IBM Tivoli Workload Automation

Developer's Guide: Extending Tivoli Workload Automation

Version 9 Release 1

SC14-7623-02

IBM Tivoli Workload Automation

Developer's Guide: Extending Tivoli Workload Automation

Version 9 Release 1

SC14-7623-02

Note

Before using this information and the product it supports, read the information in “Notices” on page 27.

This edition applies to version 9, release 1, modification level 0 of Tivoli Workload Scheduler (program number 5698-WSH) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Figures

Tables

About this publication

Developer's Guide: Extending Tivoli Workload Automation describes how to use the Web Services Interface to control IBM® Tivoli® Workload Scheduler and IBM Tivoli Workload Scheduler for z/OS® objects in their respective plans.

What is new in this release

For information about the new or changed functions in this release, see *Tivoli Workload Automation: Overview*, section *Summary of enhancements*.

For information about the APARs that this release addresses, see the Tivoli Workload Scheduler Release Notes at http://www-01.ibm.com/support/docview.wss?rs=672&uid=swg27038323 and the Dynamic Workload Console Release Notes at http://www-01.ibm.com/support/docview.wss?rs=672&uid=swg27038328.

What is new in this release for extending Tivoli Workload Automation

This section describes what has changed in this release regarding extending Tivoli Workload Automation since version 8.5 Fix Pack 01.

You now have the possibility to create a custom job-type plug-in. This plug-in adds a new job type to the job definition selection in the Dynamic Workload Console. You can also create an appropriate JSDL file for use in job definition in **composer**. The plug-in also contains a part that runs the job on the dynamic agent.

What is new in this publication

This is a new publication, but it has been developed from those chapters of the *Tivoli Workload Scheduler API Guide* and the topics of the Integration Workbench help of version 8.5 which related to plug-ins. Treat all information in it as new.

Who should read this publication

This publication provides information about how to add functionality to Tivoli Workload Automation products by creating Java™ plug-ins.

The reader of this book should be an *application programmer* expert in Java, who has a reasonable understanding of the Tivoli Workload Automation infrastructure and its inter-component interactions. Alternatively, it should be the manager of such a person, who wants to better understand what you can achieve using plug-ins.

The publication assumes that the application programmer is experienced at creating and working with plug-ins and Java. It also assumes that any product knowledge required to program the API or the web services interface is obtained from the product documentation. This publication does not attempt to explain any of the Tivoli Workload Automation concepts, procedures, and practices to which it refers.

This book also contains information useful to the *IT administrator* and the *Tivoli Workload Automation IT administrator*, for planning purposes.

Publications

Full details of Tivoli Workload Scheduler publications can be found in *Tivoli Workload Automation: Publications*. This document also contains information about the conventions used in the publications.

A glossary of terms used in the product can be found in *Tivoli Workload Automation: Glossary*.

Both of these are in the Information Center as separate publications.

Accessibility

Accessibility features help users with a physical disability, such as restricted mobility or limited vision, to use software products successfully. With this product, you can use assistive technologies to hear and navigate the interface. You can also use the keyboard instead of the mouse to operate all features of the graphical user interface.

For full information with respect to the Dynamic Workload Console, see the Accessibility Appendix in the *IBM Tivoli Workload Scheduler User's Guide and Reference*.

Tivoli technical training

For Tivoli technical training information, refer to the following IBM Tivoli Education website:

http://www.ibm.com/software/tivoli/education

Support information

If you have a problem with your IBM software, you want to resolve it quickly. IBM provides the following ways for you to obtain the support you need:

- Searching knowledge bases: You can search across a large collection of known problems and workarounds, Technotes, and other information.
- Obtaining fixes: You can locate the latest fixes that are already available for your product.
- Contacting IBM Software Support: If you still cannot solve your problem, and you need to work with someone from IBM, you can use a variety of ways to contact IBM Software Support.

For more information about these three ways of resolving problems, see the appendix on support information in *Tivoli Workload Scheduler: Troubleshooting Guide*.

Chapter 1. Introduction to extending Tivoli Workload Automation

Provides an overview and introduction to how you can extend Tivoli Workload Automation.

You can extend Tivoli Workload Automation by creating plug-ins that add functionality relevant to your business activities in two main areas:
- Event-driven workload automation
- Job types with advanced options

In addition, you can create Java jobs that implement a Java project of your creation on the target workstation.

Event-driven workload automation

Event-driven workload automation is a feature of Tivoli Workload Scheduler which you use to trigger Tivoli Workload Scheduler or external actions when Tivoli Workload Scheduler events or external events occur.

Event actions are started automatically when they are triggered because a certain event condition took place. An event condition and the corresponding action are normally defined in an event rule. When a rule is active, this means that a monitoring device runs to detect if the event defined in the rule takes place. When the event is detected, the action defined in the rule is started.

Go to http://publib.boulder.ibm.com/infocenter/tivihelp/v3r1/topic/com.ibm.tivoli.itws.doc_8.5.1.1/awsrgmst84.htm#dqx2evntdrivworkauto for more information about event-driven workload automation.

Tivoli Workload Scheduler comes supplied with a useful set of event conditions and actions which you configure into event rules. However, if these are not sufficient for your uses, you can use the Integration Workbench to create Java plug-ins for event conditions and event actions to perform the required tasks.

For example, suppose you want to send a Java Message Service message when a job stream fails. You can create a Java plug-in to perform this service, implement it in Tivoli Workload Scheduler and combine it in an event rule with the appropriate condition in the Dynamic Workload Console.

Custom job types with advanced options

When you create a job definition you have the choice of job types between those with standard options and those with advanced options. These latter are implemented differently than the job types with standard options, in these ways:
- They require the dynamic agent to be installed on the workstation where they are to be run
- They are implemented by a separate plug-in for each type.

For example, File Transfer type jobs can be run only on workstations where the dynamic agent is installed, and are run by a File Transfer plug-in.

If you have a job with characteristics that cannot be accommodated within the job types predefined in Tivoli Workload Scheduler or Tivoli Workload Scheduler for z/OS (as appropriate), you can create a plug-in with the required characteristics. Once implemented in the product it then becomes available for selection as a job type in the Dynamic Workload Console.

