jar 2> nul 1> nul
@copy interfaces.jar _sampleplugin-ejb.jar 2> nul 1> nul
@copy source\Sample.gif com\bea\wlpi\tour\po\plugin\Sample.gif 2> nul 1> nul
@copy source\SamplePlugin.properties
com\bea\wlpi\tour\po\plugin\SamplePlugin.properties 2> nul 1> nul
@rem Add the standard and vendor-specific XML deployment descriptors.
"%JAVA_HOME%\bin\jar" -uf _sampleplugin-ejb.jar META-INF\ejb-jar.xml
META-INF\weblogic-ejb-jar.xml
rem Add the bean implementation classes, and helper classes.
"%JAVA_HOME%\bin\jar" -uf _sampleplugin-ejb.jar com
dir /b _sampleplugin-ejb.jar

echo Compiling EJB container classes

erase /f sampleplugin-ejb.jar 2> nul 1> nul
"%JAVA_HOME%\bin\java" -Dweblogic.ejb20.ejbc.debug=1 weblogic.ejbc -compiler
"%JAVA_HOME%\bin\javac" _sampleplugin-ejb.jar sampleplugin-ejb.jar
dir /b sampleplugin-ejb.jar
if not exist sampleplugin-ejb.jar echo *** ERROR: ejbc failed to create the
sampleplugin-ejb.jar file.


echo Building Sample Plugin WAR file for JavaHelp/HTML Help
"%JAVA_HOME%\bin\jar" -cf sampleplugin.war WEB-INF
"%JAVA_HOME%\bin\jar" -uf sampleplugin.war htmlhelp
dir /b sampleplugin.war
if not exist sampleplugin.war echo *** ERROR: failed to create the
sampleplugin.war file.

del _sampleplugin-ejb.jar
echo Done.
```

Updating the Configuration File

To deploy the plug-in, you must update the configuration file, `config.xml`, to specify the associated deployment descriptor files as part of the WebLogic Integration application.

To specify the plug-in EJB descriptor files, use the `<EJBComponent>` element. You can control the order in which the EJB JAR files are deployed using the `DeploymentOrder` attribute. In general, if plug-in A is dependent upon plug-in B, then plug-in B must be deployed first. The BPM plug-in framework ultimately dictates the order in which plug-ins are loaded. For example, if the plug-in framework attempts to load plug-in A, and plug-in A is dependent upon plug-in B, which is not yet loaded, the plug-in framework will load plug-in B.

To specify the plug-in online help files, use the `<WebAppComponent>` element. You must set the `Name` attribute to the value of the `pluginName` parameter of the `com.bea.wlpi.common.plugin.HelpSetInfo` object, which is set when you are defining the plug-in online help. For more information about defining plug-in online help, see "Defining Plug-In Online Help" on page 8-1.

The following code listing is excerpted from the samples domain `config.xml` showing the information required to deploy the plug-in sample. This file is located in the `WLI_HOME/config/samples` directory. Notable lines of code are shown in **bold**.

Listing 9-6 Deploying the Plug-In Sample EJB in the config.xml File

```
      .
      .
      .
<Application Deployed="true" Name="WLI" Path="E:\bea\wlintegration2.1\lib">
   <EJBComponent DeploymentOrder="0" Name="repository-ejb.jar"
      Targets="myserver" URI="repository-ejb.jar"/>
   <WebAppComponent Name="XTPlugin" Targets="myserver" URI="wlxtpi.war"/>
   <WebAppComponent Name="wlai" ServletReloadCheckSecs="1"
      Targets="myserver" URI="wlai.war"/>
   <EJBComponent DeploymentOrder="2" Name="wlpi-master-ejb.jar"
      Targets="myserver" URI="wlpi-master-ejb.jar"/>
   <EJBComponent DeploymentOrder="1" Name="wlpi-ejb.jar"
      Targets="myserver" URI="wlpi-ejb.jar"/>
   <EJBComponent DeploymentOrder="4" Name="wlc-wlpi-plugin.jar"
```

