Developing Custom Controls

BEA Workshop for WebLogic Platform allows you to create custom controls tailored to your project or application. Custom controls can be used to create re-usable controls that might be found in a company for sharing, or those provided by ISVs for their products. This section explains how to create these controls and how to share them.

For a complete overview of controls in Workshop for WebLogic, including how to create them, see Getting Started with Beehive Controls.

Topics Included in This Section

Creating Custom Controls
: Describes the basics of creating and using custom controls.

Source Files for Custom Controls
: Describes the files that are necessary in any custom control.

Testing Controls
: Discusses how to test custom controls.

Exporting Controls into JARs
: Describes how to export controls into a JAR file that can be shared.

Distributing Controls as Plug-Ins
: Shows you how to customize controls more extensively and how to package/distribute controls for a wider audience.

Related Topics

Using System Controls

Creating Custom Controls

This topic describes how to use a custom custom control. It explains how to:

- Create a custom control
- Use a custom control in your application

Custom control files can be located:

- In your web project.
- In a utility project. To access such controls in a web application, both the web project and the utility project must be linked to the same EAR project.

To Create a Custom Control

The following instruction assume you are in the J2EE perspective (**Window > Open Perspective > J2EE**).

1. You cannot create a control in the default package. So the first step is to create a package for the control. For example:

   <ProjectRoot>/src/controls/myControl/

2. Right-click the package and select **New** > **Custom Control**.

3. In the **Control name** field, enter the class name for the control.

   The Java interface and implementation classes will be based on the name entered here. For example, if you enter Hello, two classes will be created:

   Hello.java (=the interface class)

   and

   HelloImpl.java (=the implementation class)

4. Click **Finish**.

Default control interface and implementation classes are produced. Assuming that your control is named Hello, the following class files are produced:

Hello.java Interface Class File

```
package controls.myControl;
```

```
import org.apache.beehive.controls.api.bean.ControlInterface;

@ControlInterface
public interface Hello {

}
```

HelloImpl.java Implementation Class File

```
package controls.myControl;

import org.apache.beehive.controls.api.bean.ControlImplementation;
import java.io.Serializable;

@ControlImplementation
public class HelloImpl implements Hello, Serializable {
        private static final long serialVersionUID = 1L;

}
```

Continue the composition of the custom control by adding methods to these class files.

To Use a Custom Control in an Application

If you have an existing custom control in your project or in a utility project in the current workspace, you can add a reference to that control to a control client by right-clicking anywhere within the client's Java source file and selecting **Insert > Control**.

A list of available controls appears. The heading **Existing Project Controls** lists the controls in the same project as the client. The heading **Existing Application Controls** lists the controls in the utility projects in the same workspace.

When you add a control reference to a client, Workshop for WebLogic Platform modifies your client's source code to include an annotation and variable declaration for the control. The annotation ensures that the control is recognized by Workshop for WebLogic Platform, and the variable declaration gives you a way to work with the control from your client code. For example, if you add a new custom control named `Hello`, the following code will be added to your client:

```
    import org.apache.beehive.controls.api.bean.Control;
    import controls.myControl.Hello;

        @Control
        private Hello hello;
```

Once you have a reference to a control, your client can call methods on that control. For more detail on calling a control method, see Invoking a Control Method.

Related Topics

Invoking a Control Method

Source Files for Custom Controls

Source Files for Custom Controls

Custom controls consist of two Java source files: an **interface** class file and an **implementation** class file.

The interface class contains the control's publicly accessible methods. Clients of the control call the methods in the implementation class.

The implementation class contains the control's behind the scenes implementation code.

There is also a third class associated with each custom control: the **generated JavaBean class**. This is a build artifact created from the interface and implementation source files. The generated JavaBean class provides supplemental programmatic access to the control, especially the ability to override default annotation values in the control. For more information about this class see Overriding Control Annotation Values Through the Control JavaBean

Custom Control Interface Classes

A custom control interface class must be decorated with the `@ControlInterface` annotation.

```
package controls.hello;

import org.apache.beehive.controls.api.bean.ControlInterface;

@ControlInterface
public interface Hello {

    ...

}
```

The `@ControlInterface` annotation informs the compiler to treat this class as a part of the Beehive Control framework.

The interface class also lists the control's publicly available methods. The following example shows a control with one publicly available method.

```
package controls.hello;

import org.apache.beehive.controls.api.bean.ControlInterface;

@ControlInterface
public interface Hello {

        public String hello();

}
```

Custom Control Implementation Classes

A custom control implementation class contains the control's logic - the code that defines what the control does. In this file you define what each of the control's methods do.

The minimum requirements for a custom control implementation class are listed below.

1. The class must be decorated with the `@ControlImplementation` annotation.

```
import org.apache.beehive.controls.api.bean.ControlImplementation;

@ControlImplementation
public class HelloImpl
```

2. The class must implement the corresponding custom control interface file.

```
import org.apache.beehive.controls.api.bean.ControlImplementation;

@ControlImplementation
public class HelloImpl implements Hello
```

3. The classes must either:

   (a) implement java.io.Serializable

```
import java.io.Serializable;

@ControlImplementation
public class HelloImpl implements Hello, Serializable
```

   (b) or set @ControlImplementation(isTransient=true)

```
@ControlImplementation(isTransient=true)
public class HelloImpl implements Hello {

}
```

Related Topics

Controls: Getting Started

Testing Controls

Beehive controls can be tested either inside of an application container or outside in a standalone Java environment. Testing in a standalone Java environment is especially useful when running unit tests.

Beehive controls can be integrated into the JUnit test framework using the ControlTestCase base class. This base class provides a control container and provides help in instantiating a control declaratively via the
@Control annotation.

Note that not all controls can be tested within the test container because some controls have requirements beyond what ControlTestCase provides. For example, a control that uses JNDI lookups will not be testable with ControlTestCase. Likewise controls (such as the Service Control) that take a dependency on a J2EE container (such as WebLogic Server) may not be testable out of that J2EE container.