Java jobs

When you define a new scheduling job in Tivoli Workload Scheduler or Tivoli Workload Scheduler for z/OS, one of the job types you can choose is "Java". For each Java job you choose to define, you identify:

- A jar containing the Java classes and methods you want to run on the target workstation (on which the dynamic agent must be installed)
- A set of parameters to be used as input to those classes and methods

The Java project to be run can do more or less what you want it to, but to make this option effective you must follow a set of rules, which are described in this publication.

Chapter 2. Integration Workbench

Describes the Integration Workbench of the Software Development Kit.

IBM Tivoli Workload Automation: Software Development Kit comes with an Integration Workbench which you can use to work with the Java application programming interface and the web services interface to develop your own applications.

This section tells you how to install and use the Integration Workbench help. The help contains detailed information about the tasks you can perform with the Integration Workbench, and the detailed reference information about the methods and classes available:

Installing the Integration Workbench

Gives an overview of the Integration Workbench installation.

The Integration Workbench (part of the Software Development Kit - SDK) runs under Eclipse. The installation, which is fully described in the *Tivoli Workload Scheduler: Planning and Installation Guide*, gives you the opportunity to install the Integration Workbench and a bundled, supported version of Eclipse in one action. Alternatively, you can install the Integration Workbench as an *Eclipse site* using an existing supported version of Eclipse available in your network.

In both cases, at the end of the installation, on the panel where you click **Finish**, there is an option to display the file `readmefirst.html`. This contains information about the workbench, and how to run it. This information is also given here, in "Using the Integration Workbench help."

For more information about Eclipse, go to http://www.eclipse.org/.

Using the Integration Workbench help

Describes how to access the Integration Workbench help facility for the Tivoli Workload Scheduler API and plug-in projects.

To use the Integration Workbench help, do the following:

1. Launch the workbench, as follows:

   Integration Workbench installed with Eclipse

   UNIX Launch the following file: `<`*`TWS_home`*`>/TWS/IntegrationWorkbench/eclipse/eclipse`

   Windows
   Go to **Start → Tivoli Workload Scheduler → Integration Workbench**

   Integration Workbench installed as Eclipse site
   Open your version of Eclipse, as you normally do.
2. Select the location to save your Eclipse workspace. Eclipse requests this every time you run it or the workbench within it, unless you check the option to save a particular location as the default.

3. When the Eclipse window opens, select **Help → Help Contents**
4. Expand **IBM Tivoli Workload Scheduler Integration Workbench**
5. The options shown provide a variety of information about the Tivoli Workload Scheduler Integration Workbench. For example, to see details of all the classes and methods employed in the API, expand **Reference** and select **API reference**.

Note: This information can also be read by opening the following document in a Web browser: `<TWS_home>/TWS/IntegrationWorkbench/readmefirst.html`

Chapter 3. Tivoli Workload Scheduler event management plug-ins

Describes the different types of event management plug-ins.

The two main objects that make up event rules are:

- Event conditions
- Actions

Event condition plug-ins

An event condition is represented by an instance of the `EventCondition` class, which consists of the following:

- A `pluginName`, which identifies the event monitoring plug-in (or provider) that is able to capture the event
- An `eventName`, which identifies the condition in the rule
- An `eventType`, which qualifies the event to be captured
- A `filteringPredicate`, which defines a filter that must be applied to the event content in order to check that it matches the event condition
- A read-only `scope` attribute, which is calculated by the event plug-in when the rule is created or updated. It includes a definition of the scope of the event condition, which has different meanings for different types of events

An event plug-in is a collection of event types grouped because they share similar characteristics, focus on the same operational area, or monitor similar objects.

Event action plug-ins

An action is represented by an instance of the `RuleAction` class, which consists of:

- A `pluginName`, which identifies the action plug-in (or provider)
- One or more specific action types to be run on a specific point in time defined by `responseType` in the rule
- A read-only `scope` attribute, which is calculated by the action plug-in when the rule is created or updated. It includes a definition of the scope of the specific action.

An action plug-in is a collection of action types grouped because they share similar characteristics, focus on the same operational area, or operate on similar objects.

Structure of an event condition or action

In Tivoli Workload Scheduler, event condition and action plug-ins are structured in the following components:

TWSPluginConfiguration.xml
An XML file used to declare the event conditions or actions and the attributes related to them.

TWSPlugin.properties
A properties file that defines the following configurable characteristics of the plug-in:

- The name
- The type (condition or action)
- The Java class name that directly implements the plug-in interface

One or more Java compilation units
One of them contains a class that implements the Java interface for the event plug-in (this class name is defined in TWSPlugin.properties). The Java class can add features to the plug-in and can also generate events or actions. In the sample plug-ins these features are generally included in other compilation units contained in the same Java package.

Event management plug-in project

Describes the event management plug-in project.

A Tivoli Workload Scheduler plug-in project is a special type of Java Project that can be identified in the Navigator view with the following icon:

To simplify the creation in Eclipse of event management plug-ins, the projects have a specific structure and include a "Prepare for deployment" on page 10 action. This action, starting from user-created files in the project structure, packages the plug-in files together in a form ready to be copied onto the event processor.

"Event management plug-in project structure" describes the structure of a Tivoli Workload Scheduler plug-in project and the contents of each folder in the project.

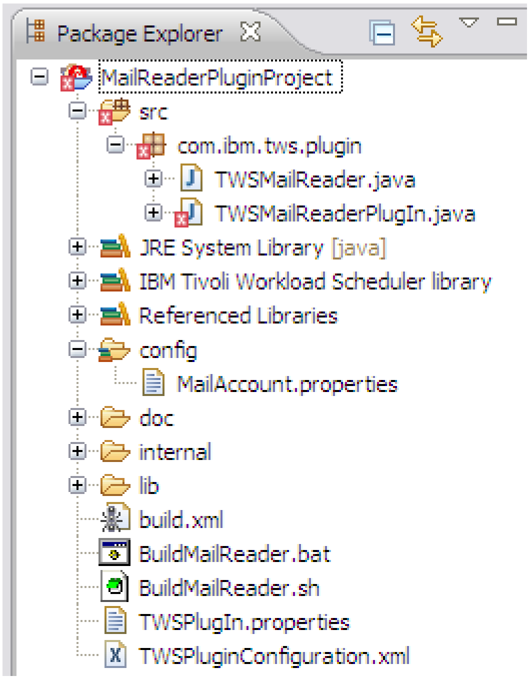
"Prepare for deployment" on page 10 describes how to package your new plug-ins and deploy them to Tivoli Workload Scheduler.