```
      Targets="myserver" URI="wlc-wlpi-plugin.jar"/>
   <EJBComponent DeploymentOrder="8" Name="wlai-admin-ejb"
      Targets="myserver" URI="wlai-admin-ejb.jar"/>
   <EJBComponent DeploymentOrder="5" Name="pobean.jar"
      Targets="myserver" URI="pobean.jar"/>
   <WebAppComponent Name="b2bconsole" ServletReloadCheckSecs="1"
      Targets="myserver" URI="b2bconsole.war"/>
   <EJBComponent DeploymentOrder="3" Name="wlpi-mdb-ejb.jar"
      Targets="myserver" URI="wlpi-mdb-ejb.jar"/>
   <EJBComponent DeploymentOrder="7" Name="wlai-ejb-server"
      Targets="myserver" URI="wlai-ejb-server.jar"/>
   <EJBComponent DeploymentOrder="6" Name="wlxtpi.jar"
      Targets="myserver" URI="wlxtpi.jar"/>
   <EJBComponent DeploymentOrder="9" Name="wlaiplugin-ejb.jar"
       Targets="myserver" URI="wlaiplugin-ejb.jar"/>
   <WebAppComponent Name="WLAIPlugin" Targets="myserver" URI="wlai-plugin.war"/>
   <EJBComponent DeploymentOrder="10" Name="sampleplugin-ejb.jar"
       Targets="myserver" URI="sampleplugin-ejb.jar"/>
   <WebAppComponent Name="com.bea.wlpi.SamplePlugin"
      Targets="myserver" URI="sampleplugin.war"/>
</Application>
     .
     .
     .
```

Note in the previous example that the `wlpi-master-ejb.jar file`, which contains the Plug-in Manager, is deployed before the plug-in sample file, `sampleplugin-ejb.jar`. The plug-in sample references the Plug-in Manager, which defines a dependency and, therefore, must be deployed after the Plug-in Manager file.

For more information about updating the `config.xml` file, see *BEA WebLogic Server Configuration Reference* at the following URL:

`http://e-docs.bea.com/wls/docs61/config_xml/index.html`

CHAPTER

10 BPM Plug-In Sample

This section describes the BPM plug-in sample in detail. It includes the following sections:

- Plug-In Sample Contents
- Using the Plug-In Sample

Plug-In Sample Contents

The BPM plug-in sample provides a set of plug-in classes that represent common plug-in scenarios, and is provided with the software in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory. The sample includes two workflow templates, Plug-in Order Processing and Plug-in Order Fulfillment. Excerpts from the plug-in sample are referenced throughout this document.

Note: The plug-in sample is loosely based on a generic Web-based sales order scenario that is described in detail in "Introduction to WebLogic Integration and the Example Workflows" in *Learning to Use BPM with WebLogic Integration*.

The following table describes the plug-in sample, shown in the figure "Plug-In Sample Workflow Templates" on page 1-14, listing the workflow component and the associated example source files located in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory. Plug-ins 1 through 3 are provided as part of the Plug-in Order Processing workflow template. Plug-ins 4 and 5 are provided as part of the Plug-in Order Fulfillment workflow template.

Table 10-1 Plug-In and Related Source File Descriptions

#	Workflow Component	Description	Related Source File(s)
1	Start Node	Triggered by the plug-in event, Start Order, to start workflow upon receipt of a plug-in message type (semicolon -delimited text string).	■ `StartObject.java`: Implements the `PluginObject` interface to read the plug-in data in XML format. ■ `StartNodeData.java`: Implements the `PluginData` interface to read and save data pertaining to the Start node. ■ `StartNodePanel.java`: Defines the `PluginTriggerPanel` class to display the plug-in GUI component in the design client. ■ `StartNode.java`: Defines the plug-in run-time component class to specify the execution information. ■ `OrderField.java`: Defines the plug-in field to access the plug-in data that is associated with the plug-in external event, Start Order.
2	Action: Check Inventory	Checks available inventory for an item.	■ `CheckInventoryActionObject.java`: Implements the `PluginObject` interface to read the plug-in data in XML format. ■ `CheckInventoryActionData.java`: Implements the `PluginActionData` interface to read and save plug-in action data in XML format. ■ `CheckInventoryActionPanel.java`: Defines the `PluginTriggerPanel` interface to display the plug-in GUI component in the design client. ■ `CheckInventoryAction.java`: Defines the run-time component class to specify the execution information.

Table 10-1 Plug-In and Related Source File Descriptions (Continued)

#	Workflow Component	Description	Related Source File(s)
3	Event Node: Wait for Confirmation	Triggered by plug-in event, Confirm Order Event, to pause a workflow until the receipt of a plug-in message type (semicolon -delimited text string).	■ `EventObject.java`: Implements the `PluginObject` interface to read the plug-in data in XML format. ■ `EventNodeData.java`: Implements the `PluginData` interface to read and save data pertaining to the Event node. ■ `EventNodePanel.java`: Defines the `PluginTriggerPanel` class to display the plug-in GUI component in the design client. ■ `EventNode.java`: Defines the plug-in run-time component class to specify the execution information. ■ `ConfirmField.java`: Defines the plug-in field to access the plug-in data that is associated with the plug-in external event, Confirm Order.
4	Function: CalculateTotalPrice	Calculates total price of an order based on an item number, quantity, and state/province of the shipping destination.	■ `CalculateTotalPriceFunction.java`: Implements the execution information to calculate the total price of an order using the item ID, quantity, and the state and/or province of the order. The state and/or province is used to look up the sales tax rate.

Table 10-1 Plug-In and Related Source File Descriptions (Continued)

#	Workflow Component	Description	Related Source File(s)
5	Action: Confirm Order Fulfillment	Generates an event message to trigger a Confirm Order Event node.	■ `SendConfirmObject.java`: Implements the `PluginObject` interface to read the plug-in data in XML format. ■ `SendConfirmActionData.java`: Implements the `PluginActionData` interface to read and save plug-in action data in XML format. ■ `SendConfirmActionPanel.java`: Implements the `PluginTriggerPanel` interface to display the plug-in GUI component in the design client. ■ `SendConfirmAction.java`: Implements the `PluginAction` interface to define the execution information.

The following table defines the additional source files that exist in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory.

Table 10-2 Additional Plug-in Sample Source Files

File	Description
`Sample.gif`	Custom icon that appears in the upper-right corner of the Start and Event nodes that contain customized plug-in properties when the WebLogic Integration Studio interface view is enabled. You can specify a custom plug-in icon when creating the remote plug-in object interface, as described in “Plug-In Home Interface” on page 3-5.
`SampleBundle.java`	Bundle of localized resources.
`SamplePlugin.properties`	Resource strings.
`SamplePluginBean.java`	Plug-in session EJB.
`SamplePluginConstants.java`	Plug-in constants file.
`StartOrderDriver.java`	Driver for triggering order processing.

Using the Plug-In Sample

The plug-in sample is ready to use out-of-the-box. For your convenience, the plug-in sample JAR, WAR, and deployment descriptor files are deployed to the appropriate directories when you install WebLogic Integration.

The following sections describe how to import and run the sample plug-in.

Importing the Plug-In Sample

To import the plug-in sample, use the Studio import package tool. The Studio import package tool enables you to import a workflow package, in the form of a JAR file, that can contain one or more of the following objects: templates, template definitions, event keys, and business operations.

To import the plug-in sample package:

1. Invoke the Studio design client.

   For more information, see *Using the WebLogic Integration Studio*.

2. Choose Tools—>Import Package.

   The Import wizard Select File dialog box is displayed.

3. Click Browse and navigate to the `WLI_HOME/samples/bpm_api/plugin` directory to open the file `sample_plug_in.jar`.

4. Click Next.

   The Select Components to Import dialog box appears, with the target organization set to the current organization, and all workflow objects in the import file selected by default.

5. Select the Activate workflows after import check box in order to activate the plug-in sample workflows.

   Note: You must activate a workflow before you can run it. Alternatively, you can activate a workflow after it has been imported, as described in *Using the WebLogic Integration Studio*.

6. Click Import to import the entire package, consisting of templates, template definitions, business operations, and event keys.

   The Review Import Summary dialog box appears with a summary of the objects imported.

7. Click Close to close the dialog box.

   The imported templates and template definitions now appear in the folder tree.

For more information about importing and exporting workflow packages, see "Importing and Exporting Workflow Packages" in *Using the WebLogic Integration Studio*.

Running the Plug-In Sample

The plug-in sample can be run by triggering the Plug-in Order Processing workflow template using a plug-in-defined event. For your convenience, a generic driver file, `StartOrderDriver.java`, is provided in the `WLI_HOME/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` directory. This driver generates a plug-in defined event that supplies the Plugin Order Processing workflow with the customer order information. This information is stored in the Plugin Order Processing workflow variables and is used to process the order.

To run the example:

1. Import the plug-in sample package, as described in "Importing the Plug-In Sample" on page 10-5.

   Note: You must activate a workflow before you can run it. Alternatively, you can activate a workflow after it has been imported, as described in *Using the WebLogic Integration Studio*.

2. Start BEA WebLogic Integration using the samples domain, as described in "Configuring and Starting the Samples Domain" in "Getting Started" in *Starting, Stopping, and Customizing BEA WebLogic Integration*.

3. Start the BPM Worklist tool, and log on using the following login and password:

   - Login: `admin`
   - Password: `security`

The admin user has been defined as a member of all of the roles used in the sample workflows. For more information, see “Logging On to the Worklist Application” in “Executing and Monitoring the Example Workflows” in *Learning to Use BPM with WebLogic Integration*.

You will need to interact with the plug-in sample using the Worklist tool.

4. Select the CDExpress organization from the drop-down list in the Worklist tool.

5. Execute the `StartOrderDriver`:

   a. On Windows, run the `WLI_HOME/samples/bpm_api/plugin/StartOrder.cmd` script.

   b. On UNIX, execute the following commands to set the environment and `CLASSPATH` variable, and execute the `StartOrderDriver` script:

   ```
   $WLI_HOME/setenv.sh
   setenv CLASSPATH
      $WL_HOME/lib/weblogic.jar;$WLI_HOME/lib/wlpi-ejb.jar;
      $WLI_HOME/lib/sampleplugin-ejb.jar
   $JAVA_HOME/bin/java -classpath "$CLASSPATH"
      com.bea.wlpi.tour.po.plugin.StartOrderDriver
      t3://localhost:7001 wlpisystem wlpisystem
   ```

   The Check Customer Credit task appears in the admin user’s task list.

6. Refer to the following sections in “Executing and Monitoring the Example Workflows” in *Learning to Use BPM with WebLogic Integration*, for information on executing the Order Processing and Order Fulfillment workflow tasks, respectively:

   - “Executing the Order Processing Workflow Tasks”
   - “Executing the Order Fulfillment Workflow Tasks”

CHAPTER

A Plug-In Component Definition Roadmap

The following table summarizes the steps required to define each type of plug-in component. For more information about defining plug-in components, see “Defining Plug-In Components” on page 4-1.

Table A-1 Plug-In Component Definition Roadmap

To define the following plug-in component . . .	Perform the following steps . . .
Action	1. Implement the `PluginActionData` interface to read and save the plug-in action data in XML format, as described in “Implementing the PluginActionData Interface” on page 4-20. 2. Define the `PluginActionPanel` class to display the plug-in action GUI component in the design client, as described in “Defining the PluginActionPanel Class” on page 4-40. 3. Define the plug-in action run-time component class, as described in “Defining the Run-Time Component Class for an Action” on page 4-61. 4. Define the `ActionCategoryInfo`, `ActionInfo` and/or `CategoryInfo` value objects, as described in “Defining Plug-In Value Objects” on page 2-6.

Table A-1 Plug-In Component Definition Roadmap (Continued)

To define the following plug-in component . . .	Perform the following steps . . .
Done node	1. Implement the `PluginData` interface to read and save the plug-in data in XML format, as described in “Implementing the PluginData Interface” on page 4-10. 2. Define the `PluginPanel` class to display the plug-in GUI component in the design client, as described in “Defining the PluginPanel Class” on page 4-25. 3. Define the plug-in run-time component class, as described in “Defining the Run-Time Component Class for a Done Node” on page 4-72. 4. Define the `DoneInfo` value object, as described in “Defining Plug-In Value Objects” on page 2-6.
Event node	1. Implement the `PluginData` interface to read and save the plug-in data in XML format, as described in “Implementing the PluginData Interface” on page 4-10. 2. Define the `PluginTriggerPanel` class to display the plug-in GUI component in the design client, as described in “Defining the PluginTriggerPanel Class” on page 4-46. 3. Define the plug-in run-time component class, as described in “Defining the Run-Time Component Class for an Event Node” on page 4-74. 4. Define the plug-in fields to access plug-in data that is associated with an external event received by the Event node, as defined in “Defining the Run-Time Component Class for a Message Type” on page 4-84. For plug-in Event and Start nodes that have content types other than `text/xml` and want to support key values, you must ensure that a valid event key expression is defined. You can then provide a plug-in field implementation to evaluate that key value against incoming event data. 5. Define the `EventInfo` value object, as described in “Defining Plug-In Value Objects” on page 2-6.
Function	1. Define the plug-in run-time component class, as described in “Defining the Run-Time Component Class for a Function” on page 4-79. 2. Define the `FunctionInfo` value object, as described in “Defining Plug-In Value Objects” on page 2-6.

Table A-1 Plug-In Component Definition Roadmap (Continued)

To define the following plug-in component . . .	Perform the following steps . . .
Message type (field)	1. Define the plug-in run-time component class, as described in "Defining the Run-Time Component Class for a Message Type" on page 4-84. 2. Define the `FieldInfo` value object, as described in "Defining Plug-In Value Objects" on page 2-6.
Start Node	1. Implement the `PluginData` interface to read and save the plug-in data in XML format, as described in "Implementing the PluginData Interface" on page 4-10. 2. Define the `PluginTriggerPanel` class to display the plug-in GUI component in the design client, as described in "Defining the PluginTriggerPanel Class" on page 4-46. 3. Define the plug-in run-time component class, as described in "Defining the Run-Time Component Class for a Start Node" on page 4-89. 4. Define the plug-in fields to access plug-in data that is associated with an external event received by the Start node, as defined in "Defining the Run-Time Component Class for a Message Type" on page 4-84. For plug-in Event and Start nodes that have content types other than `text/xml` and want to support key values, you must ensure that a valid event key expression is defined. You can then provide a plug-in field implementation to evaluate that key value against incoming event data. 5. Define the `StartInfo` value object, as described in "Defining Plug-In Value Objects" on page 2-6.
Workflow template definition properties	1. Implement the `PluginData` interface to read and save the plug-in data in XML format. 2. Define the `PluginPanel` class to display the plug-in GUI component in the design client, as described in "Defining the PluginPanel Class" on page 4-25. 3. Define the `TemplateDefinitionPropertiesInfo` value object, as described in "Defining Plug-In Value Objects" on page 2-6.
Workflow template properties	1. Implement the `PluginData` interface to read and save the plug-in data in XML format. 2. Define the `PluginPanel` class to display the plug-in GUI component in the design client, as described in "Defining the PluginPanel Class" on page 4-25. 3. Define the `TemplatePropertiesInfo` value object, as described in "Defining Plug-In Value Objects" on page 2-6.

Table A-1 Plug-In Component Definition Roadmap (Continued)

To define the following plug-in component . . .	Perform the following steps . . .
Variable types	1. Define the `PluginVariablePanel` class to display the plug-in GUI component that enables users to edit the plug-in variable type, as described in “Defining the PluginVariablePanel Class” on page 4-54. 2. Define the PluginVariableRenderer class to display the value of a plugin-defined variable type in the cell of a `javax.swing.JTable`, as described in “Defining the PluginVariableRenderer Class” on page 4-57. 3. Define the `VariableTypeInfo` value object, as described in “Defining Plug-In Value Objects” on page 2-6.

CHAPTER

B Plug-In Value Object Summary

This appendix describes the BPM plug-in value (or *Info*) objects and their methods. It includes the following topics:

- ActionCategoryInfo Object
- ActionInfo Object
- CategoryInfo Object
- ConfigurationData Object
- ConfigurationInfo Object
- DoneInfo Object
- EventHandlerInfo Object
- EventInfo Object
- FieldInfo Object
- FunctionInfo Object
- HelpSetInfo Object
- InfoObject Object
- PluginCapabilitiesInfo Object
- PluginDependency Object
- PlugInfo Object

- StartInfo Object
- TemplateDefinitionPropertiesInfo Object
- TemplateNodeInfo Object
- TemplatePropertiesInfo Object
- VariableTypeInfo Object

For more information about defining and accessing value object information, see "Using Plug-In Value Objects" on page 2-4.

ActionCategoryInfo Object

The `com.bea.wlpi.common.plugin.ActionCategoryInfo` object maintains plug-in action or action category information.

`ActionCategoryInfo` is the abstract base class for the following objects:

- `com.bea.wlpi.common.plugin.ActionInfo`
- `com.bea.wlpi.common.plugin.CategoryInfo`

The `ActionCategoryInfo` class extends the `com.bea.wlpi.common.plugin.InfoObject` class, as described in "InfoObject Object" on page B-28.

You can use the following constructor to create a new `ActionCategoryInfo` object:

```
public ActionCategoryInfo(
  java.lang.String pluginName,
  int ID,
  java.lang.String name,
  java.lang.String description,
  int parentSystemID,
  java.lang.String[] classNames
)
```

The following table describes the `ActionCategoryInfo` object information, the constructor parameters used to define the data, and the methods that can be used to access that information after the object is defined, if applicable.

Table B-1 ActionCategoryInfo Object Information

Object Information	Constructor Parameter	Get Method
Plug-in name (reverse-DNS version).	*`pluginName`*	`public java.lang.String getPluginName()`
Action or action category ID.	*`ID`*	`public int getID()`
Localized name of the action or action category.	*`name`*	`public java.lang.String getName()`
Localized description of the action or action category.	*`description`*	`public java.lang.String getDescription()`
ID of the parent category that identifies the action category in the action tree. The ID can be set to one of the following values: ■ `ID_EXCEPTION`: Exception Handling Actions category in the Studio. ■ `ID_INTEGRATION`: Integration Actions category in the Studio. ■ `ID_MISCELLANEOUS`: Miscellaneous Actions category in the Studio. ■ `ID_NEW`: Initial category. ■ `ID_TASK`: Task Actions category in the Studio. ■ `ID_WORKFLOW`: Workflow Actions category in the Studio. A new action category that does not have access to the parent system ID must use `ID_NEW`. For example, `ID_NEW` must be used to add: ■ New category under the root of the tree. ■ Subcategory of a new category with a system ID that has not yet been generated by the Plug-in Manager. Once the plug-in identifies a parent category, it can retrieve its system ID via the `getSystemID()` method.	*`parentSystemID`*	`public int getParentSystemID()`

Table B-1 ActionCategoryInfo Object Information (Continued)

Object Information	Constructor Parameter	Get Method
Array identifying related plug-in classes, including one entry (the fully-qualified Java class name) for each of the `KEY_*` values provided by the subclass.	*`classNames`*	`public java.lang.String getClassName(int` *`key`*`)`

For more information, see the `com.bea.wlpi.common.plugin.ActionCategoryInfo` Javadoc.

ActionInfo Object

The `com.bea.wlpi.common.plugin.ActionInfo` object maintains plug-in action information.

`ActionCategoryInfo` is the abstract base class for the `ActionInfo` object.

The `ActionInfo` class extends the `com.bea.wlpi.common.plugin.InfoObject` class, as described in "InfoObject Object" on page B-28.

You can use the following constructors to create a new `ActionInfo` object:

```
public ActionInfo(
  java.lang.String pluginName,
  int ID,
  java.lang.String name,
  java.lang.String description,
  byte[] iconByteArray,
  int parentSystemID,
  int actionStateMask,
  int actionStateTrans,
  java.lang.String[] subActionLabels,
  java.lang.String[] classNames
)
```

