

Using the Plug-in Development Environment

IBM

Corporation and others 2000, 2005. This page is made available under license. For full details see the LEGAL in the documentation bundle.

Table of Contents

<u>Introduction to PDE</u>	1
<u>Preparing the workbench</u>	2
<u>Error Log</u>	2
<u>Concepts</u>	3
<u>Host vs. runtime</u>	3
<u>Target Platform</u>	3
<u>External vs. workspace plug-ins</u>	4
<u>Creating a plug-in project</u>	5
<u>Plug-in manifest editor</u>	10
<u>Overview page</u>	10
<u>Dependencies page</u>	11
<u>Runtime page</u>	12
<u>Extensions page</u>	18
<u>Extension points page</u>	20
<u>Extension point schema editor</u>	20
<u>Example: Creating schema for the "Sample Parsers" extension point</u>	21
<u>Sample Parsers</u>	27
<u>Identifier</u>	27
<u>Since</u>	27
<u>Build configuration page</u>	28
<u>Source Locations</u>	30
<u>Identifier</u>	30
<u>Since</u>	30
<u>Source pages</u>	31
<u>Extension point schema</u>	35
<u>The benefits of extension point schemas</u>	35
<u>Limitations of PDE XML Schema support</u>	35
<u>Running a plug-in</u>	37
<u>Example: Running the Sample</u>	38
<u>Choosing plug-ins to run</u>	38
<u>Running with tracing</u>	39
<u>Example: Adding tracing support to your plug-in</u>	40
<u>Exporting a plug-in</u>	42
<u>Shipping Your Plug-in As A Single JAR</u>	43
<u>Generating Ant scripts</u>	44
<u>Fragments</u>	47
<u>Example: Writing a German fragment for XYZ Plug-in</u>	47
<u>Features</u>	50
<u>Setting up a feature project</u>	50

Table of Contents

Fragments

<u>Example: Setting up a feature for plug-ins and fragments</u>	50
<u>Feature manifest editor</u>	51
<u>Synchronizing versions</u>	56
<u>Automatic synchronization at build time (Recommended)</u>	56
<u>UI driven synchronization</u>	56
.....	57
<u>Exporting a Feature</u>	57
<u>Build configuration</u>	59
<u>Generating Ant scripts from the command line</u>	61
<u>Directory file format</u>	63
<u>Using the targets</u>	64
<u>Working with update sites</u>	64
<u>Setting up an update site project</u>	64
<u>Example: Setting up an update site project</u>	64
<u>Building plug-ins, fragments and features using update site editor</u>	65
<u>Previewing an update site</u>	67
<u>Example: building and previewing the sample update site</u>	67
<u>Create an RCP Template</u>	68
<u>Product Configuration</u>	73
<u>Product Editor</u>	74
<u>Feature-Based Product</u>	77
<u>Exporting a Product</u>	78
<u>Using extension point schema</u>	79
<u>Example: Using the "Sample Parsers" extension point</u>	79

Converting existing projects into PDE projects.....81

PDE Extension Points.....82

Extension Wizards.....83

<u>Identifier</u>	83
<u>Description</u>	83

Plug-in Content Wizards.....87

<u>Identifier</u>	87
<u>Description</u>	87

Extension Templates.....90

<u>Identifier</u>	90
<u>Since</u>	90

Other Reference Information.....93

Map of PDE Plug-ins.....94

<u>PDE Dynamic Classpaths FAQ</u>	94
<u>Tips and Tricks</u>	96
<u>What's New in 3.1</u>	98

Table of Contents

Map of PDE Plug-ins

<u>PDE</u>	98
<u>Notices</u>	108
<u>About This Content</u>	108
<u>License</u>	108

Introduction to PDE

The Plug-in Development Environment (PDE) is a tool that is designed to assist developers in the creation, development, testing, debugging, and deployment of Eclipse plug-ins. The mandate of PDE also encompasses tooling for the development of fragments, features, and update sites.

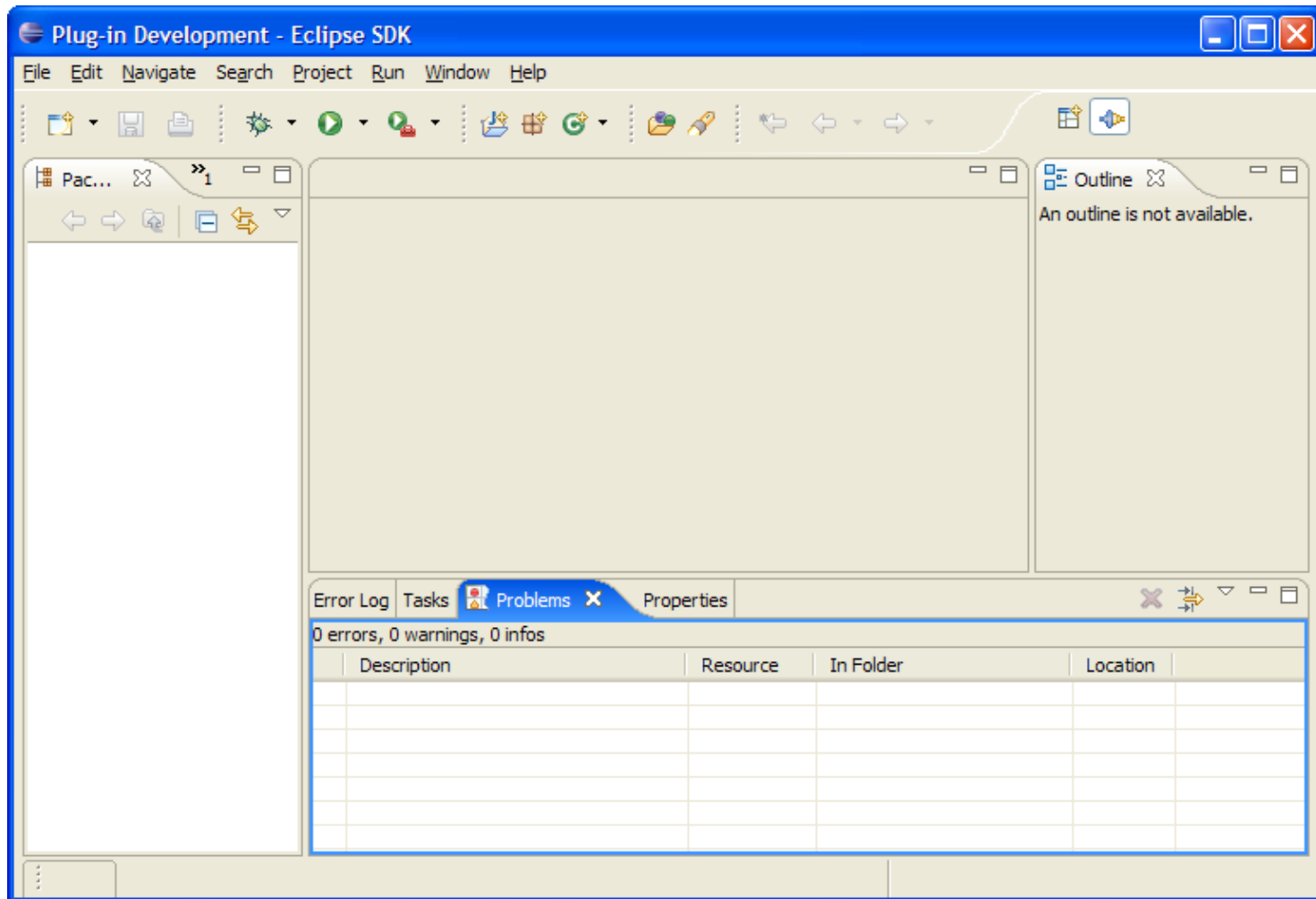
PDE is part of the Eclipse SDK and not a separately launched tool. In line with the general Eclipse platform philosophy, PDE provides a wide variety of platform contributions (e.g. views, editors, wizards, launchers, etc.) that blend transparently with the rest of the Eclipse workbench, and assist the developer in every stage of plug-in development while working inside the Eclipse workbench.

As a pre-requisite to working with PDE, you need to understand the concepts presented in the Platform ISV Guide and the JDT User Guide.

Preparing the workbench

While you can access the PDE platform contributions from any perspective, the PDE perspective is arguably the best.

From the default Resource perspective, open the PDE perspective via **Window > Open Perspective > Other...** and choose **Plug-in Development** from the offered list.



In addition to the main views and toolbar actions that are useful for Java development, the PDE perspective adds shortcuts to very frequently used wizards such as the New Plug-in Project creation wizard, etc. It also adds views that are very important to a plug-in developer including the **Error Log** view.

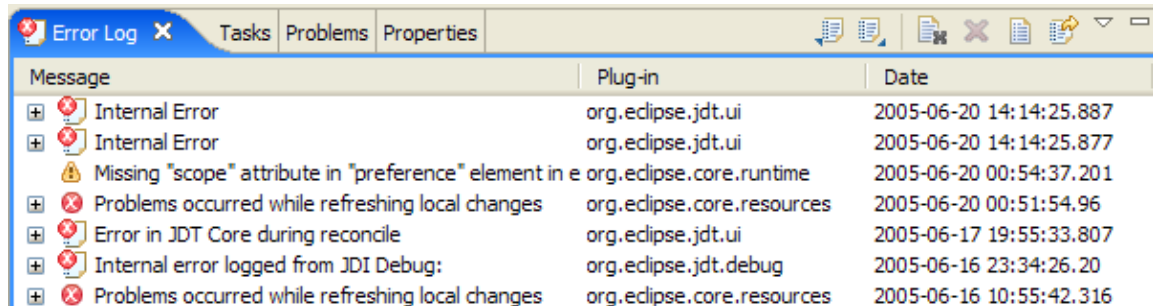
Error Log

The **Error Log** view captures all internal warnings and errors thrown by the platform and by your code. These errors are written to a **.log** file that is located in the **.metadata** subdirectory of your workspace, and the Error Log view shows the content of this file with a variety of convenient options such as filtering, sorting, etc.

Using the Plug-in Development Environment

By default, once the Error Log view is in your workspace, it will be brought to the front upon the logging of new events. This feature can be turned on/off in the drop down menu of the view.

Other features of the log view include the ability to import any arbitrary log file into the view, and to export the log contents into a file.



Message	Plug-in	Date
Internal Error	org.eclipse.jdt.ui	2005-06-20 14:14:25.887
Internal Error	org.eclipse.jdt.ui	2005-06-20 14:14:25.877
Missing "scope" attribute in "preference" element in e	org.eclipse.core.runtime	2005-06-20 00:54:37.201
Problems occurred while refreshing local changes	org.eclipse.core.resources	2005-06-20 00:51:54.96
Error in JDT Core during reconcile	org.eclipse.jdt.ui	2005-06-17 19:55:33.807
Internal error logged from JDI Debug:	org.eclipse.jdt.debug	2005-06-16 23:34:26.20
Problems occurred while refreshing local changes	org.eclipse.core.resources	2005-06-16 10:55:42.316

Concepts

Host vs. runtime

One of the most important concepts in PDE to understand is that of **host** and **runtime** workbench instances.