- **"Event management plug-in project structure"**
- **"Prepare for deployment" on page 10**
- **"Generating and sending an event" on page 10**

Event management plug-in project structure

Describes the event management plug-in project structure.

The following figure shows the typical structure of an event management plug-in project:



```
MailReaderPluginProject
  src
    com.ibm.tws.plugin
      TWSMailReader.java
      TWSMailReaderPlugIn.java
  JRE System Library [java]
  IBM Tivoli Workload Scheduler library
  Referenced Libraries
  config
    MailAccount.properties
  doc
  internal
  lib
  build.xml
  BuildMailReader.bat
  BuildMailReader.sh
  TWSPlugIn.properties
  TWSPluginConfiguration.xml
```

The structure shown is for an event condition plug-in named `MailReader`, but event condition and action plug-in projects have the same structure:

- The Java classes contained in the `src` folder are described in "Java source tree (src)."
- The Java Build Path of the plug-in project is described in "Java build path" on page 8.
- The role and contents of other project folders are described in "Other project folders" on page 8.
- The ANT `build.xml` file is described in "build.xml" on page 8.
- The `TWSPlugIn.properties` file is described in "TWSPlugin.properties" on page 9.
- The `TWSPluginConfiguration.xml` file is described in "TWSPluginConfiguration.xml" on page 9.

Java source tree (src)

Describes the event management plug-in Java source tree.

When you create a plug-in project, it is set up as a Java project with separate folders for source and class files. The source folder is named `src`. It contains the Java code of the plug-in.

A Java class is also created together with the new plug-in project. This class implements the Java interface for the event or action plug-in, as appropriate. This class must be present in every event or action plug-in.

Follow the Java documentation (Javadocs) of the interfaces as a reference to figure out which method to implement. For logging and tracing in your code, use standard JSR-047 Java Logging APIs.

Java build path

Describes the event management plug-in Java build path.

Tivoli Workload Scheduler plug-in projects are created with the following libraries:

Default JRE System library
Even if the default JRE is set by default, remember that the IBM Tivoli Workload Scheduler event processor uses IBM JDK version 1.5.

Use of IBM JDK version 1.5 is recommended for Tivoli Workload Scheduler plug-in projects.

IBM Tivoli Workload Scheduler library
This library contains all the Tivoli Workload Scheduler jars required to implement Tivoli Workload Scheduler plug-ins or to use Tivoli Workload Scheduler APIs.

The library also defines the access rules for the classes in the jars: public APIs are defined as Accessible, while internal classes are defined as Discouraged.

Use of discouraged classes is marked with compiler warnings by default. In any case the use of these classes is not supported.

Referenced libraries
These libraries contain all other classes, for example those required, in the case of a mail reader plug-in, for downloading, accessing, and reading mail.

Additional libraries needed for the plug-in Java code can be copied into the `lib` folder and added to the Java build path.

Other project folders

Describes the event management plug-in other project folders.

Beside the standard source and class file folders, the following folders are used by the `Prepare for TWS Deployment` file to package the plug-in for deployment on the event processor.

config Use this folder to store additional configuration files (such as property files) that the Tivoli Workload Scheduler administrator must edit for this operation. The `Prepare for TWS Deployment` action copies these files into the `dist` folder outside the plug-in jar. To access these files from the Java code, load them as resources from the class loader.

doc Use this folder to store the plug-in documentation like Javadocs or documentation about the defined events and actions.

lib Contains additional jar libraries required by the plug-in Java class. The `Prepare for TWS Deployment` action copies these jars into the `dist` folder for use at run time. Jars in this directory must be added manually to the Java Build Path Libraries.

dist This is the target directory of the `Prepare for TWS Deployment` process. After you have run the `Prepare for TWS Deployment` action from the project menu, this folder will contain all the files that must be copied onto Tivoli Workload Scheduler.

build.xml

Describes the event management plug-in `build.xml` file.

This is a standard ANT build file. You can modify it according to your needs, but keep in mind that the `dist` target in the file is used by the `Prepare for TWS Deployment` action.

TWSPlugin.properties

Describes the event management plug-in `TWSPlugin.properties` file.

`TWSPlugin.properties` is a properties file that is read by Tivoli Workload Scheduler when the plug-in is loaded.

When a plug-in project is created, it already contains a `TWSPlugin.properties` file set with the information collected from the new plug-in project wizard. You will not need to modify it unless any of the information is changed later.

`TWSPlugin.properties` defines the following configurable characteristics of the provider:

- The plug-in name. It must be the same value returned by the plug-in Java class with the `getPluginName` method, and the same value defined as plug-in name in `TWSPluginConfiguration.xml`.
- The type: event or action.
- The Java class name that directly implements the plug-in interface.

Example

This is a sample `TWSPlugin.properties` file for a plug-in named `MailReader`.

```
TWSPlugIn.name=MailReader
TWSPlugIn.type=event
TWSPlugIn.className=mycompany.mailreader.MailReaderPlugin
```

"Java source tree (src)" on page 7

TWSPluginConfiguration.xml

Describes the event management plug-in `TWSPluginConfiguration.xml` file.

`TWSPluginConfiguration.xml` is used to declare all the events or actions provided by the plug-in.

The file defines the list attributes for the events or actions, their syntactical constraints, and - for events only - the permitted filtering conditions.

When deployed on Tivoli Workload Scheduler, this file is used to drive the configuration of the event filtering predicates or action attributes inside event rules.

Edit this file to specify the structure of the events or actions provided by the plug-in.

Creating a plug-in project from scratch

Describes how to create an event management plug-in project from scratch.

Using the Integration Workbench you can create plug-in projects from scratch. You supply the basic information about the project you want to create and the wizard does the rest.

Creating a project from a plug-in example

Describes how to create an event management plug-in project from an example.

Using the Integration Workbench you can create projects based on provided examples. Using this method you avoid the need to create the full plug-in project structure from scratch. You choose an example which most approximates your requirements and then modify it accordingly.