For details on testing controls with ControlTestCase see Control Tutorial: Testing Controls with JUnit.

Related Topics

Testing Controls with JUnit

Exporting Controls into JARs

Workshop for Weblogic Platform lets you package your control classes as JAR files that can be reused in other Java projects. This is the simplest way to distribute controls.

This approach is somewhat limited, providing no custom labels, no custom icons, no insertion wizards. If you are creating controls that will have very wide distribution (e.g., an ISV developing controls for customers), you may want to package your custom control as a plug-in.

To package a Beehive control as a JAR file, select **File > Export > Beehive Control JAR File**.

Only control files in utility projects are available for JAR file packaging; controls in other project types are not available for export.

All Java class files in the utility project are included in the JAR file, including control interface, control implementation classes, and all other Java classes. Note that by default, **only** class files are included in the JAR file. To include the Java source files, place a checkmark next to **Include Java source files**.

To use a control in another web application:

1. Copy the JAR file to the WEB-INF/lib folder.
2. Add a reference to that control to a control client by right-clicking anywhere within the client's Java source file and selecting **Insert > Control**.
3. A list of available controls appears. The heading **Existing Project Controls** lists the available controls, including controls in JAR files.

Alternately, you can:

1. Copy the JAR file to the APP-INF/lib folder of the associated EAR project.
2. Add a reference to that control to a control client by right-clicking anywhere within the client's Java source file and selecting **Insert > Control**.
3. A list of available controls appears. The heading **Existing Application Controls** lists the available controls, including controls in JAR files.

As long as the JAR is inserted into the user's classpath as described above, the control will be discovered automatically by Workshop for WebLogic and property set/event handler features will be provided.

Related Topics

Apache Beehive Documentation

Building Controls

Distributing Controls as Plug-ins

If you want to distribute your custom control to a wide audience (e.g., if you are an ISV developing controls for your customers) or if you want to customize your control more extensively, you can package a control within a plug-in. This method allows:

- Customized label and icon
- Customized insertion wizard

This topic describes how to package a control into a plug-in. This method creates an Eclipse plug-in and basic knowledge of Eclipse plug-ins and their creation would be useful before attempting this process.

Note that this method wraps a control JAR in a plug-in. For distribution within your own company, you may simply want to share the control JAR file directly, without the additional work of creating a plug-in.

This method consists of the following steps:

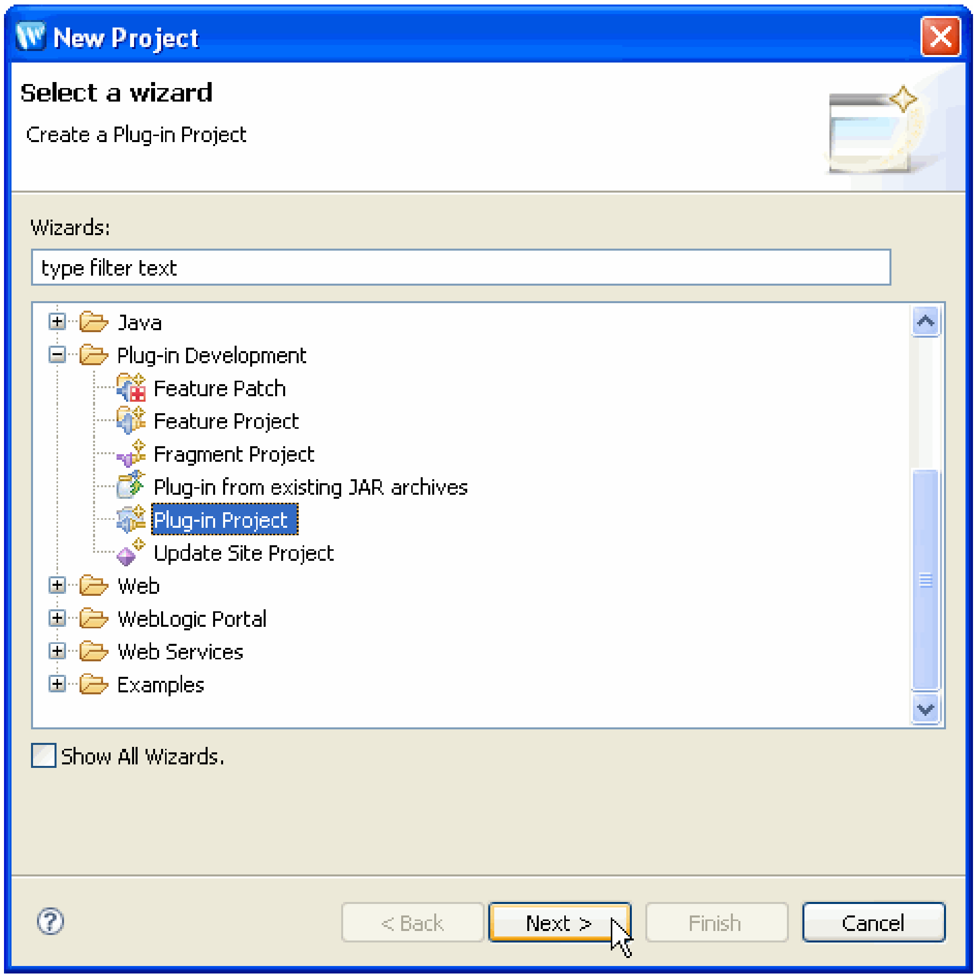
1. Export the control into a control JAR
2. Create the control plug-in project
3. Copy the control JAR into the plug-in project
4. Set plug-in project dependencies
5. Add extension and customize settings
6. Create the insertion delegate code
7. Build and test your plug-in
8. Export the plug-In

Step 1: Export the Control into a Control JAR

Follow the steps outlined in Exporting Controls into JARs.

Step 2: Create the Control Plug-in Project

1. Create a plug-in project with **File > New > Project**. Expand **Plug-in Development** and choose **Plug-in Project**. If you do not see the correct project type, you may need to click **Show All Wizards** to display it.