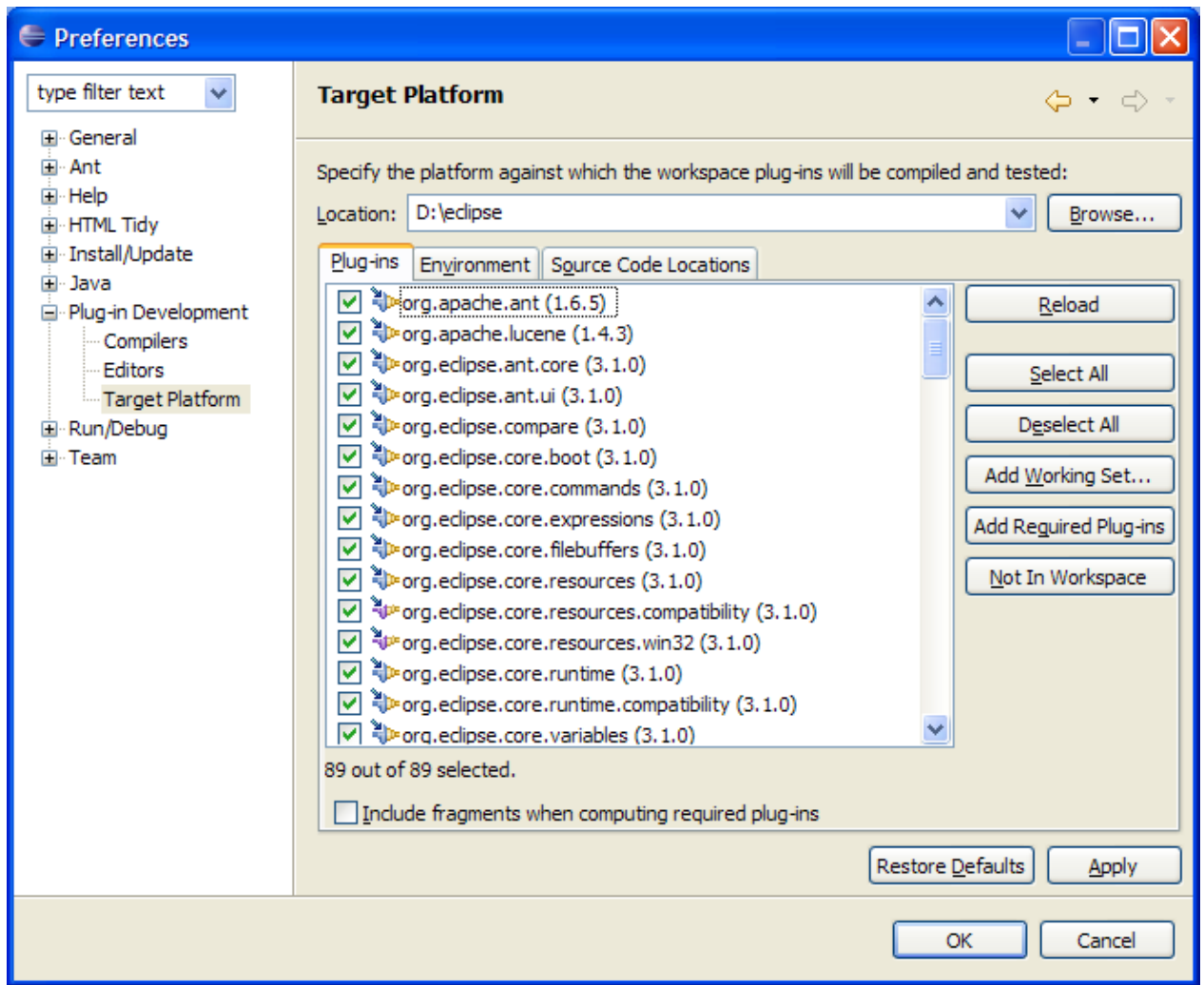
When you start up the workbench, you will use it to work on your projects that define the plug-ins you are building. The workbench instance that you are running as you develop your plug-in using the PDE and other tools is the **host** instance. The features available in this instance will come exclusively from the plug-ins that are installed with your application.

Once you are happy with your plug-in and want to test it, you can launch another workbench instance, the **runtime** instance. This instance will contain the same plug-ins as the **host** instance, but will also have the plug-ins you were working on in the **host** instance. PDE launcher will take care of merging your plug-ins with the host plug-ins and creating the run-time instance.

Target Platform

Target Platform refers to the Eclipse product against which the plug-ins you are developing will be compiled and tested. The Target Platform must therefore be the same platform in which you plan to deploy your plug-ins.

The location of the target platform is set on the **Plug-in Development > Target Platform** preference page. By default, the target platform is the same as the platform you are using for development, but this is not required. You can set the target platform to whatever Eclipse-based product you want. For example, if you want to take advantage of the latest and greatest Eclipse 3.0 features to develop for plug-ins that will be deployed in a product based on a 2.x Eclipse, you can use Eclipse 3.0 as your development platform and a 2.x-based product as your target platform.



All the plug-ins found in the target platform location specified by the user are listed on the preference page. However, only the plug-ins that are explicitly checked constitute the target platform; the rest are ignored by PDE. By default, all plug-ins are checked.

External vs. workspace plug-ins

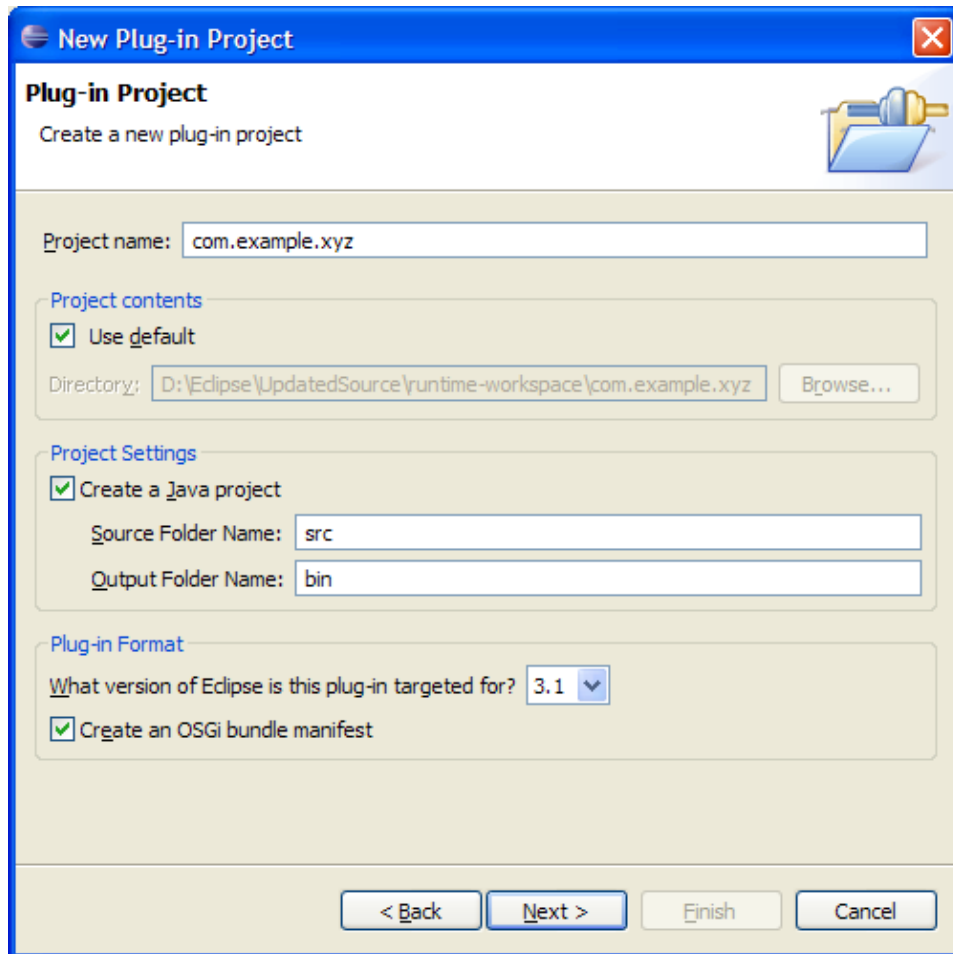
When developing Eclipse plug-ins, the set of plug-ins that you will be used to run the runtime workbench come from two distinct places: the workspace of the host instance and the target platform. Although, to PDE, all plug-ins are the same no matter where they come from, there are a few differences that quickly become evident to users.

- **Workspace plug-ins** are those plug-ins under development in your host workbench. They are under your control and can be added, deleted and modified by the user.
- **External plug-ins** are plug-ins that arrived with the basic platform installation and are simply referenced from their original location without modification. You can reference them, browse them, view their source and debug them, but they are read-only.

Creating a plug-in project

In the workspace, a plug-in is represented by a single project that encapsulates all the code and resources of the plug-in.

To create a plug-in project, bring up the New Plug-in Project creation wizard via **File > New > Plug-in Project**.



It is a convention that plug-in project names are the same as plug-in IDs, but they can be different.

The plug-in project can be created in one of two flavors: a Java project or a simple project. Most plug-ins are meant to contain executable Java code and must therefore be housed in a Java project. On the other hand, if, for example, you are creating a documentation plug-in, then a simple project will suffice.

A plug-in with an OSGi bundle manifest is the recommended plug-in format. In addition to faster startup and classloading, it allows the plug-in to take advantage to many new runtime capabilities.

Click *Next*.

Using the Plug-in Development Environment

The screenshot shows the 'New Plug-in Project' dialog box with the 'Plug-in Content' tab selected. The dialog has a blue title bar and a close button in the top right corner. Below the title bar, the text 'Enter the data required to generate the plug-in.' is displayed next to a small icon of a plug-in. The main area is divided into three sections: 'Plug-in Properties', 'Plug-in Class', and 'Rich Client Application'. The 'Plug-in Properties' section contains five text input fields: 'Plug-in ID' (com.example.xyz), 'Plug-in Version' (1.0.0), 'Plug-in Name' (Xyz Plug-in), 'Plug-in Provider' (EXAMPLE), and 'Classpath' (empty). The 'Plug-in Class' section contains a checked checkbox 'Generate the Java class that controls the plug-in's life cycle' and a text input field 'Class Name' (com.example.xyz.XyzPlugin). Below this is another checked checkbox 'This plug-in will make contributions to the UI'. The 'Rich Client Application' section contains a question 'Would you like to create a rich client application?' with two radio buttons: 'Yes' (unselected) and 'No' (selected). At the bottom of the dialog are four buttons: '< Back', 'Next >', 'Finish', and 'Cancel'.

New Plug-in Project

Plug-in Content

Enter the data required to generate the plug-in.

Plug-in Properties

Plug-in ID: com.example.xyz

Plug-in Version: 1.0.0

Plug-in Name: Xyz Plug-in

Plug-in Provider: EXAMPLE

Classpath:

Plug-in Class

☒ Generate the Java class that controls the plug-in's life cycle

Class Name: com.example.xyz.XyzPlugin

☒ This plug-in will make contributions to the UI

Rich Client Application

Would you like to create a rich client application? ☐ Yes ☒ No

< Back Next > Finish Cancel

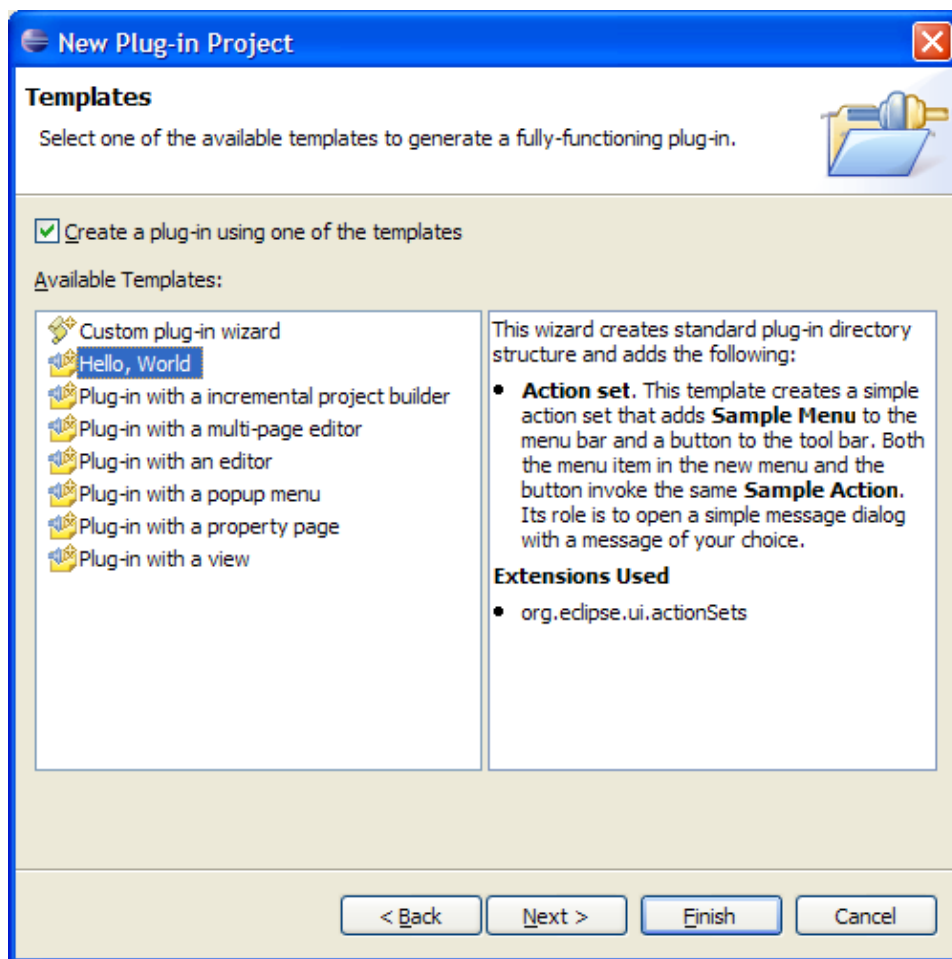
On the Plug-in Content page, you set the data with which the plugin.xml file will be initialized, including the plug-id, version and name.