The examples you can choose from

Describes the examples available to create an event management plug-in project.

The examples you can select from are as follows:

MailReaderPlugin
Generates a Tivoli Workload Scheduler event when an email is received.

SmsSenderPlugin
Generates and sends an SMS.

Prepare for deployment

Describes how to prepare to deploy an event management plug-in.

This action is available on the menu of Tivoli Workload Scheduler plug-in projects. It runs the `dist` target of the `build.xml` ANT build file in the project.

This action places in the project's `dist` folder the files that you must later copy onto the Tivoli Workload Scheduler event processor. They are:

- A jar file with the Java class files and a copy of `TWSPlugin.properties` and `TWSPluginConfiguration.xml`
- A copy of the user jar libraries present in the `lib` folder, if any
- A copy of the configuration file present in the `config` folder, if any
- A `doc` folder with your plug-in documentation in html

"build.xml" on page 8
"Other project folders" on page 8

Generating and sending an event

Describes the different methods you can use to generate and send an event.

You can use several ways to generate events and to send them to Tivoli Workload Scheduler. Some of them are:

- Coding the instructions directly in the definition of your plug-in
- Using the `sendevent` action externally from the definition of your plug-in in one of the following ways:
  – Use the `sendEvent` method invoked by Java classes
  – Run the `conman sendevent` command

Connect to Tivoli Workload Scheduler

Describes how to connect to Tivoli Workload Scheduler from an event management plug-in.

Connect to Tivoli Workload Scheduler with the following code:

```
private Subject getSubject(String serverName,
                           String serverPort,
                           String uid,
                           String pwd){
 Subject subject = null;
 try {
   LoginContext lc = null;

   Hashtable env = new Hashtable();
   env.put(Context.INITIAL_CONTEXT_FACTORY,INITIAL_CONTEXT_FACTORY);
    env.put(Context.PROVIDER_URL,"corbaloc:iiop:"+serverName+":"+serverPort);

   final InitialContext initialContext = new InitialContext(env);
   Object obj = initialContext.lookup("");


lc = new LoginContext(JAAS_MODULE, new WSCallbackHandlerImpl(uid,pwd));
   lc.login();

   subject = lc.getSubject();

 }   catch(javax.naming.NoPermissionException exc) {
   System.err.println("[TWSConn] - Login Error: "+exc);

 }   catch(Exception exc) {
   System.err.println("[TWSConn] - Error: "+exc);
 }
   return subject;
 }
```

The following user input data is required:
- `serverName`: your Tivoli Workload Scheduler installation name
- `serverPort`: usually 33116 or 33117
- `uid`: your Tivoli Workload Scheduler user ID
- `pwd`: your Tivoli Workload Scheduler account password

Reference material

Describes where to look for reference material on event management plug-ins.

The Integration Workbench help contains all the reference material you require.

To access this material, take the following steps:
1. From the Integration Workbench select **Help → Help Contents**
2. Expand **Tivoli Workload Scheduler Integration Workbench** and then **Reference**
3. Obtain reference material for any of the following:
   - What information is required to run the wizards that create plug-in projects, either from scratch or from an example
   - Information about the libraries of object and runtime jars
   - Description of the XML schemas
   - A link to information about the Tivoli Event Integration Facility
   - Full reference for every Java class and method used in the plug-in

Chapter 4. Custom job type with advanced options plug-in

Provides an overview of the custom job-type with advanced options plug-in.

In Tivoli Workload Scheduler and Tivoli Workload Scheduler for z/OS, you can create jobs of different types. The job type determines the characteristics of the job. For example, a file transfer type job has all the characteristics necessary to perform a file transfer. When you select to create a job of this type, you are invited to enter the file transfer parameters (which file, from where, to where) in a dedicated panel on the Dynamic Workload Console.

When you choose a job type in the Dynamic Workload Console the job types are divided into two groups:

Jobs with standard options
The basic jobs that run scripts and commands.

Jobs with advanced options
These run more specialized jobs. They are implemented with a plug-in, and can be run only on workstations where the dynamic agent has been installed, even if they are not to be scheduled dynamically. In the Dynamic Workload Console, you choose from a set of these job types which cover the more common activities that an organization might want to run (for example, File Transfer, Web Services, Database, and Java). However, if you want to schedule a type of job that has different requirements from any of those pre-defined job types, you can create your own plug-in to add a new job type appropriate to your business needs. Do this with the Integration Workbench.

To perform the operations included in custom job-type with advanced options plug-ins, you need the **run** access defined in the security file on the workstation on which you plan to run the plug-in. The following conditions apply:

- If the operation is performed on the Tivoli Workload Scheduler Connector, display and run access is required on the CPU corresponding to the workstation where the job is created.
- If the operation is performed on the workstation where the job runs, display access is required on the workload broker workstation.

See the *Tivoli Workload Scheduler: Administration Guide* for more information about configuring the security file and about defining the `run` and the `object` access keywords.

The anatomy of a custom job-type plug-in

A custom job-type plug-in consists of:

- A Dynamic Workload Console panel where the user enters the job definition details
- The server part of the plug-in which performs optional validation and creates the job definition
- The agent part of the plug-in which runs the job on the target workstation

The steps to create a custom job type plug-in

The creation of a custom job type plug-in requires the following steps:

1. Design a graphic interface panel

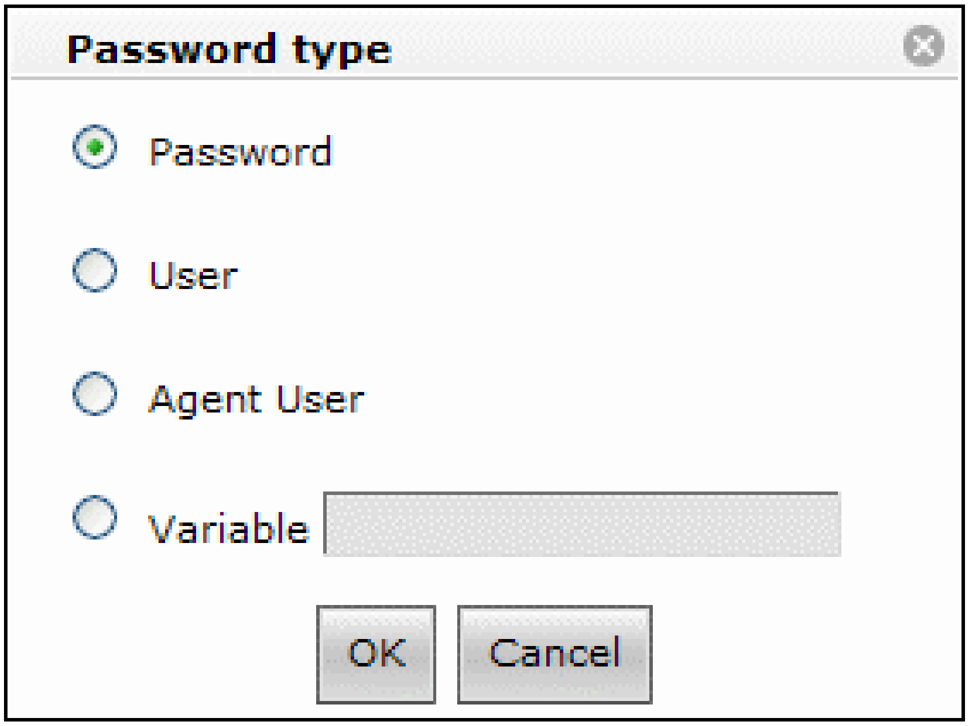
You generate your own panel for the Dynamic Workload Console to be used as an input form for your custom job type. You use the Abstract User Interface Markup Language (AUIML) VisualBuilder tool to create a user interface panel. You can add:

- Buttons
- Check boxes
- Combo boxes
- Edit boxes
- List boxes
- Groups
- Radio boxes
- Dynamic lists
- Dynamic pairs lists

Note the following restrictions:

- When you name your GUI objects, do not use the `tws` prefix which is reserved for internal use. For the same reason, avoid using the following terms as distinct names (but they can be parts of strings):
  - name
  - member
  - workstation
  - job
  - jobstream
- No two level of nesting is allowed for check box or radio groups. Also, you cannot place required fields in two level nested groups.

If you include `user name` and `password` fields in your panels, you can take advantage of the following widget (provided with the product) that enables the user (who would define the job in most cases) to choose the source for the requested password. The widget is displayed by clicking on an ellipsis (...) button placed next to the password field and it looks as follows:



where the choices are as follows:

Password
Takes the password value entered in the `Password` field.

User

Adds the `${password:`*`value_of_user_name_field`*`}` string in the password field.

On dynamic agents
It is resolved at runtime with the password value defined for `User Name` in the Tivoli Workload Scheduler database using either the `User` definition panel or the `composer user` command.

You can also specify the user (and its password) of another workstation if it is defined in the database. See the description of the **Variable** button.

Attention: User definitions lack referential integrity. This implies that, if a user definition referenced in the credentials section of a job is changed or deleted, no warning or error message is returned until the job is run.

On Tivoli Workload Scheduler for z/OS agents
It is resolved at runtime with the password value defined for `User Name` in the Tivoli Workload Scheduler for z/OS database using the USSREC initialization statement, where the value of `User Name` is defined by the `USRNAM` parameter and the password by `USRPSW`.

Agent User

Adds the `${agent:password.`*`value_of_user_name_field`*`}` string in the password field.

It is resolved at runtime with the password value defined for `User Name` locally on the dynamic agent or Tivoli Workload Scheduler for z/OS agent that will run the job (or on any agent of a pool or dynamic pool that may run the job) with the `param` command.

Variable

It is resolved at runtime with the value defined for the variable you enter in the field (using the ${*variable_name*} notation).

On dynamic agents

The variable must have been defined either locally on the agent, using the `param` command, or in the Tivoli Workload Scheduler database, utilizing the `User` panel or the `composer username` command. For example:

- A variable defined locally on the agent, enter here as:
  ```
  ${agent:file_With_Sections.password.dbPwd}
  ```
- A variable defined in the database, enter here as:
  ```
  ${password:workstation#user}
  ```

You can use this button to specify the password of the remote user of a different workstation (as long as it was defined in the database) by entering the following string in the adjacent field:

`${password:`*`workstation_name#value_of_user_name_field`*`}`

On Tivoli Workload Scheduler for z/OS agents

Use this field if you want to utilize the password defined for a user different from the one specified in the `User Name` field.

For example if you are defining a File Transfer job and the local and remote user names are identical (`user1`), you can differentiate the password by defining two USRREC initialization statement entries (for example one for `user1` and one for `user1remote`). After doing this, in the remote user password field you specify:

```
${password:user1remote}
```

The traditional variable substitution mechanism which uses variables defined in the Tivoli Workload Scheduler for z/OS database variable tables is not supported in this field.

Variables are resolved both when you generate a plan and when you submit a job or a job stream. While defining jobs, the variables are not resolved and cannot be used in lists or for test connections.

You must add the following statement in the coding of the password field to display the widget and associate the chosen option to the field:

```
BINDING="passwordSelector#modeling.widgets.password.PasswordSelector:
  <userName>"
```

2. Optionally add buttons to the panel to perform service actions

You can add buttons to your custom-made panels in the Dynamic Workload Console, for example to perform the following service actions:

- Open a pick list of possible values for a job parameter from the connector
- Open a pick list of possible values for a job parameter from an agent

When you add the button you associate the corresponding action in an ACTION field within the plug-in. The definition of an action requires the following:

- A definition of the action in the AUIML plug-in definition
- An option which determines whether the action is to be performed on the connector (as an immediate and direct result of clicking the button)
- A method which performs the required action

One or more of the following actions can be triggered:

- A field can be put in an error state with a message explaining what is wrong
- A pop-up menu can be displayed
- Diagnostic information can be traced on the Dashboard Application Services Hub
- A pick list can be displayed

3. Create a plug-in project

Use the Integration Workbench to generate a plug-in project.