New Project

Select a wizard

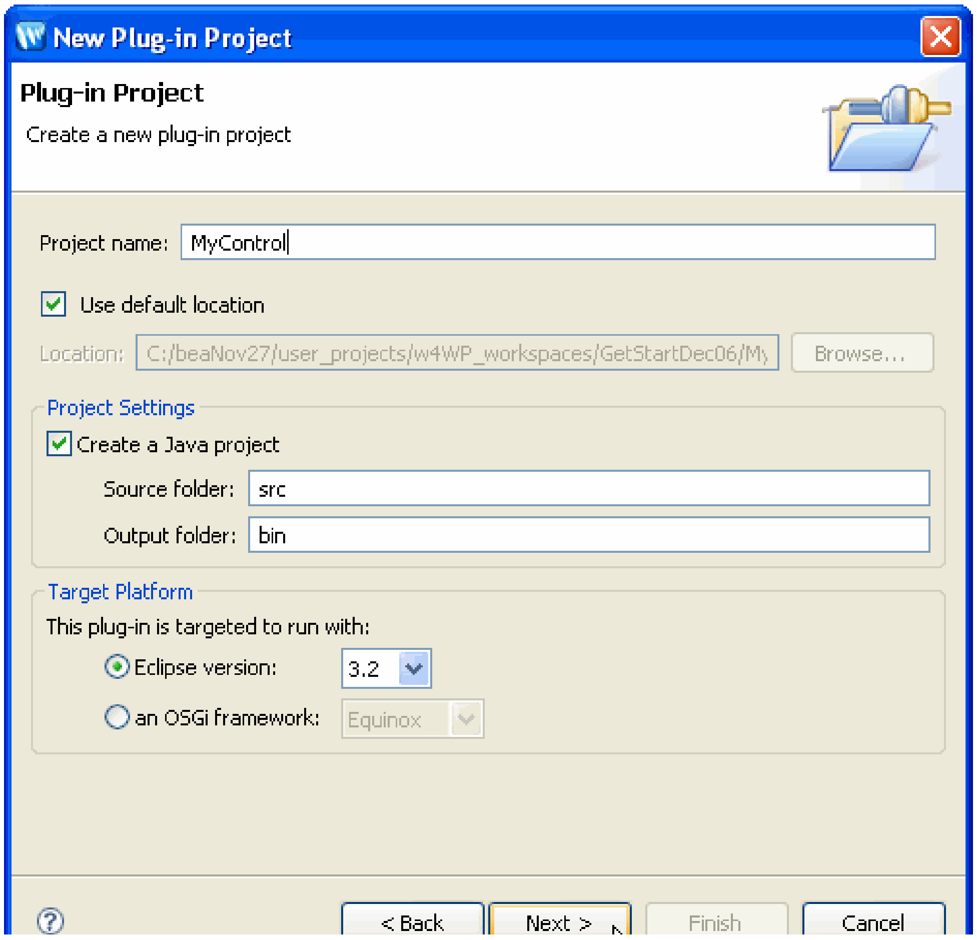
Create a Plug-in Project

Wizards: type filter text

- Java
- Plug-in Development
  - Feature Patch
  - Feature Project
  - Fragment Project
  - Plug-in from existing JAR archives
  - Plug-in Project
  - Update Site Project
- Web
- WebLogic Portal
- Web Services
- Examples

Show All Wizards.

Click **Next** to proceed.



New Plug-in Project

Plug-in Project

Create a new plug-in project

Project name: MyControl

[✓] Use default location

Location: C:/beaNov27/user_projects/w4WP_workspaces/GetStartDec06/My

Project Settings

[✓] Create a Java project

Source folder: src

Output folder: bin

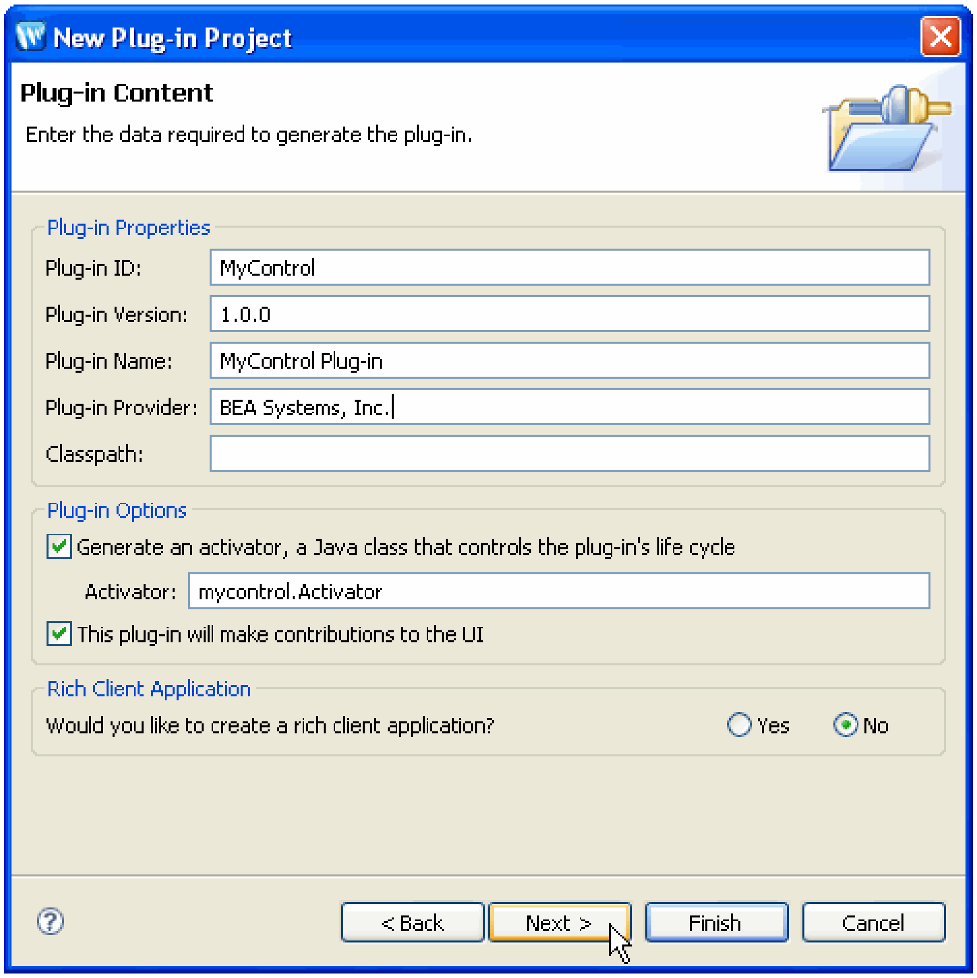
Target Platform

This plug-in is targeted to run with:

[● Eclipse version: 3.2]

an OSGi framework: Equinox

From the next screen, fill in the **Plug-in Provider** field and click **Finish** to create the project.



Plug-in Content

Enter the data required to generate the plug-in.

Plug-in Properties

Plug-in ID:	MyControl
Plug-in Version:	1.0.0
Plug-in Name:	MyControl Plug-in
Plug-in Provider:	BEA Systems, Inc.
Classpath:	

Plug-in Options

✓ Generate an activator, a Java class that controls the plug-in's life cycle

Activator: mycontrol.Activator

✓ This plug-in will make contributions to the UI

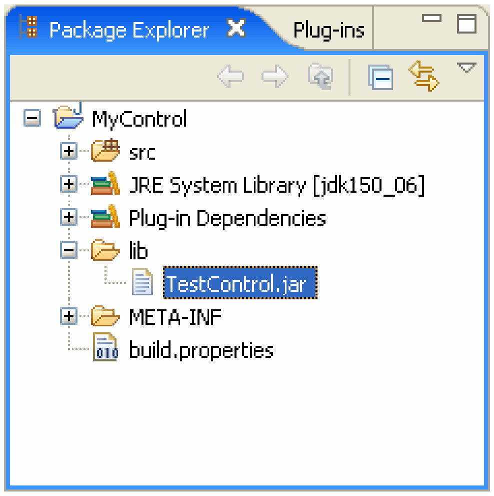
Rich Client Application

Would you like to create a rich client application? Yes [✓ No]

Click **Yes** to change to Plug-in Development perspective.

Step 3: Copy the Control JAR into the Plug-In Project

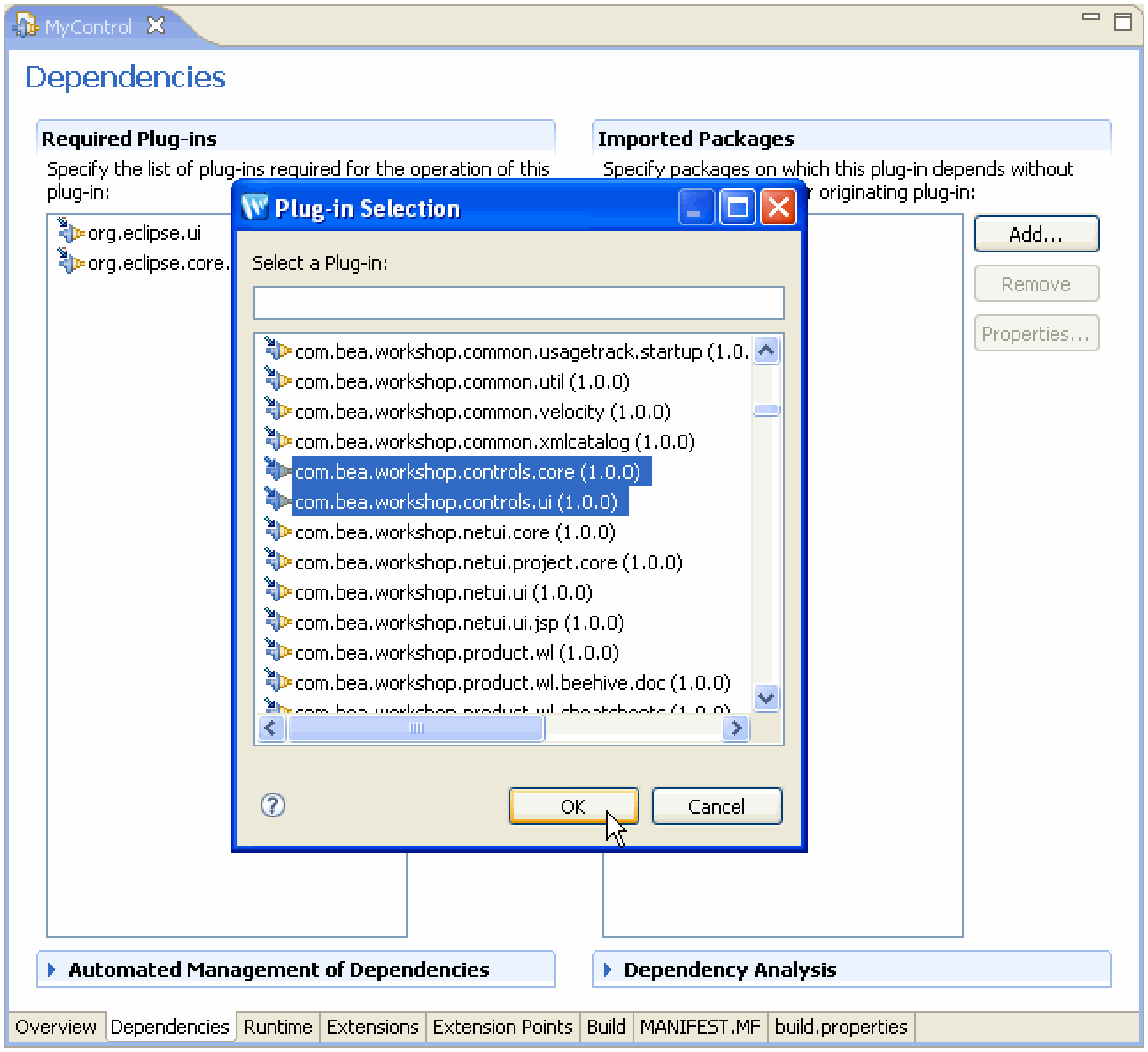
Create a folder named **lib** in the root of your plug-in project (**not** under the **src** folder). Copy the control JAR (created in the previous step) into the **lib** directory.