The recommended deployed form of a plug-in is to be shipped as a single JAR with all the classes and resources at the root of the JAR. For this format, you may keep the *Classpath* field empty, or enter '.' (without quotes).

The Plug-in class is a top-level Java class that represents the entire plug-in. It will be used at runtime to control the plug-in's life cycle, i.e. its implementation will determine what happens when the plug-in starts up or shuts down.

Click *Next*.

Using the Plug-in Development Environment

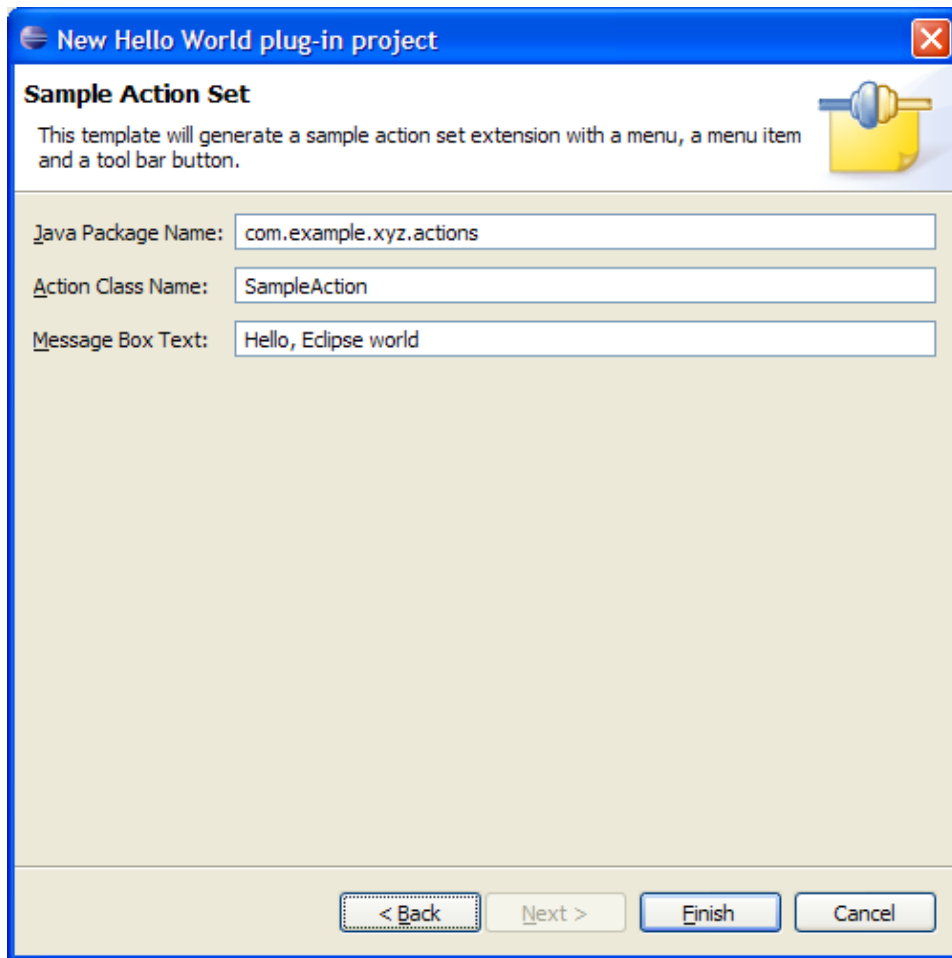


The next page shows the various templates that PDE provides which generate useful content such as views, editors, property pages etc.

In this example, we will create a plug-in with the "Hello, World" template. You can read about the wizard in the area to the right of the wizard list.

Click **Next**.

Using the Plug-in Development Environment



New Hello World plug-in project

Sample Action Set

This template will generate a sample action set extension with a menu, a menu item and a tool bar button.

Java Package Name:

Action Class Name:

Message Box Text:

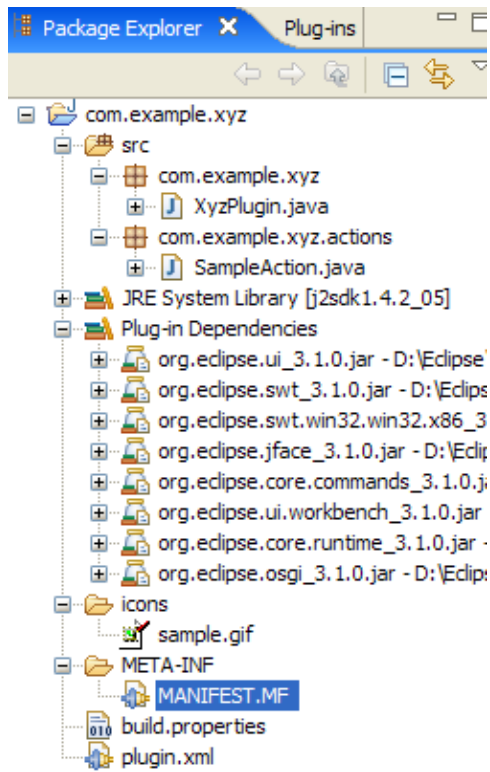
< Back Next > Finish Cancel

The next page will let you customize the sample extension that we are creating.

When you press **Finish**, the wizard will create the new project, all the specified folders and files, and the initial Java build path. The build path is important for correct compilation of Java classes that are generated. The wizard will also open the plug-in manifest editor.

After the wizard is finished, the initial project structure should look like this:

Using the Plug-in Development Environment



Plug-in manifest editor

When the plug-in project is created, the manifest file is open in the plug-in manifest editor.

This multi-page editor is the central place to manage your plug-in and can be used to edit all the plug-in's file (manifest.mf , plugin.xml and build.properties).

When you use the editor's forms, PDE transparently handles the task of writing the changes to the right files.

The best way to learn about the plug-in manifest editor is to visit each page.

Overview page

The Overview page is designed to be a quick reference on how to develop, test and deploy a plug-in. It is also a navigational center where you can follow the hyperlinks to navigate a particular page or execute a particular command.

The screenshot shows the 'Overview' page of the Plug-in Manifest Editor. The window title is 'com.example.xyz'. The page is divided into several sections:

- General Information:** This section describes general information about this plug-in. It contains input fields for ID (com.example.xyz), Version (1.0.0), Name (Xyz Plug-in), Provider (EXAMPLE), Class (com.example.xyz.XyzPlugin with a 'Browse...' button), and Platform filter.
- Plug-in Content:** This section explains the structure and content of the plug-in manifest. It lists four 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 (declares contributions this plug-in makes to the platform), and Extension Points (declares new function points this plug-in adds to the platform).
- Testing:** This section provides shortcuts to quickly launch a runtime workbench to test and debug the plug-in. It includes links for 'Launch an Eclipse application' and 'Launch an Eclipse application in Debug mode'.
- Exporting:** This section lists the steps required to successfully build and package the plug-in. It includes a list of steps: 1. Specify what needs to be packaged in the deployable plug-in on the Build Configuration page, and 2. Export the plug-in in a format suitable for deployment using the Export Wizard.

At the bottom of the page, there is a navigation bar with tabs for Overview, Dependencies, Runtime, Extensions, Extension Points, Build, MANIFEST.MF, plugin.xml, and build.properties. The 'Overview' tab is currently selected.

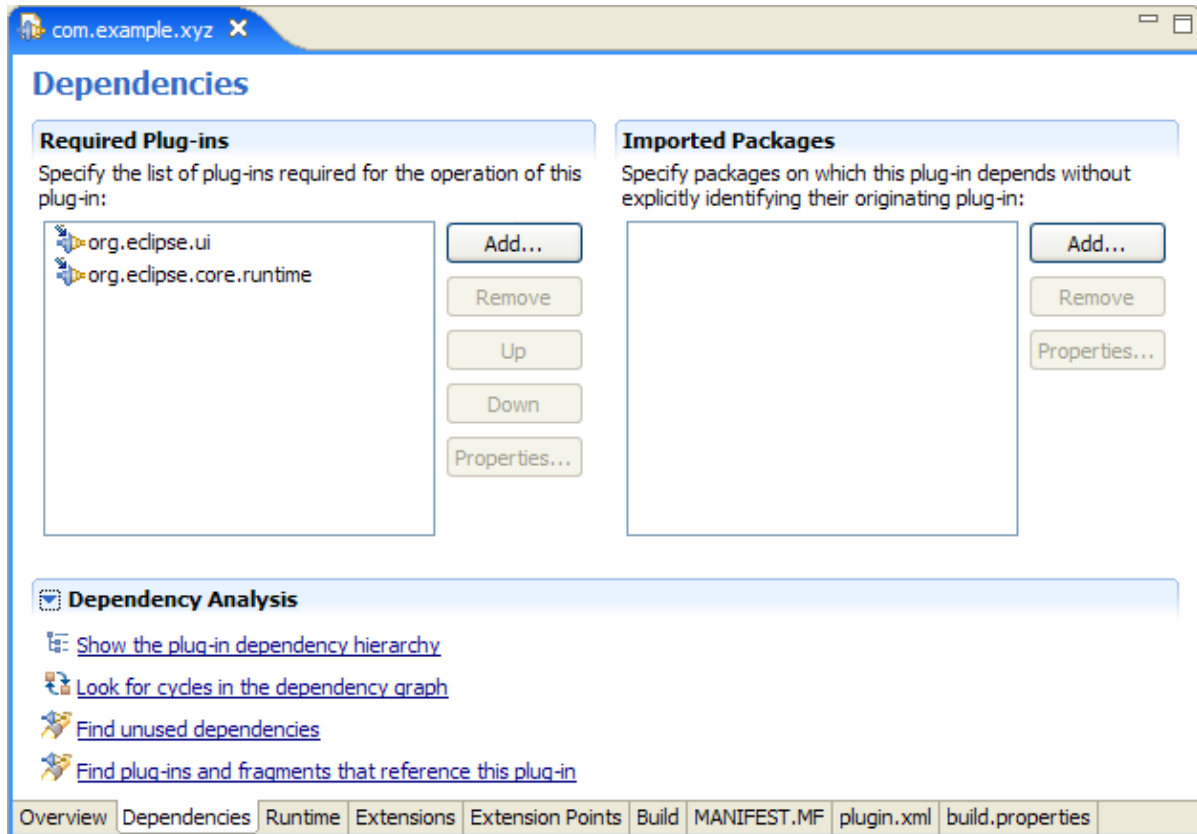
The **Plug-in Content** section explains the structure and content of each section of the plug-in manifest.