This step consists in creating a package with your plug-in files and all you need to deploy them. Do the following:

1. On the toolbar select **File→New→Project...**. The `New Project` window opens.
2. In the list of wizards double-click first on **IBM Tivoli Workload Scheduler** and then on **TWS Executor Project**. The `TWS Executor Project` wizard opens.
3. Enter a name for your project.
4. Enter a package name for the plug-in.
5. Enter the name of the job executor related to the plug-in.
6. Enter the *panel_filename*.AUIML name (or use the search tool).
7. Enter the names of the one or more *panel_filename*.properties files (or use the search tool to select them).
8. Enter the name of the related icon file (or use the search tool). The icon file can be of type PNG only.
9. Click **Finish**.

The wizard creates the executor project and generates:

- An .XSD file that defines the schema of the job definition
- All the classes required to map the JSDL file
- A labels.properties file where you can insert your labels

To insert more labels, edit labels.properties located in the RESOURCES folder for the project. The file already includes a default label. Add each localized label in the format:

```
language code = label
```

using the same locale used in the corresponding properties file.

The plug-in ID is the plug-in name in lower case.

Edit the plug-in project files

Edit the files created for the project:

- Give a name to the panel that will be shown in the **New ▸ Job Definition** drop-down menu on the Dynamic Workload Console (you can add localized names as well)
- Set the plug-in version if you do not want to use the default of 1.0.0
- Add messages, including any localized messages, that you want to refer to in your coding
- Code any validation that you want performed, both on input and on execution
- Create pick lists for field values
- Code any special output requirements you need for this job type

4. Test the plug-in

Use the test project (automatically generated by the `TWS Executor Project` wizard when you created the plug-in project) to test the plug-in.

5. Deploy the plug-in

Export the packaged plug-in files from the Eclipse workbench to a folder in your, or another, computer so that you can then install them on the agents and make them operational. Do the following:

1. In the Package Explorer panel right click on the package name and select **Export...**.
2. In the Export window double-click first on **IBM Tivoli Workload Scheduler** and then on **TWS Executor**. A list of defined TWS Executor projects is displayed.
3. Select the project in the list and enter the destination folder in your or another computer.
4. Click **Finish**.

This results in the creation of a jar file named *plug-in_id_version*.jar (where *plug-in_id* is the name of your plug-in in lower case) in the destination folder of the computer you selected. This name cannot be modified.

6. Install the plug-in on the master domain managers, dynamic domain managers, and related backup workstations, if any

Copy the jar file of the plug-in into the `installation_dir/TWS/applicationJobPlugIn` path master domain managers, dynamic domain managers, and related backup workstations, if any. Restart the WebSphere Application Server to make the changes effective.

7. Install and configure the plug-in on the agents

Do the following:

1. Copy the plug-in into the `../TWA/TWS/JAVAEXT/eclipse/plugins` path of every dynamic agent where you want to run it.
2. On every agent edit the `config.ini` file located in path `../TWA/TWS/JAVAEXT/eclipse/configuration` by adding the line:

```
plug-in_id@4:start
```

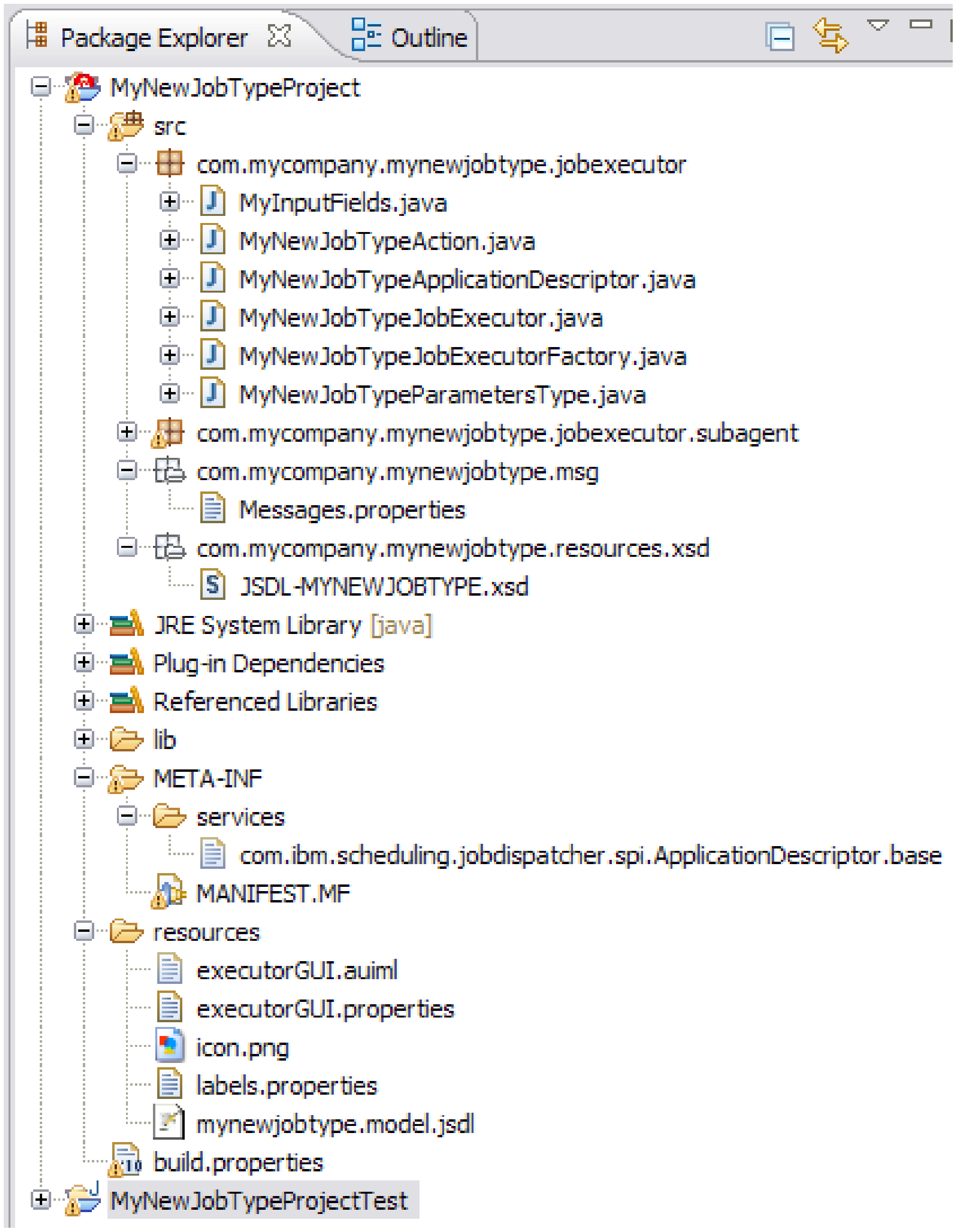
   where `plug-in_id` is the name in lower case of your plug-in.