Step 4: Set Plug-in Project Dependencies

If the manifest editor window is not visible, double click on the MANIFEST.MF file to open it. From the editor, click on the **Dependencies** tab or click on the **Dependencies** link in the **Plug-in Content** section. From the **Required Plug-ins** section, click on the **Add** button and select the following plug-ins:

- com.bea.workshop.controls.core
- com.bea.workshop.controls.ui
- org.eclipse.core.runtime
- org.eclipse.core.resources
- org.eclipse.jdt.core
- org.eclipse.ui

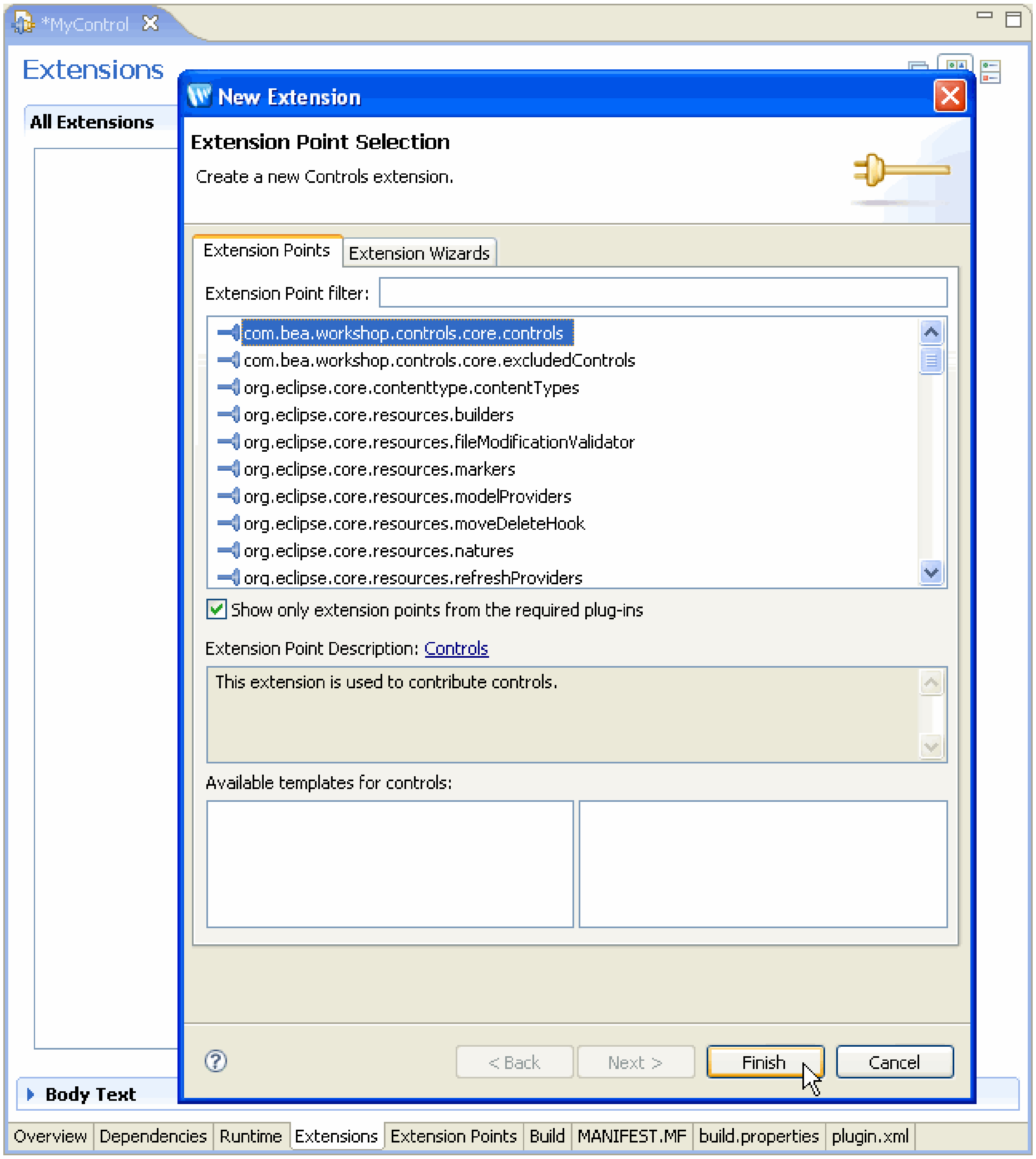

MyControl
Dependencies
Required Plug-ins
Specify the list of plug-ins required for the operation of this plug-in:
org.eclipse.ui
org.eclipse.core.
Imported Packages
Specify packages on which this plug-in depends without
r originating plug-in:
Add...
Remove
Properties...
Plug-in Selection
Select a Plug-in:
com.bea.workshop.common.usagetrack.startup (1.0.
com.bea.workshop.common.util (1.0.0)
com.bea.workshop.common.velocity (1.0.0)
com.bea.workshop.common.xmlcatalog (1.0.0)
com.bea.workshop.controls.core (1.0.0)
com.bea.workshop.controls.ui (1.0.0)
com.bea.workshop.netui.core (1.0.0)
com.bea.workshop.netui.project.core (1.0.0)
com.bea.workshop.netui.ui (1.0.0)
com.bea.workshop.netui.ui.jsp (1.0.0)
com.bea.workshop.product.wl (1.0.0)
com.bea.workshop.product.wl.beehive.doc (1.0.0)
OK
Cancel
Automated Management of Dependencies
Dependency Analysis
Overview
Dependencies
Runtime
Extensions
Extension Points
Build
MANIFEST.MF
build.properties

Click **OK** to add the dependencies. Click **File > Save All** to save the dependencies in the MANIFEST.MF file.