The **Testing** section provides shortcuts to quickly launch a runtime workbench to test and debug the plug-in.

The **Exporting** section lists the steps required to successfully build and package the plug-in.

Dependencies page

The **Dependencies** page shows the dependencies that your plug-in has on other plug-ins. You must list on this page all the plug-ins that contribute code required on your plug-in project's classpath to compile. When you modify the list of dependencies and save the file, PDE will automatically update your classpath.



Note that the order of the plug-ins in the list is important because it dictates the classloading order at runtime, so use the **Up** and **Down** buttons to organize the list as appropriate.

A plug-in listed in the **Required Plug-ins** section can be marked as re-exported in its Properties dialog. Re-exporting a dependency means that clients of your plug-in will get that dependency for free. It is important that you do not abuse this functionality and use it only when it makes sense to do so.

If your plug-in requires a specific version of a plug-in to function properly, then you can specify the version required along with the version match rule. You can read more about valid values in the Platform ISV guide.

The Eclipse runtime gives the flexibility to declare a dependency on a package without explicitly identifying its originating plug-in. These packages are listed in the **Imported Packages** section.

The **Dependency Analysis** contain several useful features such as finding cycles in the dependency graph. Such cycles are forbidden by the runtime, making the analysis useful for performing a sanity check on your plug-in's dependency graph before testing it.

Using the Plug-in Development Environment

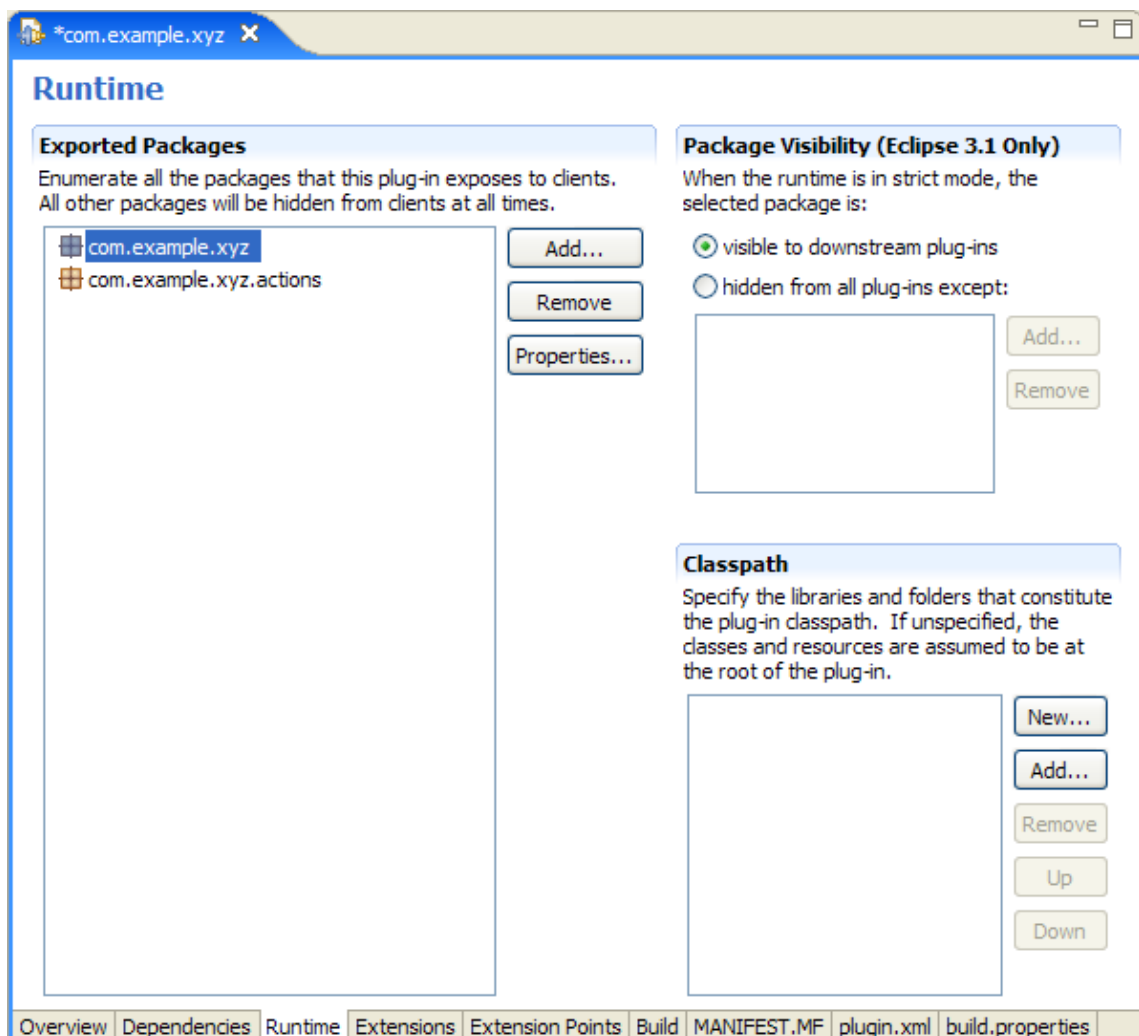
For a selected plug-in in the list, **Compute Dependency Extent** will give you a list of all the Java types and all the extension points that your plug-in needs from that dependency. So, in essence, it tells you why you need that plug-in.

Since the JARs from all the plug-ins in the list of dependencies will be on your plug-in's classpath at runtime, it is very important to not have any dependencies that you do not need, as they would slow down classloading. To find such extraneous entries and remove them, use the **Find Unused Dependencies** feature available on this page.

Runtime page

The **Runtime** page shows all the packages that your plug-in makes visible to other plug-ins.

Press the **Add...** button in the **Exported Packages** section to add the *com.example.xyz* and *com.example.xyz.actions* packages to the list.



The **Package Visibility** section allows you to control on a per-package basis the visibility of your plug-in code to downstream plug-ins. Refer to the [Access Restrictions](#) document for full details.

Using the Plug-in Development Environment

The **Classpath** section is the place to declare the names of the libraries that constitute the plug-in classpath. Since JAR'd plug-ins are the recommended format for 3.1, it is recommended to leave this section blank.

Access Restrictions

The Eclipse 3.1 runtime gives the plug-in developer the *option* to control on a per-package basis the visibility of the plug-in code to downstream plug-ins.

A package may be classified as one of the following:

1. Accessible
2. Forbidden
3. Internal
4. Internal with friends

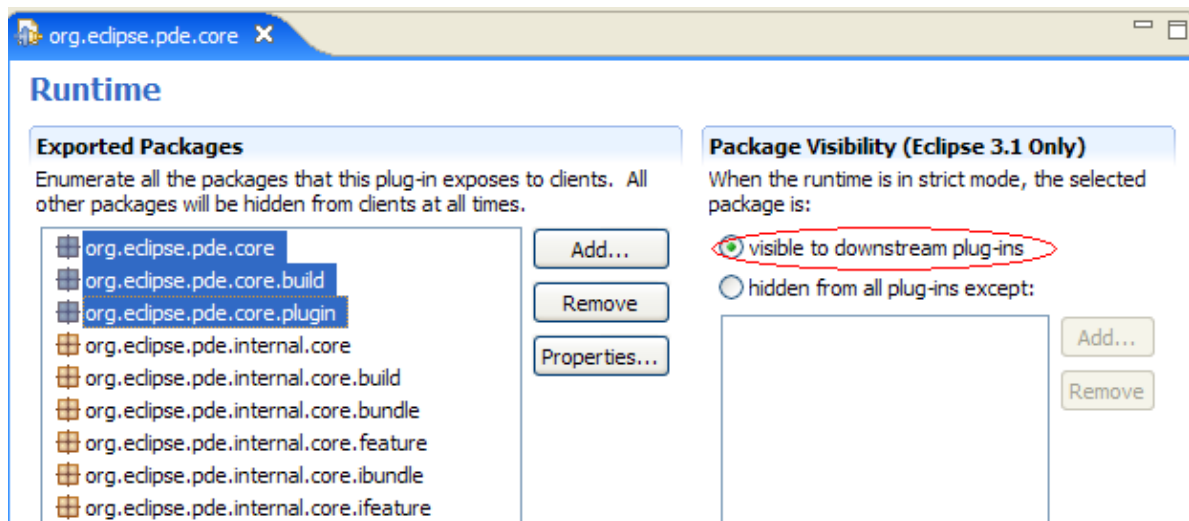
PDE translates these runtime visibility rules into compiler access restriction rules at compile time. As a result, a violation of a visibility rule is flagged by the compiler as a warning or an error, depending on the severity of that violation.

With the availability of this support at compile time, one is never get caught by surprise by classloading errors at runtime, and is always aware when referencing internal types.

Accessible Packages

Accessible packages are visible to downstream plug-ins unconditionally. While API packages must clearly fall in this category, it is completely up to the developer to decide what other packages exported by the plug-in ought to be given this level of visibility.

In order to declare a package as accessible, you must list it in the **Exported Packages** section on the **Runtime** of the plug-in manifest editor and leave the default visibility setting as-is.



Forbidden Packages

You can hide a package from downstream plug-ins at all times by **excluding** it from the list in the **Exported Packages** section on the **Runtime** page of the plug-in manifest editor.

References to types from a forbidden package result in classloading errors at runtime.

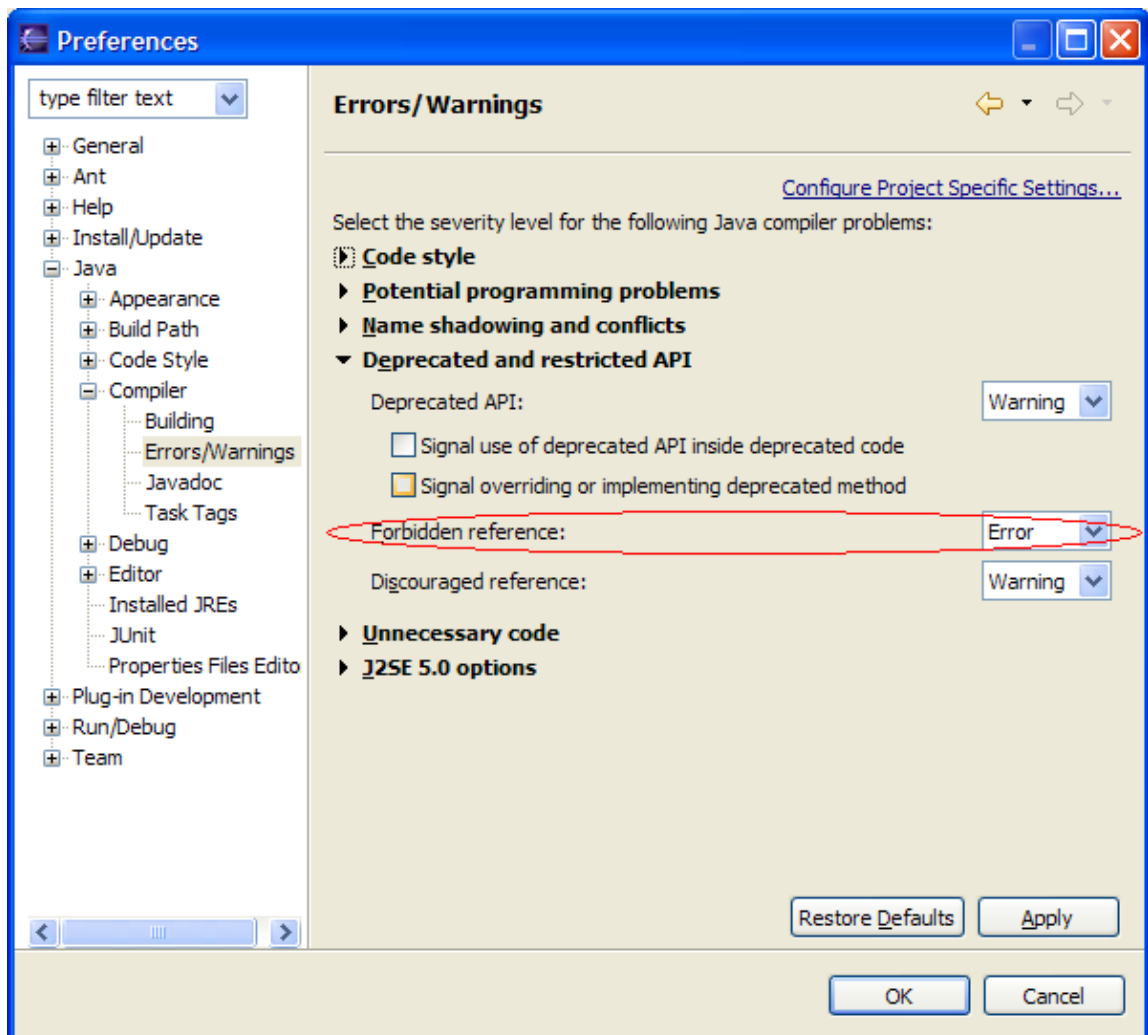
To avoid such unpleasant situations:

1. The compiler will flag references to forbidden packages with an **ERROR**.
2. Types from forbidden packages are **NOT** available as proposals in the content assist.