```
public ActionInfo(
  java.lang.String pluginName,
  int ID,
  java.lang.String name,
  java.lang.String description,
  byte[] iconByteArray,
  int parentSystemID,
  int actionStateMask,
  java.lang.String[] classNames
)
```

The following table describes the `ActionInfo` object information, the constructor parameters used to define the data, and the methods that can be used to access that information after the object is defined.

Table B-2 ActionInfo Object Information

Object Information	Constructor Parameter	Get Method
Plug-in name (reverse-DNS version).	*`pluginName`*	`public java.lang.String getPluginName()`
Action ID.	*`ID`*	`public int getID()`
Localized name of the action.	*`name`*	`public java.lang.String getName()`
Localized description of the action.	*`description`*	`public java.lang.String getDescription()`

Byte array representation of the graphical image (icon) for this plug-in, used by the Studio to represent this action when interface view is enabled.
For more information about generating the byte array representation, see "InfoObject Object" on page B-28.
	`iconByteArray`	`public javax.swing.Icon getIcon()`
		`public static final byte[] imageStreamToByteArray(java.io.InputStream` *`inputStream`*`) throws java.io.IOException`

Table B-2 ActionInfo Object Information (Continued)

Object Information	Constructor Parameter	Get Method
ID of the parent category that identifies the action category in the action tree. The ID can be set to one of the following values: ■ `ID_EXCEPTION`: Exception Handling Actions category in the Studio. ■ `ID_INTEGRATION`: Integration Actions category in the Studio. ■ `ID_MISCELLANEOUS`: Miscellaneous Actions category in the Studio ■ `ID_NEW`: Initial category. ■ `ID_TASK`: Task Actions category in the Studio. ■ `ID_WORKFLOW`: Workflow Actions category in the Studio. A new action category that does not have access to the parent system ID must use `ID_NEW`. For example, `ID_NEW` must be used to add: ■ New category under the root of the tree. ■ Subcategory of a new category with a system ID that has not yet been generated by the Plug-in Manager. Once the plug-in identifies a parent category, it can retrieve its system ID via the `getSystemID()` method.	*`parentSystemID`*	`public int getParentSystemID()`

Table B-2 ActionInfo Object Information (Continued)

Object Information	Constructor Parameter	Get Method
Bit mask describing the action states in which this action is valid. This value is formed by performing a bitwise OR of the specified values. The bit mask can be set to one or more of the following values to specify when the action is valid: ■ `ACTION_STATE_ALL`: Action is valid for all action states. ■ `ACTION_STATE_COMMIT`: Action is valid in an exception handler. ■ `ACTION_STATE_COMMIT_ASYNC`: Action is valid as an asynchronously executed subaction in an exception handler commit path. ■ `ACTION_STATE_COMMIT_SYNC`: Action is valid as a synchronously executed subaction in an exception handler commit path. ■ `ACTION_STATE_NODE`: Action is valid in a node. ■ `ACTION_STATE_NODE_ASYNC`: Action is valid as an asynchronously executed subaction in a node. ■ `ACTION_STATE_NODE_SYNC`: Action is valid as a synchronously executed subaction in a node. ■ `ACTION_STATE_ROLLBACK`: Action is valid in an exception handler rollback path. ■ `ACTION_STATE_ROLLBACK_ASYNC`: Action is valid as an asynchronously executed subaction in an exception handler rollback path. ■ `ACTION_STATE_ROLLBACK_SYNC`: Action is valid as a synchronously executed subaction in an exception handler rollback path.	*actionStateMask*	`public boolean isValidActionState (int` *`actionStateMask`*`)`

Table B-2 ActionInfo Object Information (Continued)

Object Information	Constructor Parameter	Get Method
Action state transition that results from executing the action's subactions. The state can be set to one of the following values: ■ `ACTION_STATE_TRANS_NONE`: Action has no subactions. ■ `ACTION_STATE_TRANS_ASYNC`: Action has subactions that system invokes asynchronously. ■ `ACTION_STATE_ROLLBACK_SYNC`: Action has subactions that system invokes synchronously.	*`actionStateTrans`*	`public int getActionStateTrans()`
Localized action list labels.	*`subActionLabels`*	`public java.lang.String[] getSubActionLabels()`
Array identifying related plug-in classes, including one entry (the fully-qualified Java class name) for each of the following `KEY_*` values: ■ `KEY_ACTION`: Key value specifying the `com.bea.wlpi.server.plugin.PluginAction` implementation class name. ■ `KEY_DATA`: Key value specifying the `com.bea.wlpi.common.plugin.PluginActionData` implementation class name. ■ `KEY_PANEL`: Key value specifying the `com.bea.wlpi.common.plugin.PluginActionPanel` implementation class name.	*`classNames`*	`public java.lang.Object getClass(int` *`key`*`)`

For more information, see the `com.bea.wlpi.common.plugin.ActionInfo` Javadoc.

CategoryInfo Object

The `com.bea.wlpi.common.plugin.CategoryInfo` object maintains information about the plug-in action category.

`ActionCategoryInfo` is the abstract base class for the `CategoryInfo` object.

The `CategoryInfo` class extends the `com.bea.wlpi.common.plugin.InfoObject` class, as described in "InfoObject Object" on page B-28.

You can use the following constructor to create a new `CategoryInfo` object:

```
public CategoryInfo(
  java.lang.String pluginName,
  int ID,
  java.lang.String name,
  java.lang.String description,
  int parentSystemID,
  com.bea.wlpi.common.plugin.ActionCategoryInfo[] subNodes
)
```

The following table describes the `ActionCategoryInfo` object information, the constructor parameters used to define the data, and the methods that can be used to access that information after the object is defined, if applicable.

Table B-3 CategoryInfo Object Information

Object Information	Constructor Parameter	Get Method	Set Method
Plug-in name (reverse-DNS version).	*`pluginName`*	`public java.lang.String getPluginName()`	N/A
Action category ID.	*`ID`*	`public int getID()`	N/A
Localized name of the action category.	*`name`*	`public java.lang.String getName()`	N/A

Table B-3 CategoryInfo Object Information (Continued)

Object Information	Constructor Parameter	Get Method	Set Method
Localized description of the action category.	*`description`*	`public java.lang.String getDescription()`	N/A
ID of the parent category that identifies the category in the action tree. The ID can be set to one of the following values: ■ `ID_EXCEPTION`: Exception Handling Actions category in the Studio. ■ `ID_INTEGRATION`: Integration Actions category in the Studio. ■ `ID_MISCELLANEOUS`: Miscellaneous Actions category in the Studio. ■ `ID_NEW`: Initial category. ■ `ID_TASK`: Task Actions category in the Studio. ■ `ID_WORKFLOW`: Workflow Actions category in the Studio. A new action category that does not have access to the parent system ID must use `ID_NEW`. For example, `ID_NEW` must be used to add: ■ New category under the root of the tree. ■ Subcategory of a new category with a system ID that has not yet been generated by the Plug-in Manager. Once the plug-in identifies a parent category, it can retrieve its system ID via the `getSystemID()` method.	*`parentSystemID`*	`public int getParentSystemID()`	N/A

Table B-3 CategoryInfo Object Information (Continued)

Object Information	Constructor Parameter	Get Method	Set Method
Array identifying related plug-in classes, including one entry (the fully-qualified Java class name) for each of the `KEY_*` values provided by the subclass.	*`subNodes`*	`public com.bea.wlpi.common.plugin. ActionCategoryInfo[] getSubnodes()`	`public void addSubNode(com. bea.wlpi.common .plugin. ActionCategoryInfo` *`node`*`)`
Plug-in system ID.	`N/A`	`public int getSystemID()`	`public int setSystemID(int` *`systemID`*`)`

Note: In addition to the methods defined in the previous table, the following method recursively searches the action category and its subcategories to locate the category with a matching system ID.

For more information, see the `com.bea.wlpi.common.plugin.CategoryInfo` Javadoc.

ConfigurationData Object

The `com.bea.wlpi.common.plugin.ConfigurationData` object maintains plug-in configuration information.

You can use the following constructor to create a new `ConfigurationData` object:

```
public ConfigurationData(
  java.lang.String pluginName,
  com.bea.wlpi.common.VersionInfo version,
  int status,
  int startMode,
  java.lang.String xml
)
```

The following table describes the `ConfigurationData` object information, the constructor parameters used to define the data, and the methods that can be used to access that information after the object is defined, if applicable.

Table B-4 ConfigurationData Object Information

Object Information	**Constructor Parameter**	**Get Method**
Plug-in name (reverse-DNS version).	*`pluginName`*	`public java.lang.String getPluginName()`
Plug-in version.	*`version`*	`public com.bea.wlpi.common.VersionInfo getVersion()`
Plug-in status.	*`status`*	`public int getStatus()`
Plug-in start mode.	*`startMode`*	`public int getStartMode()`
XML configuration document.	*`xml`*	`public java.lang.String getXML()`

For more information, see the `com.bea.wlpi.common.plugin.ConfigurationData` Javadoc.

ConfigurationInfo Object

The `com.bea.wlpi.common.plugin.ConfigurationInfo` object maintains plug-in configuration information.

The `ConfigurationInfo` class extends the `com.bea.wlpi.common.plugin.InfoObject` class, as described in "InfoObject Object" on page B-28.

You can use the following constructor to create a new `ConfigurationInfo` object:

```
public ConfigurationInfo(
  java.lang.String pluginName,
  int ID,
  java.lang.String description,
  java.lang.String[] classNames
)
```

The following table describes the `ConfigurationInfo` object information, the constructor parameters used to define the data, and the methods that can be used to access that information after the object is defined, if applicable.

Table B-5 ConfigurationInfo Object Information

Object Information	Constructor Parameter	Get Method
Plug-in name (reverse-DNS version).	*pluginName*	`public java.lang.String getPluginName()`
Plug-in ID.	*ID*	`public int getID()`
Plug-in description.	*description*	`public int getStatus()`
Plug-in start mode.	*startMode*	`public int getStartMode()`

Table B-5 ConfigurationInfo Object Information (Continued)

Object Information	Constructor Parameter	Get Method
Array identifying related plug-in classes, including one entry (the fully-qualified Java class name) for each of the following `KEY_*` values: ■ `KEY_DATA`: Key value specifying the `com.bea.wlpi.common.plugin.PluginData` implementation class name. ■ `KEY_PANEL`: Key value specifying the `com.bea.wlpi.common.plugin.PluginPanel` implementation class name.	*`classNames`*	`public java.lang.String getClassName(int` *`key`*`)`

For more information, see the `com.bea.wlpi.common.plugin.ConfigurationInfo` Javadoc.

DoneInfo Object

The `com.bea.wlpi.common.plugin.DoneInfo` object maintains information about a plug-in Done node.

The `DoneInfo` class extends the following classes:

- `com.bea.wlpi.common.plugin.InfoObject` class, as described in "InfoObject Object" on page B-28
- `com.bea.wlpi.common.plugin.TemplateNodeInfo` class, as defined in "TemplateNodeInfo Object" on page B-38

You can use the following constructor to create a new `DoneInfo` object:

```
public DoneInfo(
  java.lang.String pluginName,
  int ID,
  java.lang.String name,
  java.lang.String description,
  byte[] iconByteArray,
  java.lang.String[] classNames
)
```

The following table describes the `ConfigurationInfo` object information, the constructor parameters used to define the data, and the methods that can be used to access that information after the object is defined, if applicable.

Table B-6 DoneInfo Object Information

Object Information	Constructor Parameter	Get Method
Plug-in name (reverse-DNS version).	*pluginName*	`public java.lang.String getPluginName()`
Plug-in ID.	*ID*	`public int getID()`
Localized name of the Done node.	*name*	`public java.lang.String getName()`
Localized description of the Done node.	*description*	`public java.lang.String getDescription()`

Byte array representation of the graphical image (icon) for this plug-in, used by the Studio to represent this action when interface view is enabled.
For more information about generating the byte array representation, see “InfoObject Object” on page B-28.
	iconByteArray	`public javax.swing.Icon getIcon()`
		`public static final byte[] imageStreamToByteArray(java.io.InputStream` *inputStream*`) throws java.io.IOException`

Table B-6 DoneInfo Object Information (Continued)

Object Information	Constructor Parameter	Get Method
Array identifying related plug-in classes, including one entry (the fully-qualified Java class name) for each of the following `KEY_*` values: ■ `KEY_DATA`: Key value specifying the `com.bea.wlpi.common.plugin.PluginData` implementation class name. ■ `KEY_PANEL`: Key value specifying the `com.bea.wlpi.common.plugin.PluginPanel` implementation class name. ■ `KEY_DONE`: Key value specifying the `com.bea.wlpi.server.plugin.PluginDone` implementation class name.	*`classNames`*	`public java.lang.String getClassName(int` *`key`*`)`

For more information, see the `com.bea.wlpi.common.plugin.DoneInfo` Javadoc.

EventHandlerInfo Object

The `com.bea.wlpi.common.plugin.EventHandlerInfo` object maintains information about a plug-in event handler.

The `EventHandlerInfo` class extends the `com.bea.wlpi.common.plugin.InfoObject` class, as described in "InfoObject Object" on page B-28.

You can use the following constructor to create a new `EventHandlerInfo` object:

```
public EventHandlerInfo(
  java.lang.String pluginName,
  java.lang.String name,
  java.lang.String description,
  java.lang.String[] classNames
)
```

The following table describes the `EventHandlerInfo` object information, the constructor parameters used to define the data, and the methods that can be used to access that information after the object is defined, if applicable.

Table B-7 EventHandlerInfo Object Information

Object Information	Constructor Parameter	Get Method
Plug-in name (reverse-DNS version).	*`pluginName`*	`public java.lang.String getPluginName()`
Localized name of the event handler.	*`name`*	`public java.lang.String getName()`
Localized description of the event handler.	*`description`*	`public java.lang.String getDescription()`
Array identifying the related plug-in class, including one entry (the fully-qualified Java class name) for the following `KEY_*` value. `KEY_HANDLER`: Key value specifying the `com.bea.wlpi.server.plugin.EventHandler` implementation class name.	*`classNames`*	`public java.lang.String getClassName(int key)`

For more information, see the `com.bea.wlpi.common.plugin.EventHandlerInfo` Javadoc.

EventInfo Object

The `com.bea.wlpi.common.plugin.EventInfo` object maintains information about a plug-in event handler.

The `EventInfo` class extends the following classes:

- `com.bea.wlpi.common.plugin.InfoObject` class, as described in "InfoObject Object" on page B-28
- `com.bea.wlpi.common.plugin.TemplateNodeInfo` class, as defined in "TemplateNodeInfo Object" on page B-38

You can use the following constructor to create a new `EventInfo` object:

```
public EventInfo(
  java.lang.String pluginName,
  int ID,
  java.lang.String name,
  java.lang.String description,
  byte[] iconByteArray,
  java.lang.String[] classNames,
  com.bea.wlpi.common.plugin.FieldInfo fieldInfo
)
```

The following table describes the `EventHandlerInfo` object information, the constructor parameters used to define the data, and the methods that can be used to access that information after the object is defined, if applicable.

Table B-8 EventInfo Object Information

Object Information	Constructor Parameter	Get Method
Plug-in name (reverse-DNS version).	*pluginName*	`public java.lang.String getPluginName()`
Plug-in ID.	*ID*	`public int getID()`
Localized name of the event.	*name*	`public java.lang.String getName()`

Table B-8 EventInfo Object Information (Continued)

Object Information	Constructor Parameter	Get Method
Localized description of the event.	*`description`*	`public java.lang.String getDescription()`

Byte array representation of the graphical image (icon) for this plug-in, used by the Studio to represent this action when interface view is enabled.
For more information about generating the byte array representation, see "InfoObject Object" on page B-28.
	`iconByteArray`	`public javax.swing.Icon getIcon()`
		`public static final byte[] imageStreamToByteArray(java.io.InputStream` *`inputStream`*`) throws java.io.IOException`

Array identifying related plug-in classes, including one entry (the fully-qualified Java class name) for each of the following `KEY_*` values: ■ `KEY_DATA`: Key value specifying the `com.bea.wlpi.common.plugin.PluginData` implementation class name. ■ `KEY_PANEL`: Key value specifying the `com.bea.wlpi.common.plugin.PluginTriggerPanel` implementation class name. ■ `KEY_EVENT`: Key value specifying the `com.bea.wlpi.server.plugin.PluginEvent` implementation class name.	*`classNames`*	`public java.lang.String getClassName(int` *`key`*`)`
Plug-in field information.	*`fieldInfo`*	`public com.bea.wlpi.common.plugin.FieldInfo getFieldInfo()`

For more information, see the `com.bea.wlpi.common.plugin.EventInfo` Javadoc.

FieldInfo Object

The `com.bea.wlpi.common.plugin.FieldInfo` object maintains information about a plug-in field.

The `FieldInfo` class extends the `com.bea.wlpi.common.plugin.InfoObject` class, as described in "InfoObject Object" on page B-28.

You can use the following constructor to create a new `FieldInfo` object:

```
public FieldInfo(
  java.lang.String pluginName,
  int ID,
  java.lang.String name,
  java.lang.String description,
  java.lang.String[] classNames,
  boolean supportsQualifiers
)
```

The following table describes the `FieldInfo` object information, the constructor parameters used to define the data, and the methods that can be used to access that information after the object is defined, if applicable.

Table B-9 FieldInfo Object Information

Object Information	Constructor Parameter	Get Method
Plug-in name (reverse-DNS version).	*`pluginName`*	`public java.lang.String getPluginName()`
Plug-in ID.	*`ID`*	`public int getID()`
Localized name of the field.	*`name`*	`public java.lang.String getName()`
Localized description of the field.	*`description`*	`public java.lang.String getDescription()`

Table B-9 FieldInfo Object Information (Continued)

Object Information	Constructor Parameter	Get Method
Array identifying related plug-in classes, including one entry (the fully-qualified Java class name) for each of the following `KEY_*` values: ■ `KEY_FIELD`: Key value specifying the `com.bea.wlpi.common.plugin.PluginField` implementation class name. ■ `DEFAULT_FIELD`: Key value specifying the built-in field type that supports expression validation but not evaluation. This field type supports qualifiers and is used to validate expressions that reference plug-in data fields if the plug-in itself is not currently loaded. ■ `XMLFIELD`: Key value specifying a built-in field type that returns XML single-element text values. This field type supports qualifiers (because XML is hierarchical).	*`classNames`*	`public java.lang.String getClassName(int` *`key`*`)`
Flag specifying whether or not the field type supports dot-delimited names. For example, if the `PostalCode` field is embedded in the `Address` field, the name for the field would be `Address.PostalCode`. The expression evaluator uses this flag to determine whether a field reference is valid in an expression in a plug-in Start, Event, or Event key.	*`supportsQualifiers`*	`public boolean supportsQualifiers()`

For more information, see the `com.bea.wlpi.common.plugin.FieldInfo` Javadoc.

FunctionInfo Object

The `com.bea.wlpi.common.plugin.FunctionInfo` object maintains information about a plug-in function.

The `FieldInfo` class extends the `com.bea.wlpi.common.plugin.InfoObject` class, as described in "InfoObject Object" on page B-28.

You can use the following constructor to create a new `FunctionInfo` object:

```
public FunctionInfo(
  java.lang.String pluginName,
  int ID,
  java.lang.String name,
  java.lang.String description,
  java.lang.String prototype,
  java.lang.String[] classNames,
  int argcmin,
  int argcmax
)
```

The following table describes the `FunctionInfo` object information, the constructor parameters used to define the data, and the methods that can be used to access that information after the object is defined, if applicable.

Table B-10 FunctionInfo Object Information

Object Information	**Constructor Parameter**	**Get Method**
Plug-in name (reverse-DNS version).	*`pluginName`*	`public java.lang.String getPluginName()`
Plug-in ID.	*`ID`*	`public int getID()`
Localized name of the function.	*`name`*	`public java.lang.String getName()`
Localized description of the function.	*`description`*	`public java.lang.String getDescription()`
Localized prototype for this function.	*`prototype`*	`public java.lang.String prototype()`

Table B-10 FunctionInfo Object Information (Continued)

Object Information	Constructor Parameter	Get Method
Array identifying related plug-in classes, including one entry (the fully-qualified Java class name) for the following `KEY_*` value. `KEY_EVALUATOR`: Key value specifying the `com.bea.wlpi.common.plugin.PluginFunction` implementation class name.	*`classNames`*	`public java.lang.String getClassName(int` *`key`*`)`
Minimum number of arguments permitted. The expression evaluator uses this information to validate calls to this function.	*`argcmin`*	`public int getMinArgCount()`
Maximum number of arguments permitted. The expression evaluator uses this information to validate calls to this function.	*`argcmax`*	`public int getMaxArgCount()`

For more information, see the `com.bea.wlpi.common.plugin.FunctionInfo` Javadoc.

HelpSetInfo Object

The com.bea.wlpi.common.plugin.HelpSetInfo object maintains information about the plug-in online help. Plug-ins can support both HTML and JavaHelp online help systems. The plug-in online help files must be packaged in a WAR file and deployed as part of the process engine. In order for the BPM client applications to retrieve the appropriate help files, the online help WAR file must be deployed under the name of the plug-in to which it correlates.

BPM client applications can use the Plug-in Manager (or other EJB) `ClassLoader` to determine the URL of the process engine. Using the `HelpSetInfo` object values, the client applications can obtain a full URL to access the help files through `http` or `https`.

The `HelpSetInfo` class extends the com.bea.wlpi.common.plugin.InfoObject class, as described in “InfoObject Object” on page B-28.

You can use the following constructor to create a new `HelpSetInfo` object:

```
public HelpSetInfo(
  java.lang.String pluginName,
  java.lang.String name,
  java.lang.String description,
  java.lang.String[] helpNames,
  int helpType
)
```

The following table describes the `HelpSetInfo` object information, the constructor parameters used to define the data, and the methods that can be used to access that information after the object is defined, if applicable.

Table B-11 HelpSetInfo Object Information

Object Information	Constructor Parameter	Get Method
Plug-in name (reverse-DNS version) supplying the online help set. In order for BPM client applications to form the correct URL for the help set, this name must match the web application under which the WAR file must be deployed.	*`pluginName`*	`public java.lang.String getPluginName()`
Localized name of the online help. This value is used as the label value for the user interface menu option that is used to access the non-context sensitive help.	*`name`*	`public java.lang.String getName()`
Localized description of the online help.	*`description`*	`public java.lang.String getDescription()`
Array identifying related plug-in classes, including one entry (the fully-qualified Java class name) for each of the following `KEY_*` values: ■ `KEY_HELP_SET`: Key value to retrieve the JavaHelp help set (`.hs`) filename or HTML help files root directory. ■ `KEY_HELP_ID`: Key value to retrieve the JavaHelp help key or HTML filename for the main index page or table of contents. The value of each entry is interpreted according to the value of the *`helpType`* parameter, as defined in the table "Help Types and Related Plug-In Classes" on page B-27.	*`helpNames`*	`public java.lang.String getClassName(int key)`

Table B-11 HelpSetInfo Object Information (Continued)

Object Information	Constructor Parameter	Get Method
Type of help provided by the plug-in, which can be set to one of the following values: ■ `HELP_JAVA_HELP`: Name of the JavaHelp help set, or help key for the main index page or table of contents. ■ `HELP_HTML`: Name of the HTML help files root directory, or name of the main index page or table of contents, relative to the directory specified by the `KEY_HELP_SET` entry. The value of each entry is interpreted according to the value of the *`helpNames`* parameter, as defined in the table “Help Types and Related Plug-In Classes” on page B-27.	*`helpType`*	`public int getHelpType()`

The following table defines the help types (*`helpType`* values) and related plug-in classes (*`helpNames`* values).

Table B-12 Help Types and Related Plug-In Classes

`helpType` **Value**	***`helpNames`*** **Value**	
	`KEY_HELP_SET`	`KEY_HELP_ID`
`HELP_JAVA_HELP`	Name of the JavaHelp help set file, relative to the root of the WAR file, containing the help files (for example, `javahelp/MyPluginHelpSet.hs`). If an extension is not included, JavaHelp automatically appends an `.hs` extension.	JavaHelp help key for the main index page or table of contents.
`HELP_HTML`	Name of the root directory of the HTML help files that must include a trailing slash (for example, `htmlhelp/`).	Name of the HTML file (not including the required `.htm` extension) containing the main index page or table of contents, relative to the directory specified by the `KEY_HELP_SET` entry.

For more information, see the com.bea.wlpi.common.plugin.HelpSetInfo Javadoc.

InfoObject Object

The com.bea.wlpi.common.plugin.InfoObject object provides the abstract base class for all plug-in value objects.

You can use the following constructor to create a new InfoObject object:

```
public InfoObject(
  java.lang.String pluginName,
  int ID,
  java.lang.String name,
  java.lang.String description,
  java.lang.String[] classNames
)
```

The following table describes the InfoObject object information, the constructor parameters used to define the data, and the methods that can be used to access that information after the object is defined, if applicable.

Table B-13 InfoObject Object Information

Object Information	Constructor Parameter	Get Method
Plug-in name (reverse-DNS version).	*pluginName*	public java.lang.String getPluginName()
Plug-in ID.	*ID*	public int getID()
Localized name of the object.	*name*	public java.lang.String getName()
Localized description of the object.	*description*	public java.lang.String getDescription()
Array identifying related plug-in classes, including one entry (the fully-qualified Java class name) for each of the KEY_* values provided by the subclass.	*classNames*	public java.lang.String getClassName(int *key*)

The `InfoObject` object also provides the following method for generating an *`iconByteArray`* value from an input stream that can be used when constructing `ActionInfo`, `DoneInfo`, `EventInfo`, `StartInfo`, and `TemplateDefinitionPropertiesInfo` objects:

```
public static final byte[]
imageStreamToByteArray(java.io.InputStream inputStream) throws
java.io.IOException
```

For more information, see the `com.bea.wlpi.common.plugin.InfoObject` Javadoc.

PluginCapabilitiesInfo Object

The `com.bea.wlpi.common.plugin.PluginCapabilitiesInfo` object maintains information about the plug-in capabilities.

The `PluginCapabilitiesInfo` object describes the complete set of plug-in capabilities once the plug-in has been loaded. Prior to being loaded, basic plug-in information can be accessed using the `com.bea.wlpi.common.plugin.PluginInfo` object, as described in "PlugInfo Object" on page B-33.

You can use the following constructor to create a new `PluginCapabilitiesInfo` object:

```
public PluginCapabilitiesInfo(
  com.bea.wlpi.common.plugin.PluginInfo info,
  com.bea.wlpi.common.plugin.ActionCategoryInfo[] actions,
  com.bea.wlpi.common.plugin.EventInfo[] events,
  com.bea.wlpi.common.plugin.FieldInfo[] fields,
  com.bea.wlpi.common.plugin.FunctionInfo[] functions,
  com.bea.wlpi.common.plugin.StartInfo[] starts,
  com.bea.wlpi.common.plugin.DoneInfo[] dones,
  com.bea.wlpi.common.plugin.VariableTypeInfo[] variableTypes,
  com.bea.wlpi.common.plugin.TemplatePropertiesInfo[] template,
  com.bea.wlpi.common.plugin.TemplateDefinitionPropertiesInfo[]
    templateDefinition,
  com.bea.wlpi.common.plugin.EventHandlerInfo eventHandler
)
```

The following table describes the `PluginCapabilitiesInfo` object information, the constructor parameters used to define the data, and the methods that can be used to access that information after the object is defined, if applicable.

Table B-14 PluginCapabilitiesInfo Object Information

Object Information	Constructor Parameter	Get Method
Basic plug-in information.	*`info`*	`public java.lang.String getPluginInfo()`
Actions and action categories provided by the plug-in.	*`actions`*	`public com.bea.wlpi.common.plugin.ActionCategoryInfo[] getActionInfo()`
Events provided by the plug-in.	*`events`*	`public com.bea.wlpi.common.plugin.EventInfo[] getEventInfo()`
Fields provided by the plug-in.	*`fields`*	`public com.bea.wlpi.common.plugin.FieldInfo[] getFieldInfo()`
Functions provided by the plug-in.	*`functions`*	`public com.bea.wlpi.common.plugin.FunctionInfo[] getFunctionInfo()`
Start nodes provided by the plug-in.	*`starts`*	`public com.bea.wlpi.common.plugin.StartInfo[] getStartInfo()`
Done nodes provided by the plug-in.	*`dones`*	`public com.bea.wlpi.common.plugin.DoneInfo[] getDoneInfo()`
Variables provided by the plug-in.	*`variableTypes`*	`public com.bea.wlpi.common.plugin.VariableTypesInfo[] getVariableTypesInfo()`

Table B-14 PluginCapabilitiesInfo Object Information (Continued)

Object Information	Constructor Parameter	Get Method
Workflow template properties provided by the plug-in.	*`template`*	`public com.bea.wlpi.common.plugin.TemplatePropertiesInfo[] getTemplateInfo()`
Workflow template definition properties provided by the plug-in.	*`templateDefinition`*	`public com.bea.wlpi.common.plugin.TemplateDefinitionPropertiesInfo[] getTemplateDefinitionInfo()`
Event handler information.	*`eventHandler`*	`public com.bea.wlpi.common.plugin.EventHandlerInfo getEventHandlerInfo()`

For more information, see the `com.bea.wlpi.common.plugin.PluginCapabilitiesInfo` Javadoc.

PluginDependency Object

The `com.bea.wlpi.common.plugin.PluginDependency` object maintains information about the plug-in dependencies.

The `PluginDependency` class extends the `com.bea.wlpi.common.plugin.InfoObject` class, as described in "InfoObject Object" on page B-28.

You can use the following constructor to create a new `PluginCapabilitiesInfo` object:

```
public PluginDependency(
  java.lang.String pluginName,
  java.lang.String description,
  java.lang.String masterPluginName,
  java.lang.String vendor,
  com.bea.wlpi.common.VersionInfo version
)
```

The following table describes the `PluginDependency` object information, the constructor parameters used to define the data, and the methods that can be used to access that information after the object is defined, if applicable.

Table B-15 PluginDependency Object Information

Object Information	Constructor Parameter	Get Method
Plug-in name (reverse-DNS version).	*`pluginName`*	`public java.lang.String getPluginName()`
Localized description of the object.	*`description`*	`public java.lang.String getDescription()`
Master plug-in name (reverse-DNS version).	*`masterPluginName`*	`public java.lang.String getMasterPluginName()`
Master plug-in vendor name.	*`vendor`*	`public java.lang.String getVendor()`
Master plug-in version.	*`version`*	`public com.bea.wlpi.common.VersionInfo getVersion()`

For more information, see the `com.bea.wlpi.common.plugin.PluginDependency` Javadoc.

PlugInfo Object

The `com.bea.wlpi.common.plugin.PluginInfo` object maintains basic information about the plug-in.

The `PluginInfo` object describes the basic set of plug-in capabilities before the plug-in has been loaded. After being loaded, basic plug-in information can be accessed using the `com.bea.wlpi.common.plugin.PluginCapabilitiesInfo` object, as described in "PluginCapabilitiesInfo Object" on page B-29.

The `PluginInfo` class extends the `com.bea.wlpi.common.plugin.InfoObject` class, as described in "InfoObject Object" on page B-28.

You can use the following constructor to create a new `PluginCapabilitiesInfo` object:

```
public PluginInfo(
  java.lang.String pluginName,
  java.lang.String name,
  java.util.Locale lc,
  java.lang.String vendor,
  java.lang.String url,
  com.bea.wlpi.common.VersionInfo version,
  com.bea.wlpi.common.VersionInfo pluginFrameworkVersion,
  com.bea.wlpi.common.plugin.PluginDependency[] dependencies,
  com.bea.wlpi.common.plugin.ConfigurationInfo config,
  com.bea.wlpi.common.plugin.HelpSetInfo helpSet
)
```

The following table describes the `PluginDependency` object information, the constructor parameters used to define the data, and the methods that can be used to access that information after the object is defined, if applicable.

Table B-16 PluginInfo Object Information

Object Information	Constructor Parameter	Get Method
Plug-in name (reverse-DNS version).	*pluginName*	`public java.lang.String getPluginName()`
Localized name of the object.	*name*	`public java.lang.String getName()`

Table B-16 PluginInfo Object Information (Continued)

Object Information	Constructor Parameter	Get Method
Locale to localize the display strings.	*`lc`*	`public java.lang.String getLocale()`
Plug-in vendor name.	*`vendor`*	`public java.lang.String getVendor()`
Plug-in vendor URL.	*`url`*	`public java.lang.String getURL()`
Plug-in version.	*`version`*	`public com.bea.wlpi.common.VersionInfo getVersion()`
Plug-in framework version.	*`pluginFrameworkVersion`*	`public com.bea.wlpi.common.VersionInfo getPluginFrameworkVersion()`
Plug-in dependencies.	*`dependencies`*	`public com.bea.wlpi.common.plugin.PluginDependency[] getDependencyInfo()`
Plug-in configuration information.	*`config`*	`public com.bea.wlpi.common.plugin.ConfigurationInfo getConfigurationInfo()`
JavaHelp help set provided by plug-in.	*`helpSet`*	`public com.bea.wlpi.common.plugin.HelpSetInfo getHelpSetInfo()`

For more information, see the `com.bea.wlpi.common.plugin.PluginInfo` Javadoc.

StartInfo Object

The `com.bea.wlpi.common.plugin.StartInfo` object maintains information about a plug-in Start node.

The `StartInfo` class extends the following classes:

- `com.bea.wlpi.common.plugin.InfoObject` class, as described in "InfoObject Object" on page B-28
- `com.bea.wlpi.common.plugin.TemplateNodeInfo` class, as defined in "TemplateNodeInfo Object" on page B-38

You can use the following constructor to create a new `StartInfo` object:

```
public StartInfo(
  java.lang.String pluginName,
  int ID,
  java.lang.String name,
  java.lang.String description,
  byte[] iconByteArray,
  java.lang.String[] classNames,
  com.bea.wlpi.common.plugin.FieldInfo fieldInfo
)
```

The following table describes the `ConfigurationInfo` object information, the constructor parameters used to define the data, and the methods that can be used to access that information after the object is defined, if applicable.

Table B-17 StartInfo Object Information

Object Information	Constructor Parameter	Get Method
Plug-in name (reverse-DNS version).	*pluginName*	`public java.lang.String getPluginName()`
Plug-in ID.	*ID*	`public int getID()`
Localized name of the Start node.	*name*	`public java.lang.String getName()`

Table B-17 StartInfo Object Information (Continued)

Object Information	Constructor Parameter	Get Method
Localized description of the Start node.	*description*	`public java.lang.String getDescription()`

Byte array representation of the graphical image (icon) for this plug-in, used by the Studio to represent this action when interface view is enabled.
For more information about generating the byte array representation, see "InfoObject Object" on page B-28.
	iconByteArray	`public javax.swing.Icon getIcon()`
		`public static final byte[] imageStreamToByteArray(java.io.InputStream` *`inputStream`*`) throws java.io.IOException`

Array identifying related plug-in classes, including one entry (the fully-qualified Java class name) for each of the following `KEY_*` values: ■ `KEY_DATA`: Key value specifying the `com.bea.wlpi.common.plugin.PluginData` implementation class name. ■ `KEY_PANEL`: Key value specifying the `com.bea.wlpi.common.plugin.PluginPanel` implementation class name. ■ `KEY_START`: Key value specifying the `com.bea.wlpi.server.plugin.PluginStart` implementation class name.	*classNames*	`public java.lang.String getClassName(int` *`key`*`)`
Plug-in field information.	*fieldInfo*	`public com.bea.wlpi.common.plugin.FieldInfo getFieldInfo()`

For more information, see the `com.bea.wlpi.common.plugin.StartInfo` Javadoc.

TemplateDefinitionPropertiesInfo Object

The `com.bea.wlpi.common.plugin.TemplateDefinitionPropertiesInfo` object maintains information about the plug-in template definition properties.

The `TemplateDefinitionPropertiesInfo` class extends the `com.bea.wlpi.common.plugin.InfoObject` class, as described in "InfoObject Object" on page B-28.

You can use the following constructor to create a new `TemplateDefinitionPropertiesInfo` object:

```
public DoneInfo(
  java.lang.String pluginName,
  java.lang.String name,
  java.lang.String description,
  java.lang.String[] classNames
)
```

The following table describes the `TemplateDefinitionPropertiesInfo` object information, the constructor parameters used to define the data, and the methods that can be used to access that information after the object is defined, if applicable.

Table B-18 TemplateDefinitionPropertiesInfo Object Information

Object Information	Constructor Parameter	Get Method
Plug-in name (reverse-DNS version).	*pluginName*	`public java.lang.String getPluginName()`
Localized name of the object. This string defines the contents of the plug-in tab in the Template Definition Properties dialog box.	*name*	`public java.lang.String getName()`
Localized description of the object.	*description*	`public java.lang.String getDescription()`

Table B-18 TemplateDefinitionPropertiesInfo Object Information (Continued)

Object Information	Constructor Parameter	Get Method
Array identifying related plug-in classes, including one entry (the fully-qualified Java class name) for each of the following `KEY_*` values: ■ `KEY_DATA`: Key value specifying the `com.bea.wlpi.common.plugin.PluginData` implementation class name. ■ `KEY_PANEL`: Key value specifying the `com.bea.wlpi.common.plugin.PluginPanel` implementation class name.	*`classNames`*	`public java.lang.String getClassName(int` *`key`*`)`

For more information, see the `com.bea.wlpi.common.plugin.TemplateDefinitionPropertiesInfo` Javadoc.

TemplateNodeInfo Object

The `com.bea.wlpi.common.plugin.TemplateNodeInfo` object maintains information about a plug-in template definition node.

`TemplateNodeInfo` is extended by the following classes:

- `com.bea.wlpi.common.plugin.DoneInfo`
- `com.bea.wlpi.common.plugin.EventInfo`
- `com.bea.wlpi.common.plugin.StartInfo`

The `TemplateNodeInfo` class extends the `com.bea.wlpi.common.plugin.InfoObject` class, as described in “InfoObject Object” on page B-28.

You can use the following constructor to create a new `TemplateNodeInfo` object:

```
public TemplateNodeInfo(
  java.lang.String pluginName,
  int ID,
  java.lang.String name,
  java.lang.String description,
  byte[] iconByteArray,
  java.lang.String[] classNames
)
```

The following table describes the `TemplateDefinitionPropertiesInfo` object information, the constructor parameters used to define the data, and the methods that can be used to access that information after the object is defined, if applicable.

Table B-19 TemplateDefinitionPropertiesInfo Object Information

Object Information	Constructor Parameter	Get Method
Plug-in name (reverse-DNS version).	*`pluginName`*	`public java.lang.String getPluginName()`
Plug-in ID.	*`ID`*	`public int getID()`
Localized name of the template definition node.	*`name`*	`public java.lang.String getName()`
Localized description of the template definition node.	*`description`*	`public java.lang.String getDescription()`

Byte array representation of the graphical image (icon) for this plug-in, used by the Studio to represent this action when interface view is enabled.
For more information about generating the byte array representation, see "InfoObject Object" on page B-28.
	`iconByteArray`	`public javax.swing.Icon getIcon()`
		`public static final byte[] imageStreamToByteArray(java.io.InputStream` *`inputStream`*`) throws java.io.IOException`

Array identifying related plug-in classes, including one entry (the fully-qualified Java class name) for each of the `KEY_*` values defined for the corresponding subclasses.	*`classNames`*	`public java.lang.String getClassName(int` *`key`*`)`

For more information, see the `com.bea.wlpi.common.plugin.TemplateNodeInfo` Javadoc.

TemplatePropertiesInfo Object

The `com.bea.wlpi.common.plugin.TemplatePropertiesInfo` object maintains information about the plug-in template properties.

The `TemplatePropertiesInfo` class extends the `com.bea.wlpi.common.plugin.InfoObject` class, as described in "InfoObject Object" on page B-28.

You can use the following constructor to create a new `TemplatePropertiesInfo` object:

```
public TemplatePropertiesInfo(
  java.lang.String pluginName,
  java.lang.String name,
  java.lang.String description,
  java.lang.String[] classNames
)
```

The following table describes the `TemplateDefinitionPropertiesInfo` object information, the constructor parameters used to define the data, and the methods that can be used to access that information after the object is defined, if applicable.

Table B-20 TemplatePropertiesInfo Object Information

Object Information	Constructor Parameter	Get Method
Plug-in name (reverse-DNS version).	*pluginName*	`public java.lang.String getPluginName()`
Localized name of the object. This string defines the contents of the plug-in tab in the Template Properties dialog box.	*name*	`public java.lang.String getName()`
Localized description of the object.	*description*	`public java.lang.String getDescription()`

Table B-20 TemplatePropertiesInfo Object Information (Continued)

Object Information	Constructor Parameter	Get Method
Array identifying related plug-in classes, including one entry (the fully-qualified Java class name) for each of the following KEY_* values: ■ KEY_DATA: Key value specifying the com.bea.wlpi.common.plugin.PluginData implementation class name. ■ KEY_PANEL: Key value specifying the com.bea.wlpi.common.plugin.PluginPanel implementation class name.	*classNames*	public java.lang.String getClassName(int *key*)

For more information, see the com.bea.wlpi.common.plugin.TemplatePropertiesInfo Javadoc.

VariableTypeInfo Object

The com.bea.wlpi.common.plugin.VariableTypeInfo object maintains information about a plug-in variable.

The VariableTypeInfo class extends the com.bea.wlpi.common.plugin.InfoObject class, as described in "InfoObject Object" on page B-28.

You can use the following constructor to create a new VariableTypeInfo object:

```
public VariableTypeInfo(
  java.lang.String pluginName,
  int ID,
  java.lang.String name,
  java.lang.String description,
  int variableType,
  java.lang.Class valueClass,
  java.lang.String[] classNames
)
```

The following table describes the `VariableTypeInfo` object information, the constructor parameters used to define the data, and the methods that can be used to access that information after the object is defined, if applicable.

Table B-21 VariableTypeInfo Object Information

Object Information	**Constructor Parameter**	**Get Method**
Plug-in name (reverse-DNS version).	*`pluginName`*	`public java.lang.String getPluginName()`
Plug-in ID.	*`ID`*	`public int getID()`
Localized name of the variable type.	*`name`*	`public java.lang.String getName()`
Localized description of the variable type.	*`description`*	`public java.lang.String getDescription()`
Variable type which can be set to one of the following integer values: ■ `TYPE_ENTITY`: Remote reference to an entity EJB. ■ `TYPE_OBJECT`: Local Java object. ■ `TYPE_SESSION`: Remove reference to a session EJB.	*`variableType`*	`public int getVariableType()`
Fully-qualified Java class of the allowed value type.	*`valueClass`*	`public java.lang.Class getValueClass()`
Array identifying related plug-in classes, including one entry (the fully-qualified Java class name) for each of the following `KEY_*` values: ■ `KEY_RENDERER`: Key value specifying the `com.bea.wlpi.common.plugin.PluginVariableRenderer` implementation class name. ■ `KEY_PANEL`: Key value specifying the `com.bea.wlpi.common.plugin.PluginPanel` implementation class name.	*`classNames`*	`public java.lang.`*`String getClassName(int key`*`)`

For more information, see the `com.bea.wlpi.common.plugin.VariableTypeInfo` Javadoc.

CHAPTER

C BPM Graphical User Interface Style Sheet

This section provides information to help you design custom plug-ins based on Java Swing classes. It includes the following sections:

- Designing a Plug-In
- Working with Interaction Components
- Working with Presentation Components
- Recommended Reading

Designing a Plug-In

Designing a plug-in that meets the learning and information needs of users requires careful planning and a systematic application of the principles described in "Working with Interaction Components" on page C-3 and "Working with Presentation Components" on page C-10.

To design a plug-in panel, perform the following steps:

1. Decide how you want to extend the functionality of BPM. For example, your plug-in can modify the definition and run-time behavior of the following programming constructs:
   - Actions
   - Done nodes

- Event nodes
- Functions
- Message types
- Starts
- Template properties
- Template definition properties
- Variable types

2. Identify the tasks required to modify a construct. In this case, each construct will require a separate panel. For example, to modify template properties, panel tasks may include adding, updating, or deleting new properties that are not available by default.

3. Select a control to represent each task. The following table lists the recommended controls for each type of task category.

Table C-1 Recommended Controls for Task Categories

Task Category	Control
Selecting mutually exclusive options	Radio buttons
Selecting non-exclusive options	Check boxes
Performing an action	Command buttons
Selecting an item from a set	List boxes or drop-down list boxes
Entering or viewing large amounts of information at the same time	Tables
Setting attribute values	Text-entry fields

4. Decide how to display panel controls using one or more of the following presentation elements:

   - Use color to differentiate between control types or to add visual variety. For example, use two complementary colors to highlight alternate rows in a table. This technique adds visual variety and improves readability.
   - Use 12-point Arial regular for all control labels.

- Select a horizontal or vertical flow for panel information. For example, list boxes that allow the movement of items between one another benefit from a horizontal layout. On the other hand, groups of stacked text-entry fields are best implemented as a vertical layout.
- Balance the placement of controls along the vertical, horizontal, or diagonal axis of a panel. For example, place command buttons on the bottom right-hand margin of a panel to balance other controls along the diagonal axis.

5. Design a 32x32-pixel icon to represent plug-in actions assigned to tasks if you do not want to use the default graphic.
6. Validate the value of each control to check for errors in syntax and data type. For each error condition, create an error-message dialog box that clearly identifies the problem and suggests a solution.

Working with Interaction Components

Interaction components allow users to interact with your plug-in panels. Choosing appropriate components is not simply a matter of following a formula; rather, it involves achieving a balance among industry standards, corporate standards, and user needs.

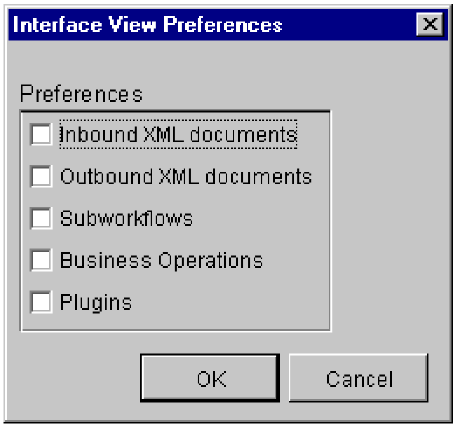
The following sections provide guidelines for designing interaction components:

- Check Boxes
- Command Buttons
- List Boxes
- Radio Buttons
- Tables
- Text-Entry Fields

Check Boxes

Check boxes can be used to replace some data-entry fields and provide a quick way to make multiple choices. The following BPM example illustrates appropriate check-box design.

Figure C-1 Check Box Design



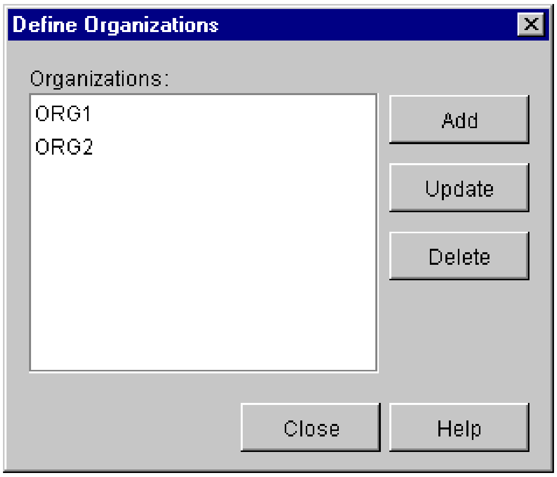
Use the following guidelines to design check boxes:

- Use check boxes to allow users to select multiple items from a group or to toggle a feature on or off.
- Present multiple check boxes together using a group box. Provide a descriptive label for the group box.
- Align check boxes vertically.
- Limit the number of check boxes to ten or fewer.

Command Buttons

Command buttons are the interaction components used most frequently by users to complete tasks within a dialog box. The following BPM example illustrates appropriate command-button design.

Figure C-2 Command Button Design



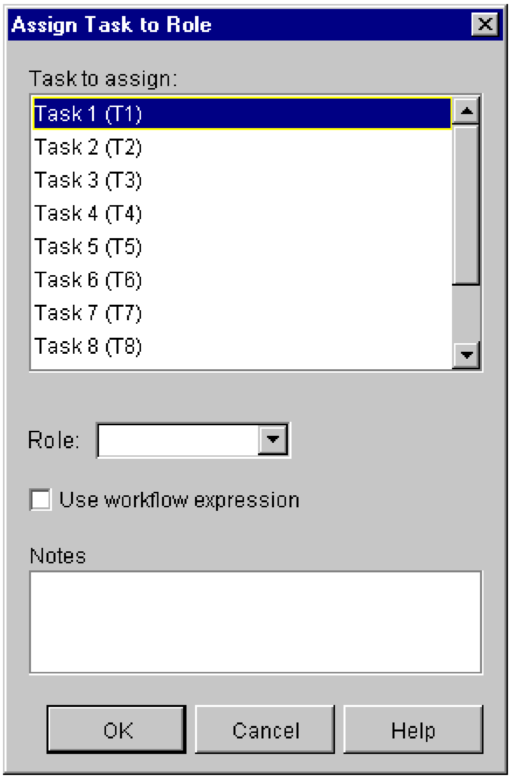
Use the following guidelines to design command buttons:

- Use command buttons only for frequent actions such as OK, Cancel, and Help.
- Label command buttons carefully. Use multiple words when necessary to convey the meaning of an action.
- Size command buttons relative to each other. If the label length for a series of buttons is similar, make all the buttons the size of the largest one. If the label length for a series of buttons varies, use two button sizes: one for shorter labels and one for longer labels.
- Group command buttons that have similar functionality.
- Separate command buttons from other controls using white space.

List Boxes

List boxes provide an alternative to data entry. The following BPM example illustrates appropriate list-box design.

Figure C-3 List Box Design



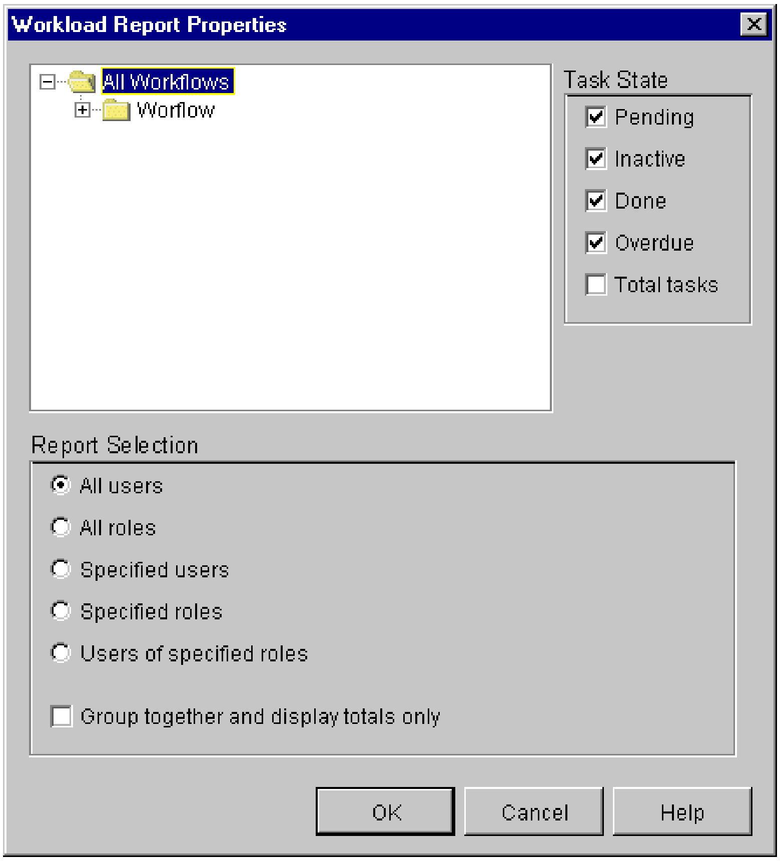
Use the following guidelines to design list boxes:

- Use a list box instead of radio buttons when there are more than six items, if possible.
- Show at least three but no more than eight items at a time. If there are more than eight items, add a vertical scroll bar.
- Choose a label that describes the items in the list. Place the label on the top of the list box, left justified, followed by a colon. For example, in the previous figure, the label Task to Assign is displayed above the list.
- Use a drop-down list box to save vertical space if most users will select the first item in the list.

Radio Buttons

Radio buttons can be used in place of many data-entry fields. The following BPM example illustrates appropriate radio button design.

Figure C-4 Radio Button Design



Task State: [✓ Pending] [✓ Inactive] [✓ Done] [✓ Overdue] Total tasks

Report Selection: [✓ All users] All roles Specified users Specified roles Users of specified roles

Group together and display totals only

Use the following guidelines to design radio buttons:

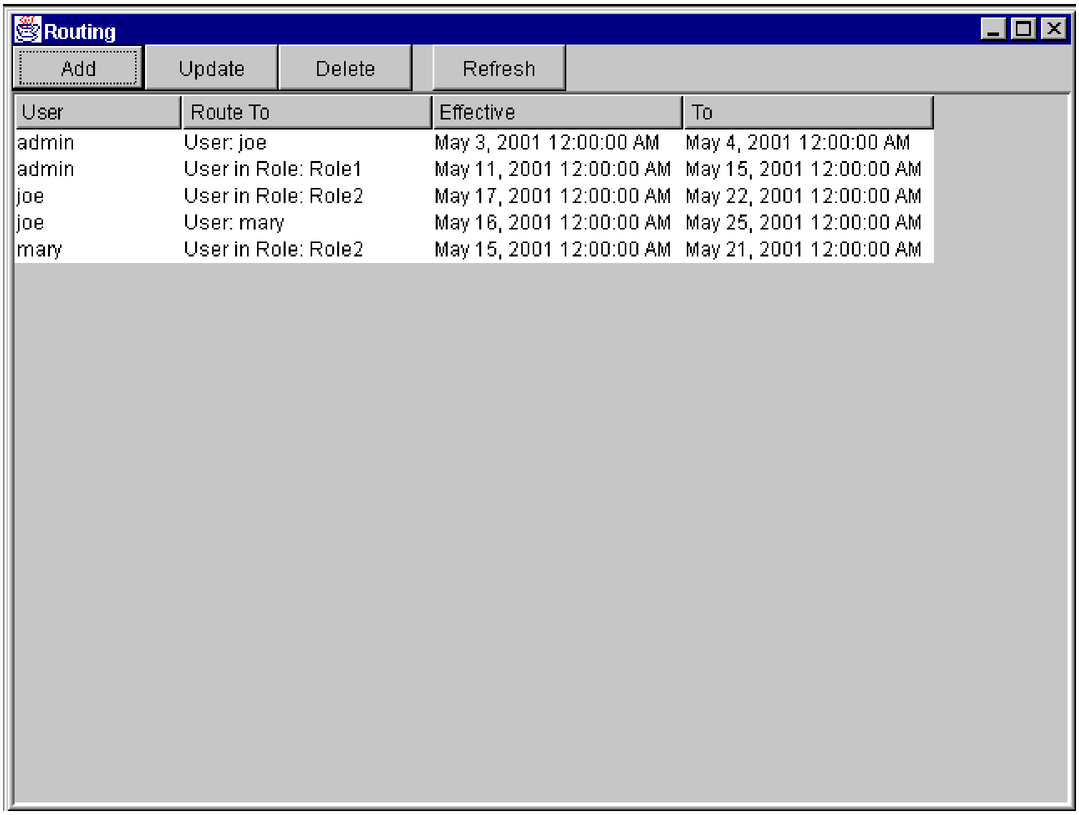
- Use radio buttons when users should select only one of multiple items.
- Use a descriptive label for each radio button.

- Present radio buttons together using a group box. Provide a descriptive label for the group box.
- Align radio buttons vertically.
- Limit the number of radio buttons to six or fewer.

Tables

Tables allow users to enter or view relatively large amounts of information at a time. The following BPM example illustrates appropriate table design.

Figure C-5 Table Design



User	Route To	Effective	To
admin	User: joe	May 3, 2001 12:00:00 AM	May 4, 2001 12:00:00 AM
admin	User in Role: Role1	May 11, 2001 12:00:00 AM	May 15, 2001 12:00:00 AM
joe	User in Role: Role2	May 17, 2001 12:00:00 AM	May 22, 2001 12:00:00 AM
joe	User: mary	May 16, 2001 12:00:00 AM	May 25, 2001 12:00:00 AM
mary	User in Role: Role2	May 15, 2001 12:00:00 AM	May 21, 2001 12:00:00 AM

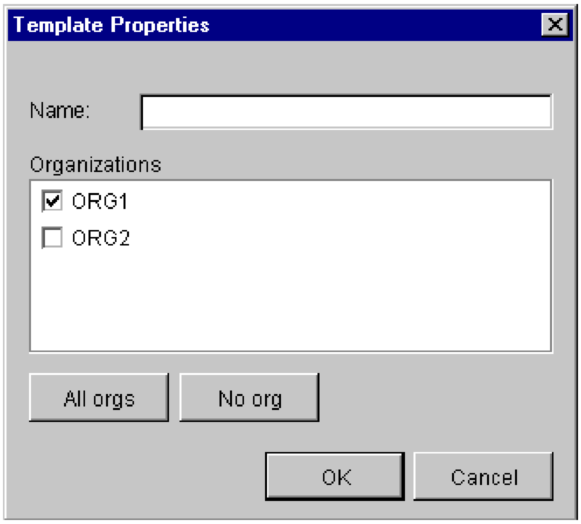
Use the following guidelines to design tables:

- Use tables if users need to compare multiple pieces of data.
- Select column labels that accurately reflect the data.
- Left justify all column labels. Do not use a colon after each column label.