Step 5: Add Extension and Customize Settings

Click on the **Extensions** tab. Click **Add** and choose

- com.bea.workshop.controls.core.controls


*MyControl
Extensions
All Extensions
New Extension
Extension Point Selection
Create a new Controls extension.
Extension Points
Extension Wizards
Extension Point filter:
com.bea.workshop.controls.core.controls
com.bea.workshop.controls.core.excludedControls
org.eclipse.core.contenttype.contentTypes
org.eclipse.core.resources.builders
org.eclipse.core.resources.fileModificationValidator
org.eclipse.core.resources.markers
org.eclipse.core.resources.modelProviders
org.eclipse.core.resources.moveDeleteHook
org.eclipse.core.resources.natures
org.eclipse.core.resources.refreshProviders
Show only extension points from the required plug-ins
Extension Point Description: Controls
This extension is used to contribute controls.
Available templates for controls:
< Back
Next >
Finish
Cancel
Body Text
Overview
Dependencies
Runtime
Extensions
Extension Points
Build
MANIFEST.MF
build.properties
plugin.xml

Click **Finish** to add the extension. Click **File > Save All** to save the change.

Note that a new file: **plugin.xml** has been added to the project. That file now contains the extension information:

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
   <extension
         point="com.bea.workshop.controls.core.controls">
   </extension>
</plugin>
```

The com.bea.workshop.controls.core.controls extension point requires a nested <control> tag with at least the **id, class, isControlExtension,** and **isExtensible** attributes specified. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin>
  <extension point="com.bea.workshop.controls.Controls">
```

```
    <control
        id="com.mycompany.example.MyExampleControlId"
        class="com.mycompany.example.control.MyExampleControl"
        isControlExtension="false"
        isExtensible="false"  />
  </extension>
</plugin>
```

The <control> tag has the following attributes:

Attribute	Description	Required	Default
id	A unique id string. Cannot be duplcated within the contributed controls in this plug-in.	Yes	[none]
class	Fully qualified classname of the control *interface* class.	Yes	[none]
isControlExtension		Yes	Indicates whether this is an extension of a Beehive extensible control. (See the Beehive control documentation for more information on extensible controls.)
isExtensible		Yes	Indicates whether this is an extensible control. Indicating true will allow the default insertion to better handle requiring the user to create a control extension rather than a regular control. (See the Beehive control documentation for more information on extensible controls.)
label	The text to be displayed on the **Insert > Control** dialog.	No	Simple, unqualified classname from the class attribute.
icon	The icon displayed to the left of the control label on the **Insert > Control** dialog.	No	generic icon
priority	Position relative to others in the same group of controls, ascending order. This is a path relative to the plugin root	No	10
groupName	Group heading for the control(s). Note that if there are less than 3 controls, no group will be created. If there are 3 or more controls, a group will be created if groupName is specified.	No	Value of the label attribute. Note that if controls are not in a group, or if there are not 3 controls in a group, they will all be listed at the top level and the label attribute will be ignored.
groupPriority	Ordering of the group relative to other groups, ascending order.	No	100
insertionDelegateClass	Class triggered when the control is inserted into an application. In addition to any desired actions, the insertion delegate must to copy the control JAR from the plug-in JAR to the user's project.	No	com.bea.workshop.controls.core.DefaultControlInsertionDelegate
description	Description of control.	No	[none]

The following is an example of a <control> tag using more attributes:

```
<extension point="com.bea.workshop.controls.core.controls">
      <control

            class="com.mycompany.controls.MyControl"
            id="MyControl12"
            groupName="My Company"
            groupPriority="10"
            includeInPalette="true"
            insertionDelegateClass="com.mycompany.workshop.MyInsertionDelegate"
            isControlExtension="false"
            isExtensible="false"
            label="Sample Control"
            palettePriority="10"
            priority="10"
      />
</extension>
```

Step 6: Create the Insertion Delegate Code

The **insertionDelegateClass** attribute of the <control> tag indicates the insertion delegate and triggers the delegate when the control is inserted into a file. You can use this for many purposes, but if it's not already in the project (e.g., as a facet or a library module), you would typically use this to copy the control JAR to the user's project, as described below.

When you ship the control in a plug-in, the JAR file is located in the plug-in, NOT in the control user's project. To copy the JAR from the plug-in to the project that is using the control, you must insert code similar to the following into your insertion delegate. This will copy the control JAR to the user's project when your insertion delegate is called.

To create an insertion delegate, create a package in the **src** folder and create a file for the class of the insertion delegate.

Sample insertion delegate code is listed below. You will need to update the package and class name, of course.

```
package org.example.controls.workshop;

import java.io.File;


import org.eclipse.core.runtime.IStatus;
import org.eclipse.core.runtime.Status;
import org.eclipse.jdt.core.IJavaElement;
import org.eclipse.jface.dialogs.ErrorDialog;
import org.eclipse.swt.widgets.Display;

import com.bea.workshop.controls.core.model.IControlInsertionDelegateContext;
import com.bea.workshop.controls.ui.actions.DefaultControlInsertionDelegate;

public class SampleInsertionDelegate extends DefaultControlInsertionDelegate {

        @Override
        public IJavaElement insertControl(IControlInsertionDelegateContext ctxt) {
                try {
                        File file = getFileFromPlugin(Activator.getDefault(), "/lib/TestControl.jar");
                        copyJarIfNecessary(ctxt, file, file.getName());
                        return super.insertControl(ctxt);
                } catch (Exception e) {
                        String message = "Error inserting control";
                        error(message,e);
                }
                return null;
        }

        private void error(String message, Exception e) {
                ErrorDialog.openError(Display.getCurrent().getActiveShell(),
                                                                "Control Insert Error",
                                                                message + " - " + e.getMessage(),
                                new Status(IStatus.ERROR,Activator.PLUGIN_ID,1,"Control Insert Error",e));
        }

}
```