Notes:

1. All plug-ins in the Eclipse SDK enumerate all their packages in the Exported Packages section. Therefore, none of the packages in the SDK have forbidden access.
2. The severity level for forbidden references is set on the **Java > Compiler > Errors/Warnings > Deprecated and restricted API** preference page.

It is strongly recommended that the severity of a forbidden reference is kept at **ERROR**.

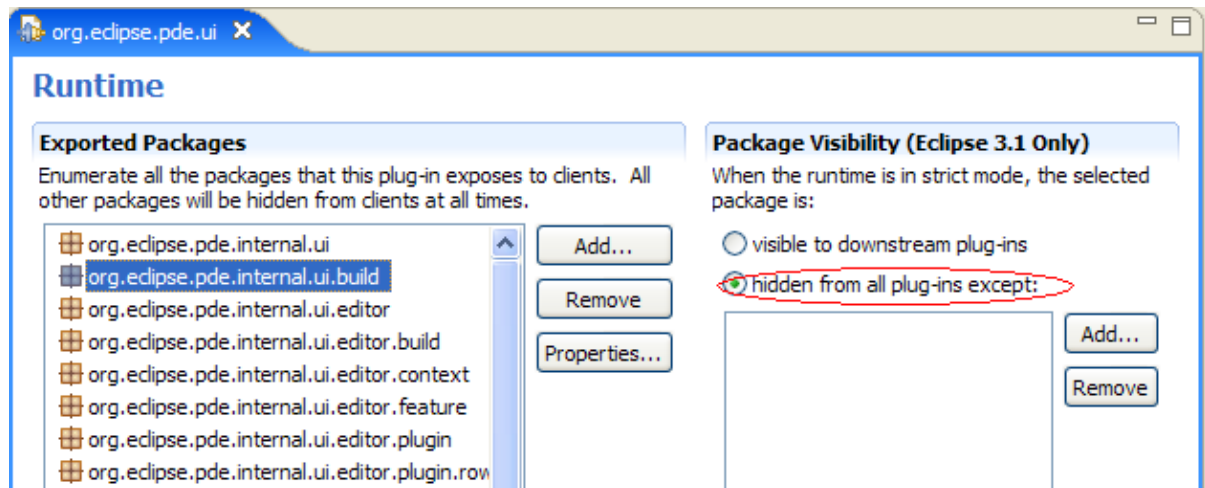


Internal Packages

Internal packages are packages that are not intended for use by downstream plug-ins. *These packages are visible to downstream plug-ins by default.*

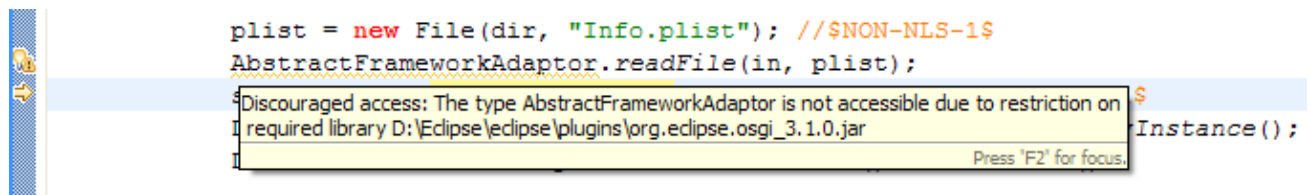
Internal packages are hidden from downstream plug-ins only when Eclipse is launched in *strict* mode (i.e. when you launch with the `-Dosgi.resolverMode=strict` VM argument).

Internal packages must be listed in the **Exported Packages** section on the **Runtime** page of the plug-in manifest editor with the *hidden* option selected.

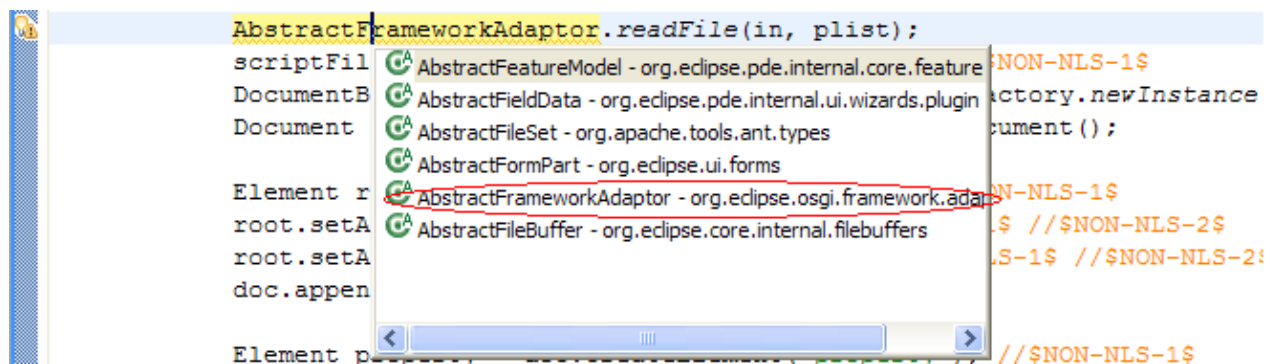


Two measures are taken to discourage downstream plug-ins from referencing internal packages:

- The compiler flags references to such packages with a WARNING.

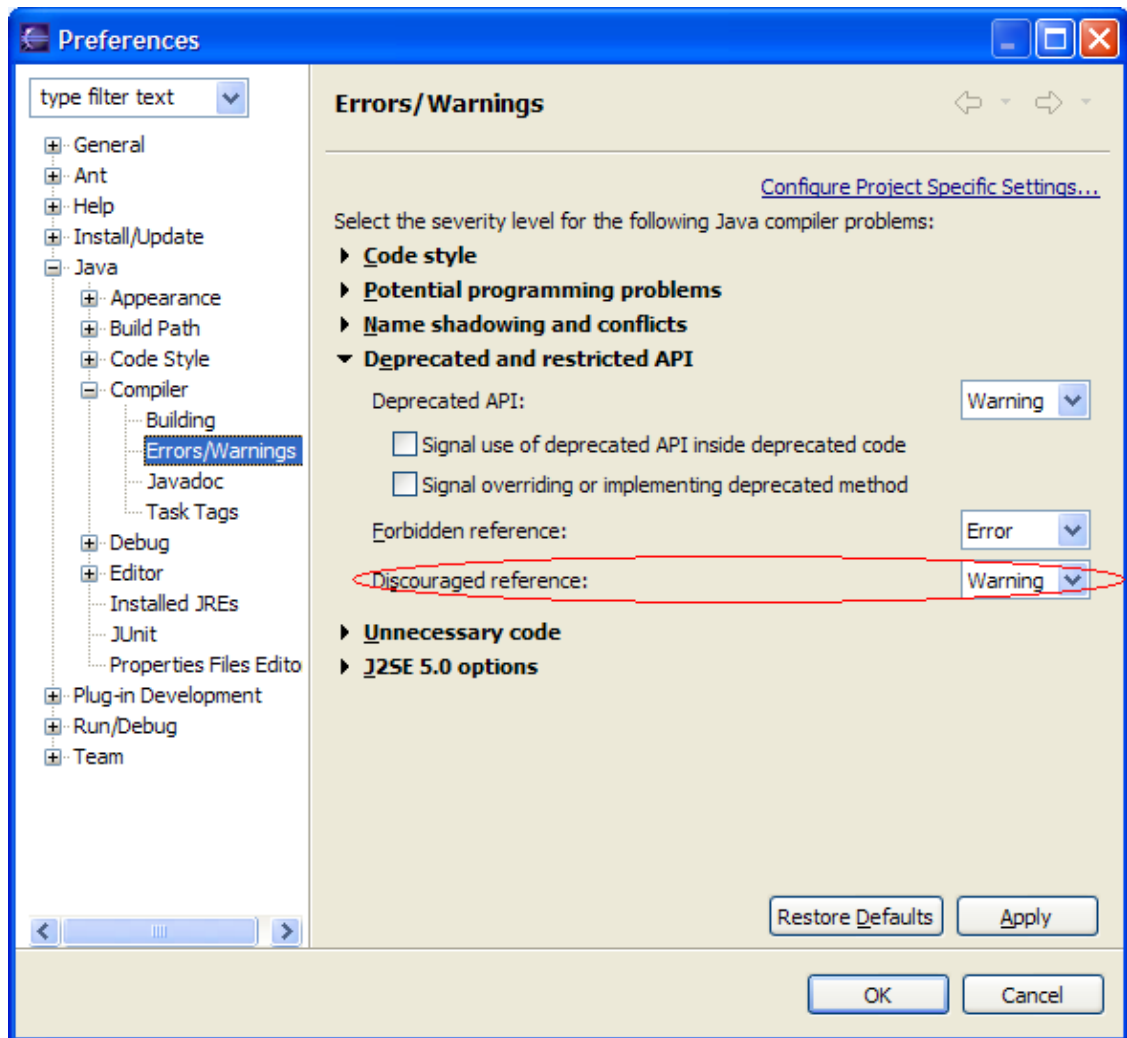


- Types from discouraged packages are available as proposals in the content assist but with a LOWER PRIORITY.



Using the Plug-in Development Environment

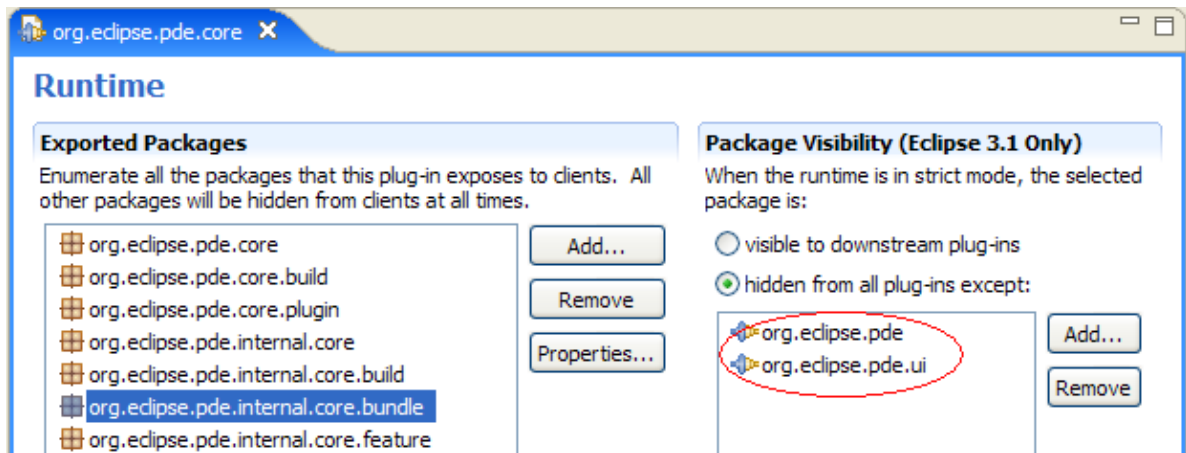
The severity level for discouraged references can be set on the **Java > Compiler > Errors/Warnings > Deprecated and restricted API** preference page.