Note: You can install only one version of the same plug-in on the master domain manager, on the dynamic domain manager, and on the agents where you want the plug-in to run. If you create a new version of the plug-in, install it on the master domain manager, on the dynamic domain manager, and on the agents and remove the previous version.

Structure of custom job-type plug-in

Provides an overview of the structure of a custom job-type plug-in.

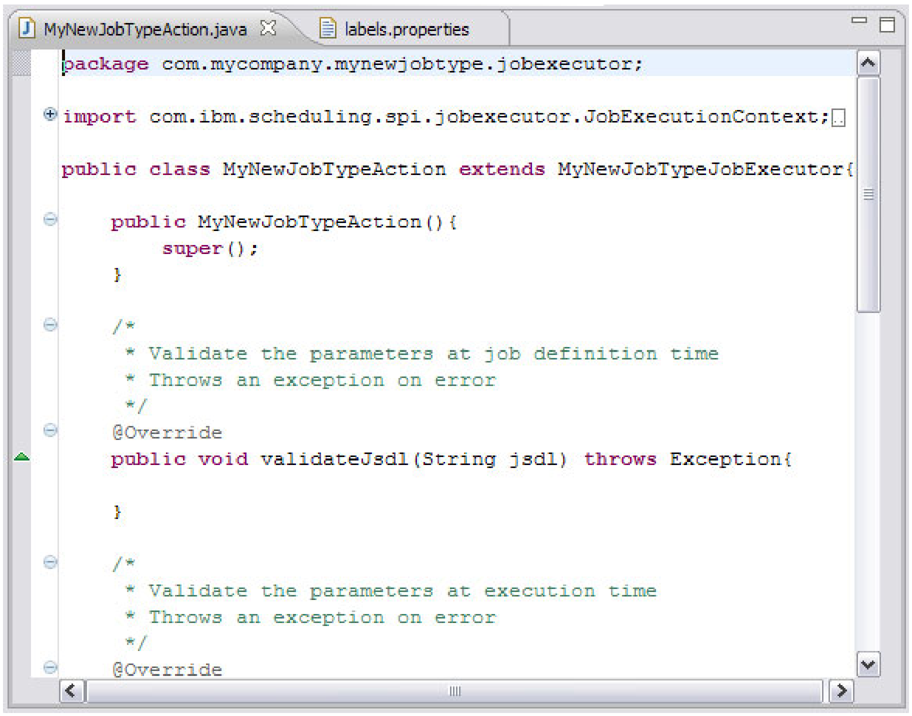
When you create a custom job-type plug-in project in the Integration Workbench, you see something similar to the following figure:



```
MyNewJobTypeProject
    src
        com.mycompany.mynewjobtype.jobexecutor
            MyInputFields.java
            MyNewJobTypeAction.java
            MyNewJobTypeApplicationDescriptor.java
            MyNewJobTypeJobExecutor.java
            MyNewJobTypeJobExecutorFactory.java
            MyNewJobTypeParametersType.java
        com.mycompany.mynewjobtype.jobexecutor.subagent
        com.mycompany.mynewjobtype.msg
            Messages.properties
        com.mycompany.mynewjobtype.resources.xsd
            JSDL-MYNEWJOBTYPE.xsd
    JRE System Library [java]
    Plug-in Dependencies
    Referenced Libraries
    lib
    META-INF
        services
            com.ibm.scheduling.jobdispatcher.spi.ApplicationDescriptor.base
        MANIFEST.MF
    resources
        executorGUI.auiml
        executorGUI.properties
        icon.png
        labels.properties
        mynewjobtype.model.jsdl
    build.properties
MyNewJobTypeProjectTest
```

The following sections give some outline information about how to modify the project for your needs.

<*ProjectName*>**Action.java**



```
package com.mycompany.mynewjobtype.jobexecutor;

import com.ibm.scheduling.spi.jobexecutor.JobExecutionContext;

public class MyNewJobTypeAction extends MyNewJobTypeJobExecutor{

    public MyNewJobTypeAction(){
        super();
    }

    /*
     * Validate the parameters at job definition time
     * Throws an exception on error
     */
    @Override
    public void validateJsdl(String jsdl) throws Exception{

    }

    /*
     * Validate the parameters at execution time
     * Throws an exception on error
     */
    @Override
```

This file contains the classes that control the job validation. As can be seen from the screen capture, you can code validation to be performed when the job is defined (typically, semantic validation) and when the job is run (for example checking that a file exists).

<*ProjectName*>**ApplicationDescriptor.java**

In the application descriptor, you can manually add the `category` property to describe the type of plug-in.

The property must have the ID of a predefined category, or a user-defined value. The IDs of the predefined plug-in categories are:

- native
- ERP
- database
- file_transfer
- cloud
- system_management
- business_analytics

If your plug-in does not fall into any of the predefined categories, you can enter a new category and it will be added to the list of plug-in types in the Workload Designer section of the Dynamic Workload Console.

The following example shows the application descriptor of a plug-in that falls within the `database` category:

```
EJB Application descriptor service provider
com.ibm.scheduling.agent.database.jobexecutor.DatabaseApplicationDescriptor
application=database
factory=com.ibm.scheduling.agent.database.jobexecutor.DatabaseJobExecutor
FactorysupportedWorkstations=agent,pool,d-pool
supportedOS=UNIX,WNT,OTHER,IBM_i
isJobStoppable=true
producesJobOutput=true
category=database
```

```
schemaLocation=com/ibm/scheduling/resources/xsd/JSDL-DATABASE.xsd
namespace=http://www.ibm.com/xmlns/prod/scheduling/1.0/jsdldatabase

version=${agent.executors.ver}
label=Database
```

Father of <*ProjectName*>Action.java
Contains various useful material, such as:
- The logger
- A method for discovering the job log name
- The possibility of customizing the success message label
- The possibility to change the exit codes
- Methods for canceling the job

com.<*CompanyName*>.<*ProjectName*>.msg
Contains the properties file where you add the messages. Messages can be localized by adding the locale code.