The sample above covers the "insert your control into the project" case. To also do a custom wizard that collects parameters and inserts additional annotations, you can update the code to look like this:

```
public IJavaElement insertControl(IControlInsertionDelegateContext ctxt) {
  try {
        //Launch and complete your wizard here, collecting parameters as necessary
        // then proceed to the next steps to copy the file and use the parameters entered
        // as annotation values

        File file = getFileFromPlugin(Activator.getDefault(), "/lib/TestControl.jar");
        copyJarIfNecessary(ctxt, file, file.getName());

        HashMap<String, String> attrs = new HashMap<String, String>();
        attrs.put("attr", "aValue");

        return super.insertControl(ctxt,"org.example.controls.SampleControl.SamplePropertySet",attrs);
  } catch (Exception e) {
        String message = "Error inserting control";
        error(message,e);
  }
  return null;
}
```

The previous example is a convenience API if you have a single additional annotation to add. If you have multiple annotations to add, you could do something like this with a list of AnnotationInfo objects:

```
public IJavaElement insertControl(IControlInsertionDelegateContext ctxt) {
  try {
        //Launch and complete your wizard here, collecting parameters as necessary
        // then proceed to the next steps to copy the file and use the parameters entered
        // as annotation values

        File file = getFileFromPlugin(Activator.getDefault(), "/lib/TestControl.jar");
        copyJarIfNecessary(ctxt, file, file.getName());

        List infos = new ArrayList();
        HashMap<String, String> attrs1 = new HashMap<String, String>();
        attrs1.put("attr", "aValue");
        AnnotationInfo info1 = new AnnotationInfo(
                        "org.example.controls.SampleControl.SamplePropertySet",attrs1);
        infos.add(info1);

        HashMap<String, String> attrs2 = new HashMap<String, String>();
        attrs2.put("anotherAttribute", "aValue");
        AnnotationInfo info2 = new AnnotationInfo(
                        "org.example.controls.SampleControl.SamplePropertySet",attrs2);
        infos.add(info2);

        return super.insertControl(ctxt,infos);
  } catch (Exception e) {
        String message = "Error inserting control";
        Activator.getDefault().logError(message, e);
        error(message,e);
  }
  return null;
}
```

For more information on the APIs provided by the DefaultControlInsertionDelegate, see the Javadoc.

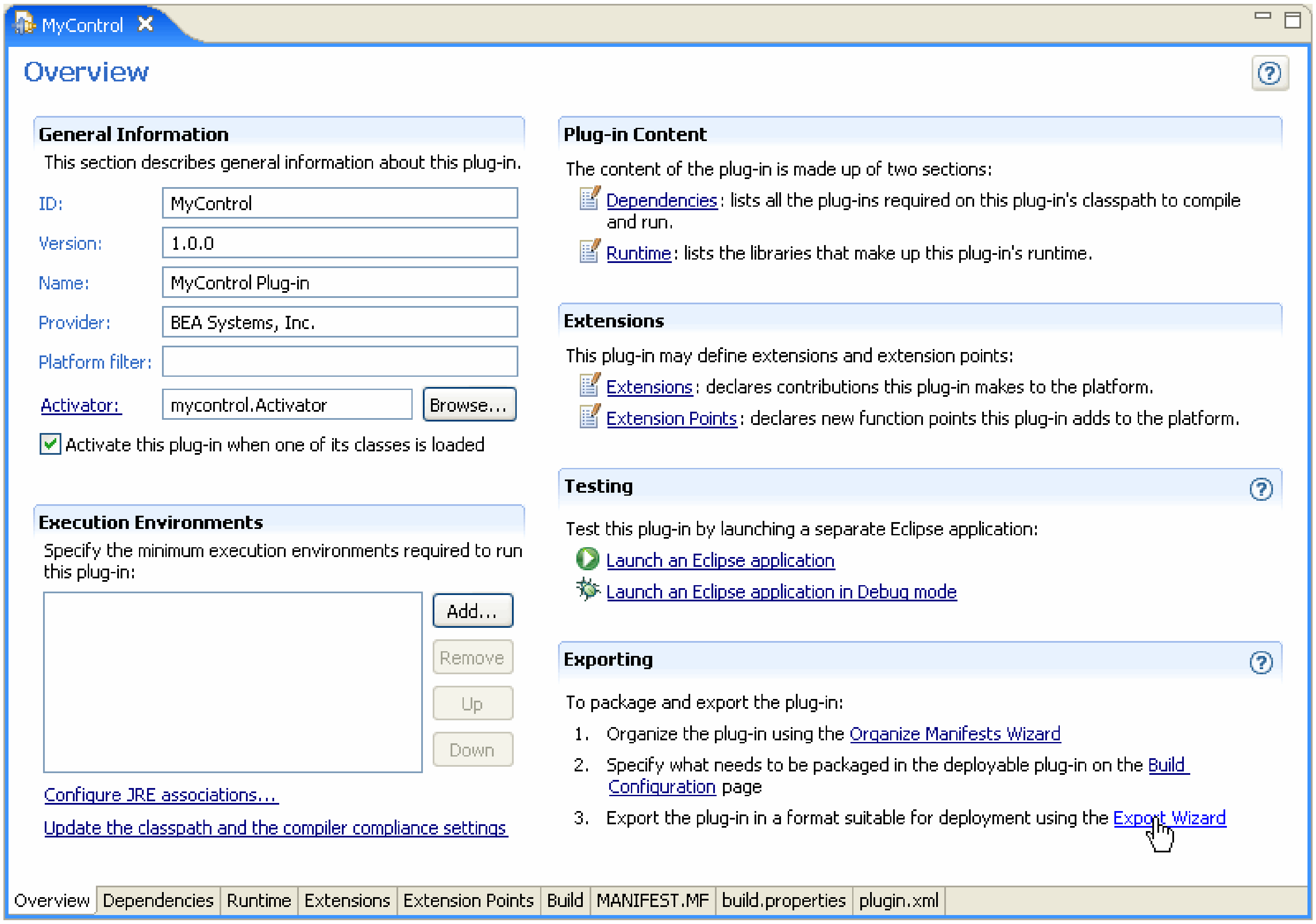
Step 7: Build and Test your Plug-in

Be sure that the plug-in includes the **lib** directly by clicking on the **Build** tab. Click on the **lib** folder under the **Binary Build** section to make sure that it is building correctly.