Internal packages with friends

It is important for a plug-in to be able to grant full access to its internal packages to designated "friend" plug-ins. For instance, the PDE code is split across multiple plug-ins, and the *org.eclipse.pde.ui* plug-in should have full access to *org.eclipse.pde.core*'s internal packages.

In the example below, the friends (the *org.eclipse.pde* and *org.eclipse.pde.ui* plug-ins) have full access to the *org.eclipse.pde.internal.core.bundle* package from the *org.eclipse.pde.core* plug-in.



The friends are free to reference any type from the *org.eclipse.pde.internal.core.bundle* package with the compiler's blessing.

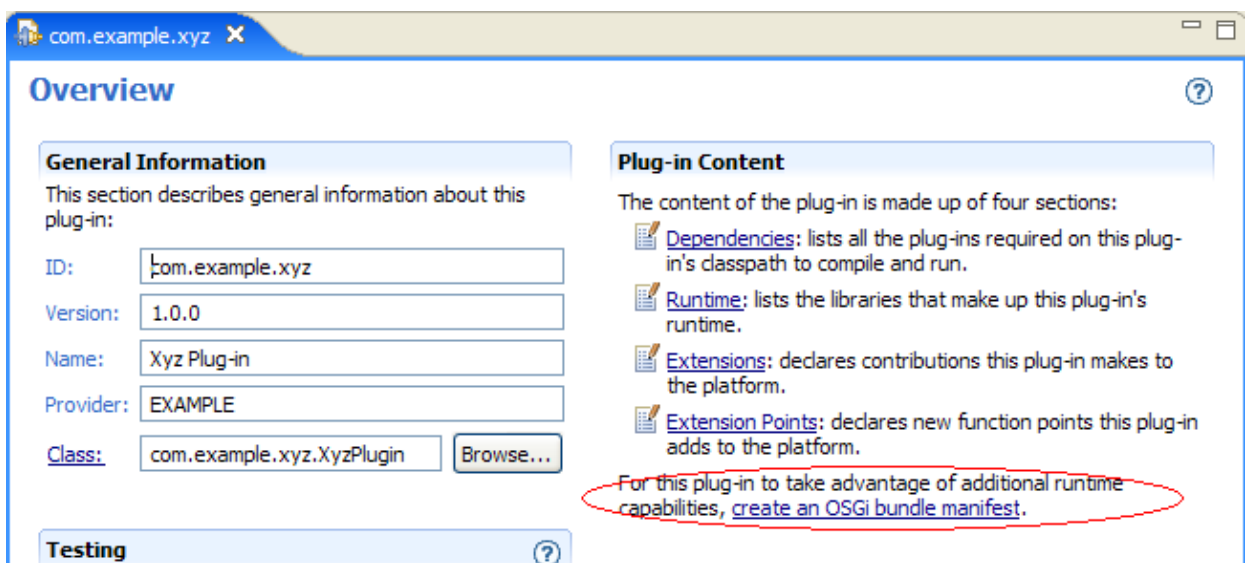
If, on the other hand, any other plug-in references a type from the *org.eclipse.pde.internal.core.bundle* package, the compiler flags the reference as a discouraged reference as described in the previous section.

How to enable access restrictions

To take advantage of the PDE access restriction support, the only requirement is that the plug-ins in question contain an OSGi bundle manifest.mf. PDE, which manages the plug-in classpath, then takes care of the rest.

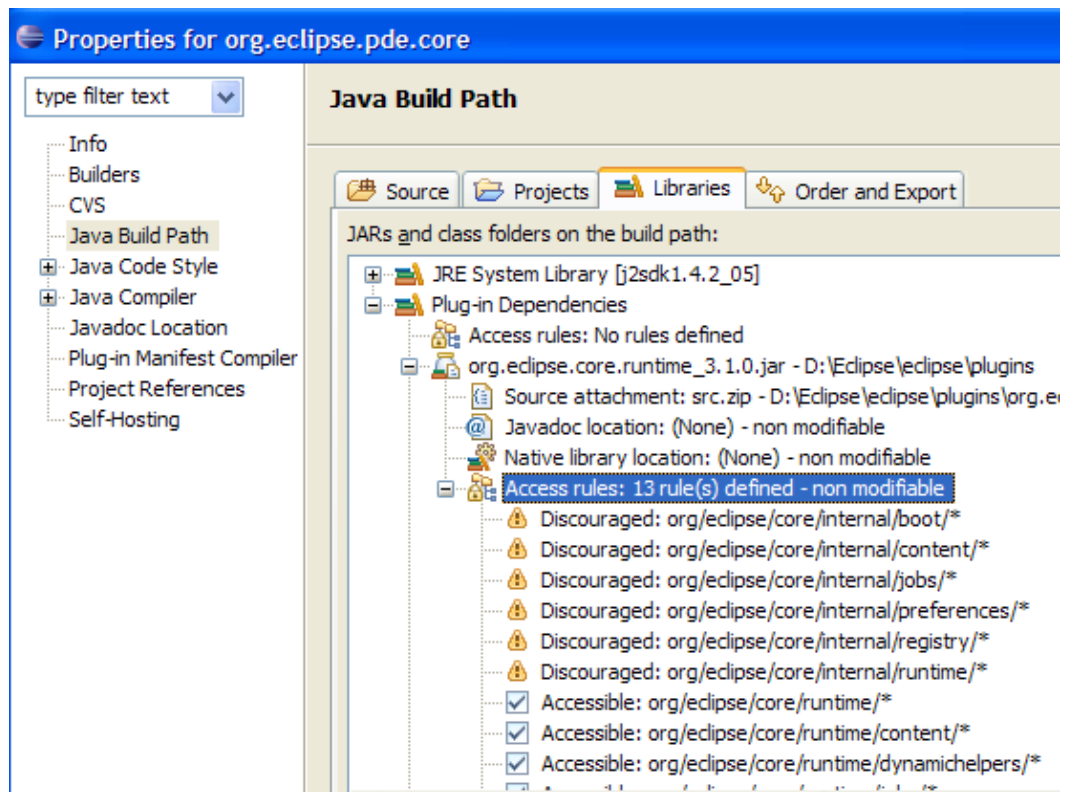
If the plug-in does not contain a manifest.mf file, that file can be created as follows:

1. Open the plugin.xml in the plug-in manifest editor.
2. In the **Plug-in Content** section of the **Overview** page, click on the '*create an OSGi bundle manifest*' link.



Inspecting access rules

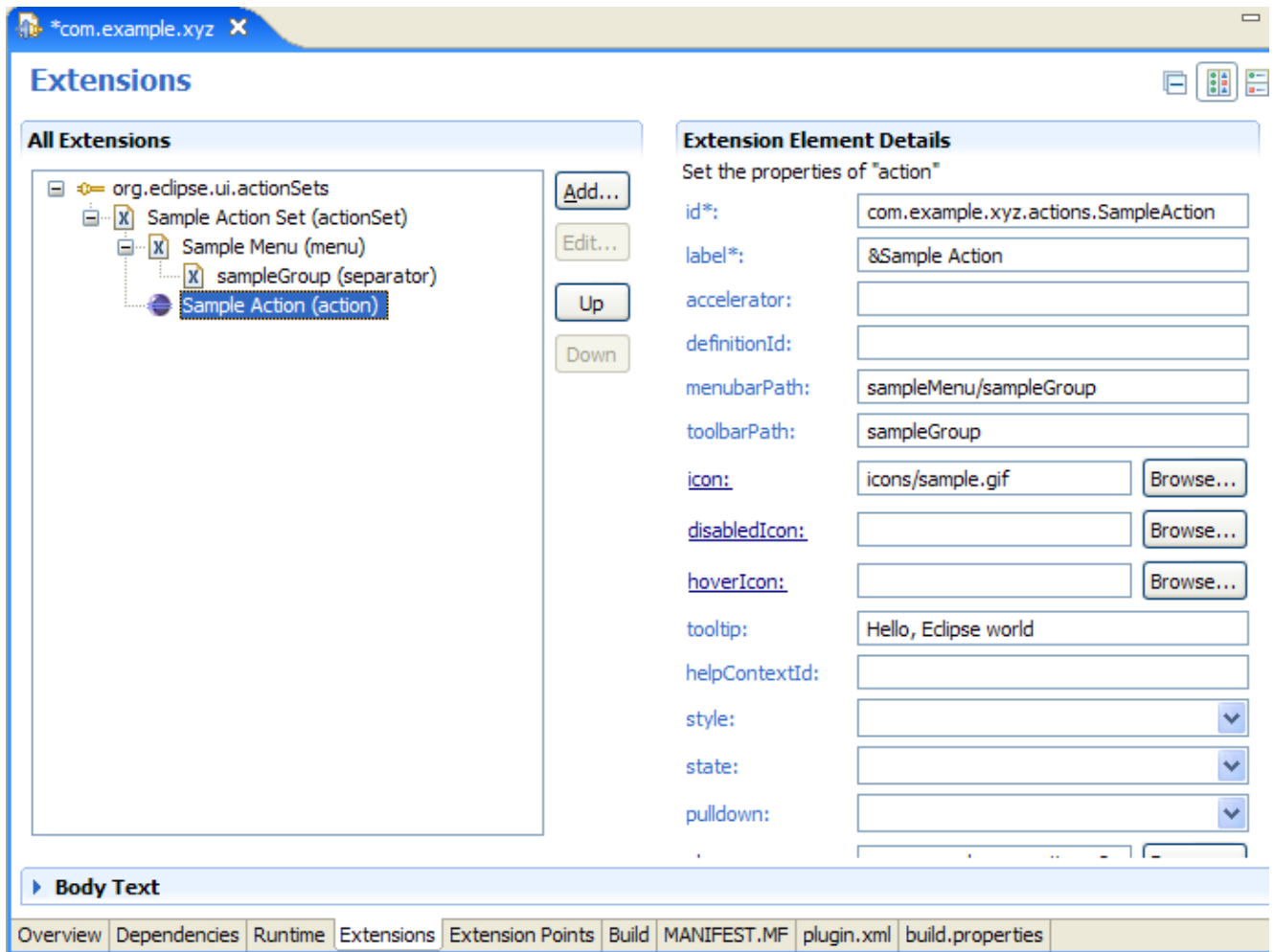
You can inspect the access restriction rules imposed on each classpath entry by PDE on the **Java Build Path** property page of your plug-in project.



Extensions page

Extensions are the central mechanism for contributing behavior to the platform. Unless your plug-in is a simple Java API library made available to other plug-ins, new behavior is contributed as an extension.

The Extensions page is where you can add, remove and modify the extensions your plug-in contributes to the platform.



Each extension point comes with an xml schema specifying its grammar. Your extension's syntax must therefore follow that grammar in order to be processed correctly. When you create a new extension, PDE extracts the grammar for the corresponding extension point and populates the context menu of each element selected in the Extensions viewer with a list of valid child elements that you can create.

Also, for each selected element in the body of an extension, PDE populates the Extension Element Details section with all the valid attributes for that element. Required attributes are denoted with an asterisk.