JSDL-<*PROJECTNAME*>.xsd
Contains the XML schema of the panel you created. It is not normally necessary to edit the schema, but it can be a useful point of reference.

MANIFEST.MF
This file records the version of your plug-in project.

labels.properties
Where you assign the name of your job type, as you want it to be displayed on the **Create Workload Definition** ▸ **New** ▸ **Job Definition** drop-down in the Dynamic Workload Console. Localized versions can also be specified here, and are automatically used when the console is in use on systems with the corresponding locale set.

Developing the plug-in

During the process of development, you might decide to modify the input data panel. In this event, you can use the option to regenerate the plug-in using the modified panel. Any coding you added is treated in the following way:
- If you removed a field from the panel, you must manually delete any associated coding that you added.
- If you modified a field in the panel, you must check if any associated coding still applies and amend it if not.
- If you added fields, supply any additional coding required.

Testing the plug-in

A test project is automatically created when you create the plug-in project. This contains a file called `job.xml`, which can be launched as a Java application to run a Tivoli Workload Scheduler job. Alternatively, the file can be used as a job definition template in **composer**.

Chapter 5. Defining Java jobs

Describes how to create Java jobs which extend the capability of Tivoli Workload Automation.

To define a job that runs a Java job by using the Dynamic Workload Console, perform the following procedure.

1. In the console navigation tree, expand **Workload** > **Design** and click **Create Workload Definitions**
2. Specify an engine name, either distributed or z/OS. The Workload Designer opens.
3. In the Working List pane, select the Java job definition.

   z/OS

   New > **Database and Integrations** > **Java**

   I .IB

   New > **Job Definition** > **Database and Integrations** > **Java**

   The properties of the job are displayed in the right-hand panel for editing.
4. In the properties panel, specify the attributes for the job definition you are creating. You can find detailed information about all the attributes in the contextual help available on each panel.
5. Click **Save** to save the job definition in the database.

When you define a job of this type you supply the following:

- A jar prepared by you, containing the classes and methods you want to implement when the job is run
- A set of parameters which provide runtime information to the job

The following sections tell you how to prepare the .jar. They also describe how you can use the **Get Class Information** button, which is displayed on the interface, to obtain information about the classes in the selected jar.

Creating Java job jar

Describes how to create the jar used with a Java job.

A Java job can use any jar that is created according to the following rules and procedure.

1. Include Tivoli Workload Automation jar in build path

The class which implements the Dynamic Workload Console interface is in the following jar:

On the server
: `<`*TWA_home*`>/TWA/TWS/applicationJobPlugIn/com.ibm.scheduling.agent.java_<`*version*`>.jar`

On a dynamic agent
: `<`*TWA_home*`>/TWA/TWS/JavaExt/eclipse/plugins/com.ibm.scheduling.agent.java_<`*version*`>.jar`

where *<version>* is the Tivoli Workload Scheduler version related to the latest fix pack applied.

This jar must be included in your Java build path.

2. Create your class which implements TWSExecutable

Create a class which implements a class called `TWSExecutable`, which has a signature as follows:

```
public abstract interface TWSExecutable
{
  public abstract void validateParameters(Parameters paramParameters)
    throws Exception;

  public abstract void execute(Parameters paramParameters)
    throws Exception;
}
```

Note: This class can be used on agents either at V8.5.1 Fix pack 1 or V8.6.

This class implements two methods:

validateParameters
This is the method which is called first when a Java job is executed. It validates the parameters input when the job was defined. If an exception occurs it is written to the job log.

execute
This is the method which actually runs the job.

The following are methods of the `validateParameters` class:

getParameter
Supplied with the argument of one of the parameters, it returns the actual value.

getParameterList
Returns a list of all the parameters defined as name/value pairs.

getOutputFile
Returns the path of the output job log.

Note: If you want to write to the log you must always remember to close it.

3. Optionally supply information to the person defining the job

The interface includes a button **Get Class Information**. To implement this button use the class `TWSExecutableInformation`, which has a signature as follows:

```
public abstract interface TWSExecutableInformation
{
  public abstract String getInformation();
}
```

Note: This class can be used only on V8.6.

This class is implemented when the user clicks the **Get Class Information** button, so you can use it to supply help information about the job in the jar that the user has selected.

4. Include any required libraries

If your Java job requires any libraries or other files, copy them to the folder where you have saved the jar. When the person defining the job identifies the jar, the product loads not just the jar, but also everything else in the folder on the agent.

The following is an example of the code discussed in this topic:

```
package com.test.Trial;

import java.io.BufferedWriter;
import java.io.FileWriter;

import com.ibm.scheduling.agent.java.jobexecutor.TWSExecutable;
import com.ibm.scheduling.agent.java.jobexecutor.TWSExecutableInformation;
import com.ibm.scheduling.agent.java.parametersdomain.Parameters;

public class Trial implements TWSExecutable, TWSExecutableInformation{

  /*
   * Writes the parameter "parm1" to the log
   */

 @Override
 public void execute(Parameters arg0) throws Exception {
  String parm1 = arg0.getParameter("parm1");
  String filename = arg0.getOutputFile();

  BufferedWriter out = new BufferedWriter(new FileWriter(filename));
  out.write(parm1);
  out.close();

 }

  /*
   * Validates the parameter "parm1", throwing the exception to the log
   */

 @Override
 public void validateParameters(Parameters arg0) throws Exception {
  String parm1 = arg0.getParameter("parm1");
  if(parm1.equals("XXX"))
   throw new Exception("The parameter parm1 is not correct");

 }

  /*
   * Tells the user of the interface who has clicked the Get Class
   * Information button what the class does.
   */

 @Override
 public String getInformation() {
  String msg = "This class writes the parm1 parameter in the output log";

  return msg;
 }

}
```

Notices

Trademarks

Index

Product Number: 5698-A17, 5698-WSH, and 5698-WSE

Printed in USA

SC14-7623-02