To run your plug-in, use the **Run As > Eclipse Application** command to test that your plug-in works correctly.

Step 8: Export the Plug-in

Click on the **Overview** tab. Click **Export Wizard** to create the plug-in JAR which will include the control JAR, the insertion delegate and any other required files. If this view is not available, you can open it by right clicking on META-INF/MANIFEST.MF and choosing **Open**.



MyControl

Overview

General Information

This section describes general information about this plug-in.

ID: MyControl
Version: 1.0.0
Name: MyControl Plug-in
Provider: BEA Systems, Inc.
Platform filter:
Activator: mycontrol.Activator [Browse...]

[✓] Activate this plug-in when one of its classes is loaded

Execution Environments

Specify the minimum execution environments required to run this plug-in:

[Add...] [Remove] [Up] [Down]

Configure JRE associations...

Update the classpath and the compiler compliance settings

Plug-in Content

The content of the plug-in is made up of two sections:

- Dependencies: lists all the plug-ins required on this plug-in's classpath to compile and run.
- Runtime: lists the libraries that make up this plug-in's runtime.

Extensions

This plug-in may define extensions and extension points:

- Extensions: declares contributions this plug-in makes to the platform.
- Extension Points: declares new function points this plug-in adds to the platform.

Testing

Test this plug-in by launching a separate Eclipse application:

- Launch an Eclipse application
- Launch an Eclipse application in Debug mode

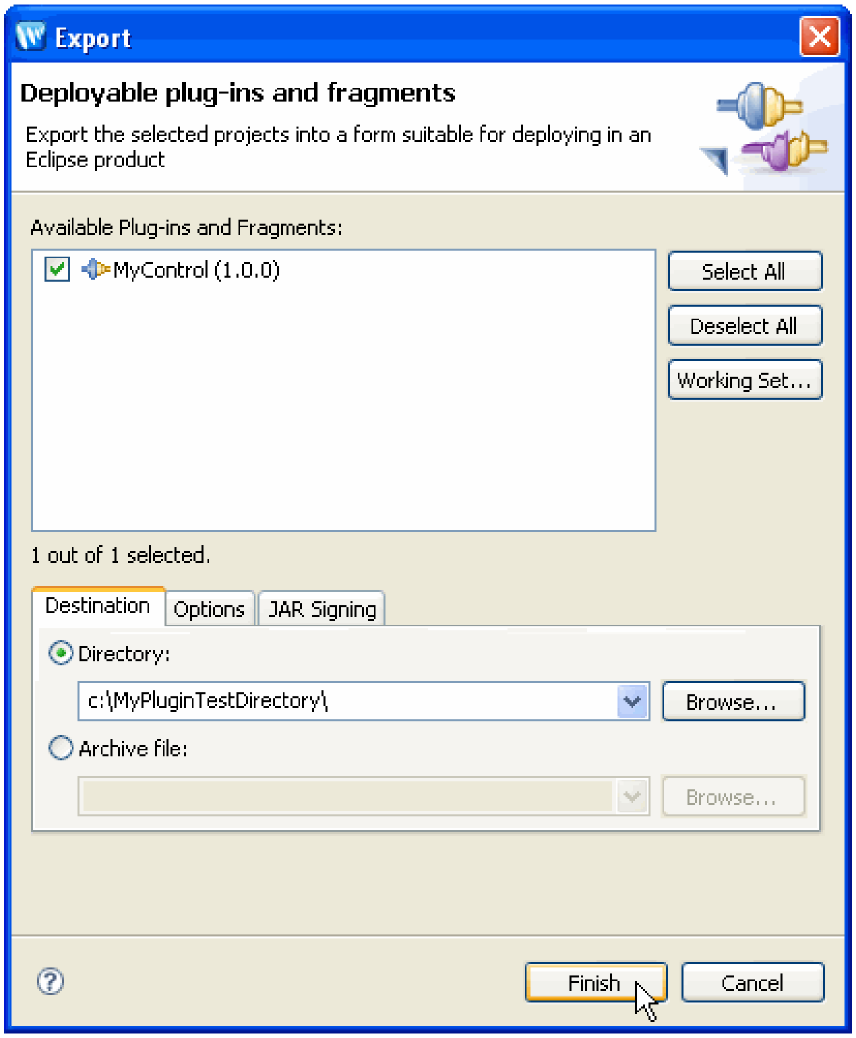
Exporting

To package and export the plug-in:

1. Organize the plug-in using the Organize Manifests Wizard
2. Specify what needs to be packaged in the deployable plug-in on the Build Configuration page
3. Export the plug-in in a format suitable for deployment using the Export Wizard

Overview | Dependencies | Runtime | Extensions | Extension Points | Build | MANIFEST.MF | build.properties | plugin.xml

From the export dialog, be sure to set the directory where the plug-in file will be created. Note that by default, the **Include source code** option is disabled so that the control's source code will not be available to plug-in users. Specify a destination directory for the plug-in and click **Finish** to create the control plug-in file.



Deployable plug-ins and fragments

Export the selected projects into a form suitable for deploying in an Eclipse product

Available Plug-ins and Fragments:

[✓] MyControl (1.0.0)

1 out of 1 selected.

Destination | Options | JAR Signing

[✓ Directory:] c:\MyPluginTestDirectory\

Archive file:

You can then distribute the resulting plug-in file to other developers, like any other Eclipse plug-in.

Related Topics

Developing Custom Controls

Exporting Controls into JARs