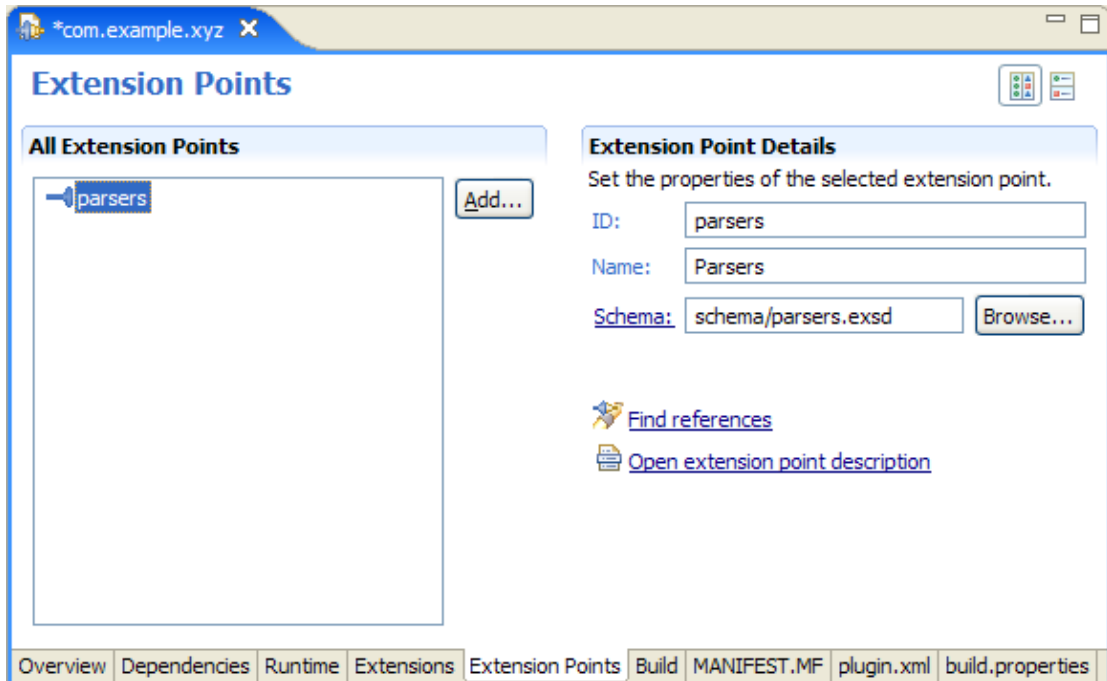
When you hover your mouse over an attribute name, a tooltip box appears describing the purpose of that attribute.

When an attribute expects the name of a Java class as a value, e.g. the *class* attribute above, clicking on the name of the attribute will open the Java file specified if it exists. If the file does not exist, then clicking on the *class* link will bring up the JDT New Class wizard to create a new Java class on the fly. PDE will prime the wizard with the correct superclass and/or interface when the schema for the extension point specifies this information for the given attribute.

Extension points page

Extension points define new function points for the platform that other plug-ins can plug into.

The Extension Points page is the place to add, remove and edit extension point declared by your plug-in.



An extension point has three attributes:

- id – a required attribute whose value is a simple name
- name – a required attribute whose value is a translatable string
- schema – an optional attribute whose value is a relative path to the schema corresponding to this extension point.

Although schema is an optional attribute, you are strongly encouraged to provide a schema, so that PDE could use it to make it easy for developers to use your extension point.

PDE provides a [schema editor](#) to help you create a schema for your extension point.

Extension point schema editor

You can open the extension point schema editor in two ways: as a by-product of creating a new extension point or by opening an existing extension point schema. By convention, new schemas have the same name as the extension point id with a **.exsd** file extension. They are placed in **schema** directory in your plug-in directory tree.

When a new extension point is created in the PDE, the initial schema file will also be created and the schema editor will be opened for editing. You can define the schema right away, or close it and do it later. Making a

complete extension point schema allows PDE to offer automated assistance to the users of your extension point.

The PDE schema editor is based on the same concepts as the plug-in manifest editor. It has two form pages and one source page. Since XML schema is verbose and can be hard to read in its source form, you should use the form pages for most of the editing. The source page is useful for reading the resulting source code.

Example: Creating schema for the "Sample Parsers" extension point

We previously created the "Sample Parsers" extension point and the initial schema. We can now open the schema by going into the **schema** folder of our project and double-clicking on the **parsers.exsd** file. This will open the schema editor.

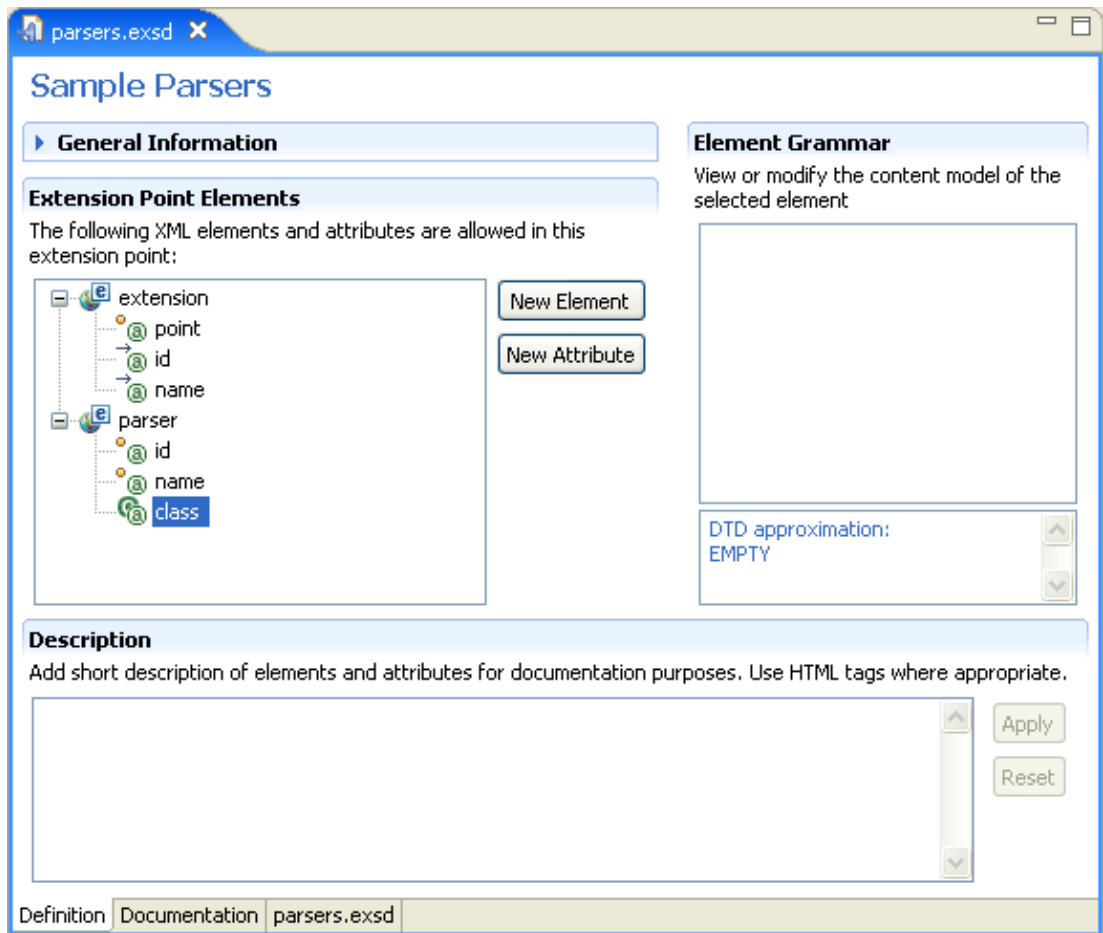
We want to do the following:

1. Define the valid XML elements and attributes for the extension point.
2. Define the grammar (content model).
3. Provide documentation snippets that will be merged into a reference document.

Every extension point schema starts with a declaration of the "extension" element. We will add a new XML element called "parser."

1. Press the button **New Element** in the **Extension Points Elements** section.
2. Move to the Properties view and change its name from "New_Element" to "parser"
3. While the new element is still selected, press the **New Attribute** button. This will create "new_attribute" as its child. Change its **name** to "id" and **use** to "required" in the property sheet.
4. While still in the property sheet, press the button "Clone this attribute" available on the local tool bar. This will create a copy of the attribute. This is useful because it allows us to quickly define all the attributes without leaving the property sheet.
5. Change the name of the new attribute into "name."
6. Clone the attribute again. This time, change the name to "class." This attribute will be used to represent a fully qualified name of the Java class that must implement a specific Java interface. We need to specify this so that PDE can later take advantage of it. Change **kind** from "string" to "java." Set the **basedOn** property to **com.example.xyz.IParser**. (This interface does not exist yet, but we will make it later.)

So far, the schema editor should look like this:

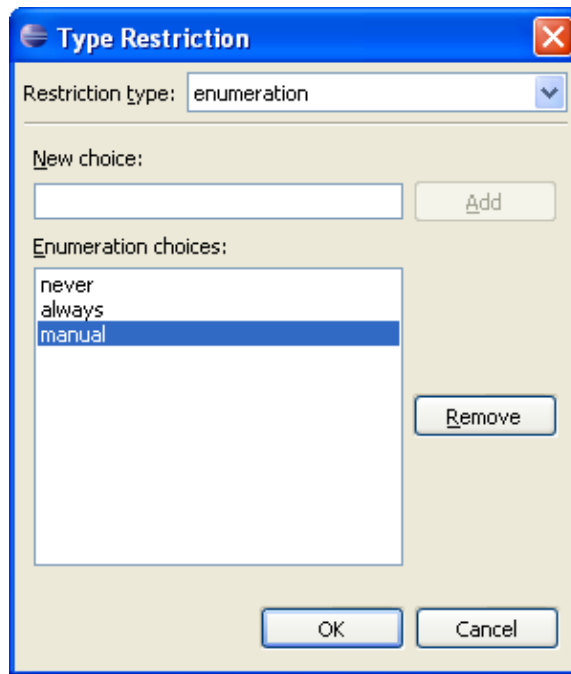


We will now add an additional attribute that takes values from a discrete list of choices. This means we need to create an enumeration restriction of the base **string** type. In addition, we will set a default value for the attribute.

1. While the "parser" element is selected, press the **New Attribute** button. Change its name in the property sheet to "mode."
2. Click in the value cell of the "restriction" property to bring the restriction dialog up.
3. Change the restriction type from "none" to "enumeration."
4. Add the following three choices: "never," "always," and "manual." (These are our three hypothetical modes for the parser extension.)