Text-Entry Fields

Text-entry fields are interaction components used most frequently by users to enter data. The following BPM example illustrates appropriate text-entry field design.

Figure C-6 Text-Entry Field Design



Use the following guidelines to design text-entry fields:

- Use a text box with a border to indicate that a user can enter or edit data. For example, in the previous figure, the box labeled Name is a text box.
- Use a field length that signifies the approximate data length.
- Left align fields to minimize the number of different margins.

- Group fields that pertain to similar information in a group box. Provide a descriptive label for the group.
- Assign a descriptive label to each field. Place the labels to the left of the fields and left justify them.

Working with Presentation Components

Presentation components control how data is displayed in dialog boxes. When designing your plug-ins, consider what a user needs to do with the data. For example, do users need to compare bits of information or make selections based on specific criteria? The appropriate display of information can make a major difference in how users perceive the usefulness of an application.

The following sections provide guidelines for designing presentation components:

- Color
- Dialog Box Layout
- Fonts
- Icons
- Messages
- Visual Balance

Color

Color is an important device for getting the user’s attention. When used judiciously, color can enhance the usability of a dialog box.

Use the following guidelines when working with color:

- Have a good reason to use color other than aesthetics. For example, use color to indicate whether the state of a server is up (green) or down (red).

- Choose muted or less saturated shades of each color. Saturated colors often appear too bright on a screen and lead to eye fatigue.
- Be aware of color blindness. Color-blind individuals see red/green and blue/yellow as brown. Therefore, do not rely on color alone to provide critical visual cues.
- Respect the significance of various colors in your target user's culture. For example, in the United States, red signifies hot, danger, and stop. In China, however, red signifies joy and festive occasions.
- Use analogous colors or colors that are adjacent to each other on the color wheel such as green and blue, violet and blue, or neutral colors such as gray and black. Muted versions of colors balance well with any color.

Dialog Box Layout

Dialog box layout is an important aspect of application usability. A layout can influence a customer's perception of whether an application is friendly or unfriendly.

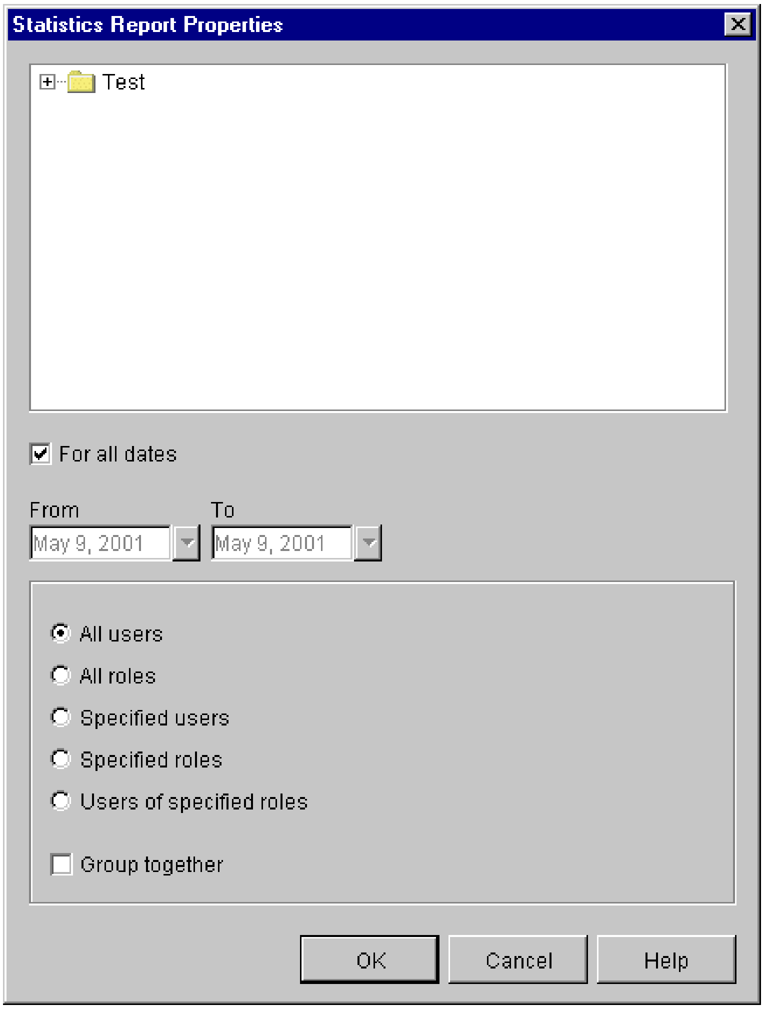
Use the following guidelines to create dialog-box layouts:

- Organize dialog boxes to match a user's workflow. For example, decide which dialog boxes are necessary and in what order they should be presented to support plug-in features.
- Do not crowd dialog boxes with controls for multiple tasks. Each dialog box should represent one task in a user's workflow.
- Select either a horizontal or vertical information flow. It is not necessary to use the same flow for every dialog box. However, do not mix horizontal and vertical flows in the same dialog box.

  A horizontal flow typically results in dialog boxes that are wider than they are tall. In this case, users process information starting in the top left corner and move left to right.

  A vertical flow typically results in dialog boxes that are taller than they are wide. In this case, users process information starting in the top left corner and move top to bottom. The following BPM example illustrates vertical information flow.

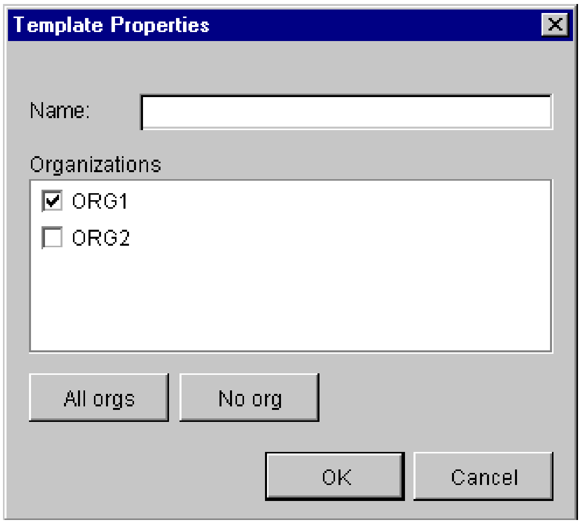
Figure C-7 Vertical Flow



Fonts

Appropriate font selection can improve the readability of control labels in dialog boxes. The following BPM example illustrates an appropriate font selection.

Figure C-8 Font Selection



Use the following guidelines to select fonts:

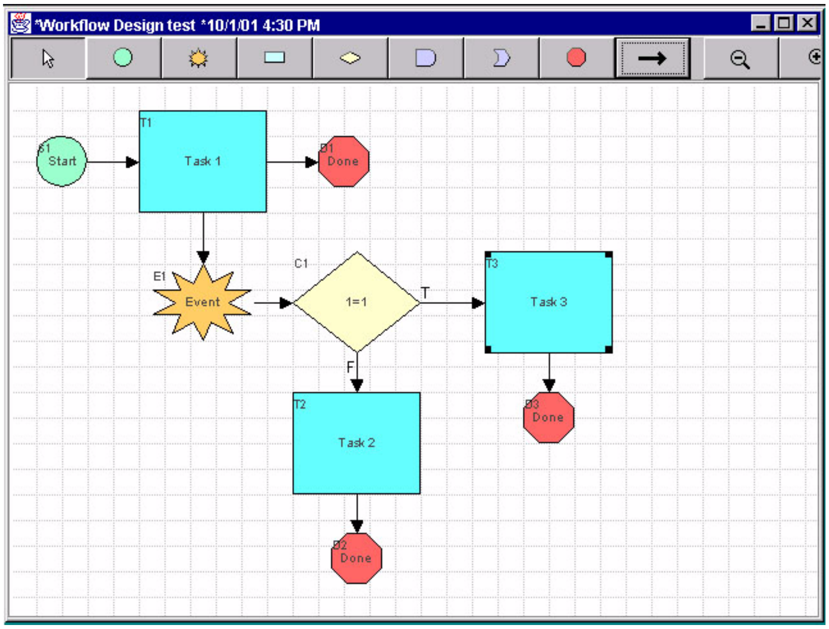
- Use 12-point Arial regular for all control labels.
- Avoid using color fonts for text. The easiest text to read is black type on a white background.
- Avoid using italic or underlining for emphasis.

Icons

Icons play an important role in plug-in design. When a plug-in is loaded, the start and event nodes in the interface view display a 16x16-pixel icon in the upper right-hand corner indicating that a workflow has triggers that are plug-in defined.

Icons are also displayed when assigning plug-in actions to tasks in the interface view. In this case, you can use the default icon provided for plug-in actions or create your own 32x32-pixel icon. The following BPM example illustrates the start, task, event, decision, and done icons in a workflow named Inventory.

Figure C-9 Start, Task, Event, Decision, and Done Icons



Use the following guidelines to create your own 32x32-pixel plug-in action icon:

- Include only enough detail for recognition. Avoid making an icon look like a photograph. Extraneous detail can make icons harder to understand.
- Use one icon to represent the same concept. For example, do not use different representations of a plug-in action in different parts of the interface view.
- Redesign an icon if its meaning is not clear. Add a tool tip to reinforcement the visual image.
- Design icons for an international audience. Focus your design on representations that are universally held. Avoid using offensive gestures. For example, a pointing finger is considered offensive in some cultures.

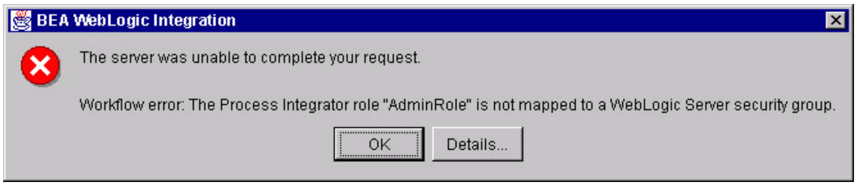
Messages

Application messages are displayed in various contexts and are categorized by severity. For example:

- Informational messages provide users with tips to help make decisions.
- Warnings alert users to conditions that may lead to failure.
- Error messages indicate the failure of one or more components.

The following BPM example illustrates appropriate message design.

Figure C-10 Error Message



The server was unable to complete your request.

Workflow error: The Process Integrator role "AdminRole" is not mapped to a WebLogic Server security group.

When writing message text, it is important to understand what response you expect from users. Use the following guidelines to create messages that are meaningful to users:

- Use terminology with which users are already familiar. Avoid text that is cryptic or difficult to understand.
- Create messages that are concise and specific. Limit the text to two or three short sentences.
- Divide error messages into two parts. The first part should tell the user what error occurred. The second part should tell the user how to recover from the error.

Visual Balance

Balance is the weight of elements in a design relative to the horizontal, vertical, or diagonal axes of the composition. Controls on either side of an axis must be seen to balance each other, whether through equality of size, color, similarity, or placement.

Use the following guidelines to balance the controls in dialog boxes:

- Use command-button placement to balance other controls in a dialog box.

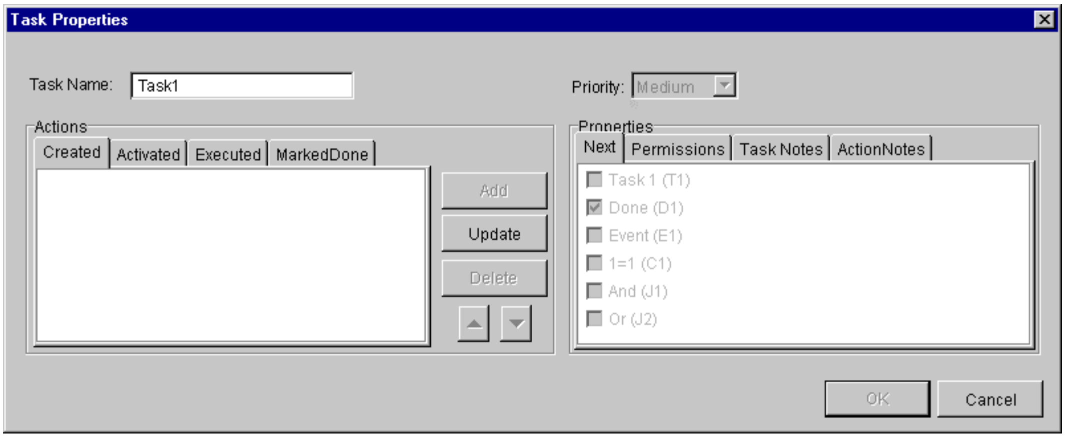
  For horizontal information flows, stack the command buttons vertically and place them along the right-hand side of the dialog box. Experiment by shifting the buttons to the top, middle, or bottom portion of the dialog box to offset larger or darker controls on the left-hand side. Continue adjusting until the buttons and other controls are symmetrical along the vertical axis.

  For vertical information flows, align the command buttons horizontally and place them along the bottom of the dialog box. In this case, shift the buttons to the left or right margin of the dialog box to offset larger or darker controls at the top. Continue adjusting until the buttons and other controls are symmetrical along the horizontal axis.

- Use the golden ratio (1:1.6) whenever possible to size dialog boxes. This ratio has been used extensively in art, architecture, and mathematics (Fibonacci numbers); it produces one of the most visually satisfying of all geometric forms.

- Use a two-column, side-by-side layout for dialog boxes with a horizontal information flow. Balance the placement of controls in each column so the tops and bottoms of individual controls are aligned as much as possible across the columns. The following BPM example illustrates a two-column horizontal information flow.

Figure C-11 Horizontal Layout


Task Properties

Task Name: Task1

Priority: Medium

- [ ] Task 1 (T1)
- [x] Done (D1)
- [ ] Event (E1)
- [ ] 1=1 (C1)
- [ ] And (J1)
- [ ] Or (J2)

Recommended Reading

Review the following sources for more information about designing intuitive and easy-to-use interfaces:

- *About Face: The Essentials of User Interface Design*. Alan Cooper, 1995.
- *GUI Bloopers: Don’ts and Do’s for Software Developers and Web Designers*. Jeff Johnson, 2000.
- *The Elements of User Interface Design*. Theo Mandel, 1997.
- *The Usability Engineering Lifecycle: A Practioner’s Handbook for User Interface Design*. Deborah Mayhew, 1999.
- *Usability Engineering*. Jakob Nielsen, 1994.
- *User and Task Analysis for Interface Design*. Joann Hackos and Janice Redish, 1998.

Index