The restriction dialog should look like this:

Using the Plug-in Development Environment

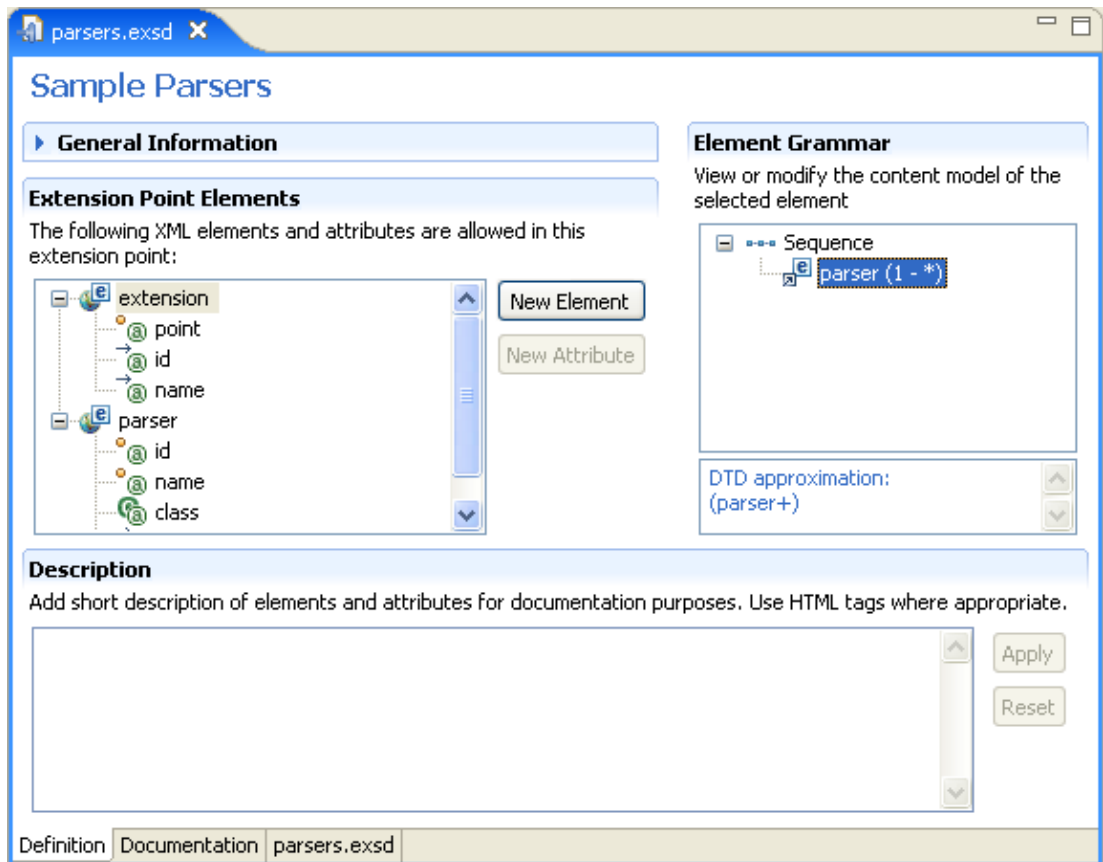


When the dialog is closed, change the "use" attribute to "default" and "value" attribute to "always." This establishes the default value. Note that the status line shows an error message as you are typing the value, since it restricts valid values to the three enumeration choices. Once you finish typing, the error message should go away because "always" is a valid value.

Now that we have defined all of the elements and attributes, we need to define grammar. Our goal is to define that the "extension" element can have any number of "parser" elements as children.

1. Select the "extension" element. Its initial content model shows an empty sequence compositor.
2. Select the sequence compositor and select **New→Reference→parser** from the popup menu. This will add the parser reference to the sequence compositor.
3. The default cardinality of references is [1,1] which means that there can be exactly one "parser" element. That is not quite what we wanted. We select the "parser" reference and change the **maxOccurs** to "unbounded."

After defining the grammar, the DTD approximation below the grammar section shows what the grammar for the selected element would look like in DTD. This information is provided to help developers who are still more comfortable with DTDs than XML schemas.



Now that we have defined valid elements, attributes and grammar, we need to provide some information about the extension point. There are two types of schema documentation snippets:

- Documentation about elements and attributes
- Documentation about the extension point usage, API, etc.

The first snippet type is provided in the Definition page of the schema manifest. As you select elements and attributes, you can add short text about them in the "Description" section. The expected format is raw HTML (as with Javadoc) and the text will be copied as-is into the final reference document.

1. Select the "id" attribute of the "parser" element and type the following in the Description editor:
a unique name that will be used to reference this parser.
2. Select the "name" attribute and add the following text:
a translatable name that will be used for presenting this parser in the UI.
3. Select the "class" attribute and add the following text (note the HTML tags):
a fully qualified name of the Java class that implements `<samp>com.example.xyz.IParser</samp>` interface.
4. Select the "mode" attribute and add the following:
an optional flag that indicates how often this parser instance will run (default is `<samp>always</samp>`).

We now have to provide a short text description of the extension point itself. In order to do that, we switch to the Documentation page:

1. You should now be on the "Overview" tab. In the text editor and add the following text:

Using the Plug-in Development Environment

This extension point is used to plug in additional parsers. The parsers actually do not work – we have just used them as an example of extension point schema.

Press **Apply**.

2. Click on the "Examples" tab and add the following text:

The following is an example of the extension point usage:

```
<p>
<pre>
    <extension point="com.example.xyz.parsers">
        <parser
            id="com.example.xyz.parser1"
            name="Sample Parser 1"
            class="com.example.xyz.SampleParser1">
        </parser>
    </extension>
</pre>
</p>
```

Press **Apply**.

3. Select the "API Information" tab and add the following text:

Plug-ins that want to extend this extension point must implement
<samp>com.example.xyz.IParser</samp> interface.

Press **Apply**.

4. Select the "Supplied Implementation" tab and add the following text:

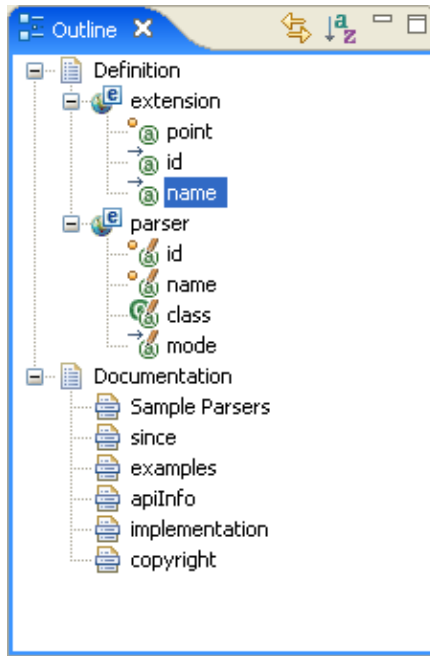
XYZ Plug-in provides default implementation of the parser.

Press **Apply**.

Note: Special consideration has to be taken when providing examples. Normally, PDE would treat the provided text as raw HTML and would not respect new line and white space greater than one character (i.e. ignorable white space). This is to be expected when regular text is concerned, but is extremely annoying when providing examples, where tabbing and vertical alignment is significant if the example is to look good. PDE has a compromise for this situation: if it detects the HTML tag <pre>, it will take the content as-is (preserving all characters without modification) until closing tag </pre> is seen. This is why we can provide an example like the above and be confident that it will look good in the final reference document.

You may have already noticed that as you type documentation, more and more elements in the editor Outline view acquire a "pen" image overlay. This little indicator tells you that the element in question has some text associated with it – a quick way to check if the documentation is missing somewhere in the document.

Using the Plug-in Development Environment



Once we have finished with the documentation, we can take a look at the reference documentation. You can do it in two ways. All the time during your work, you can preview the reference document by selecting Preview Reference Document item from the pop-up menu. Alternatively, you can set up PDE preferences (Preferences>Plug-in Development>Compilers, under Schema tab) to automatically create reference documentation on each schema file change. Regardless of the method to create it, the resulting document for this example would look like [this](#).

Sample Parsers

Identifier:

com.example.xyz.parsers

Since:

3.0

Description:

This extension point is used to plug in additional parsers. The parsers actually do not work – we have just used them as an example of extension point schema.

Configuration Markup:

```
<!ELEMENT extension EMPTY>
```

```
<!ATTLIST extension
```

```
point CDATA #REQUIRED
```

```
id CDATA #IMPLIED
```

```
name CDATA #IMPLIED>
```

```
<!ELEMENT parser (parser+)>
```

```
<!ATTLIST parser
```

```
id CDATA #REQUIRED
```

```
name CDATA #REQUIRED
```

```
class CDATA #REQUIRED
```

```
mode (manuallalwaysnever) >
```

- **id** – a unique name that will be used to reference this parser.
- **name** – a translatable name that will be used for presenting this parser in the UI
- **class** – a fully qualified name of the Java class that implements `com.example.xyz.IParser` interface
- **mode** – an optional flag that indicates how often this parser instance will run (default is `always`).

Using the Plug-in Development Environment

Examples:

The following is an example of the extension point usage:

```
<extension point=  
    "com.example.xyz.parsers"  
>  
    <parser id=  
        "com.example.xyz.parser1"  
        name=  
            s  
            "Sample Parser 1"  
        class=  
            "com.example.xyz.SampleParser1"  
    >  
</parser>  
</extension>
```

API Information:

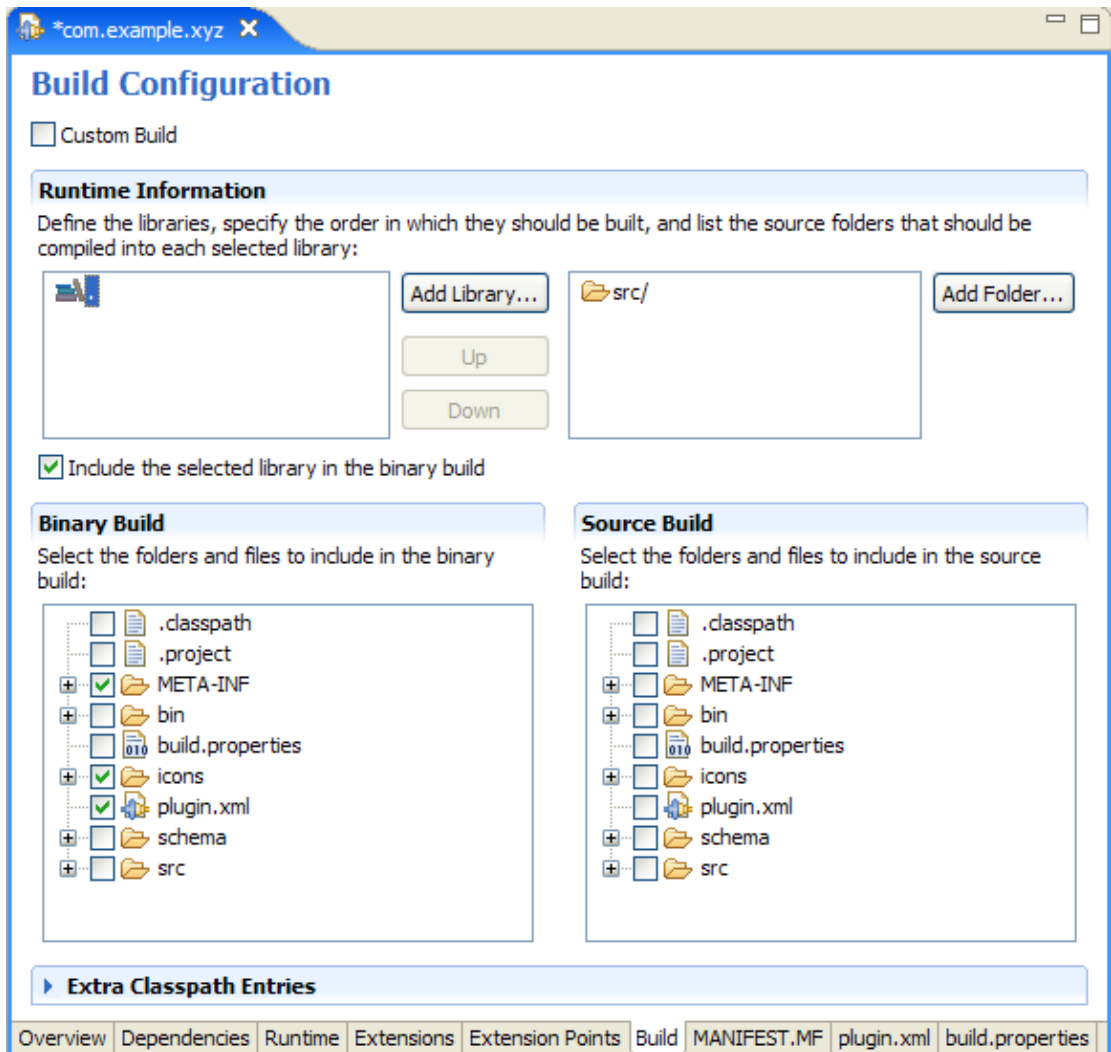
Plug-ins that want to extend this extension point must implement `com.example.xyz.IParser` interface.

Supplied Implementation:

XYZ Plug-in provides default implementation of the parser.

Build configuration page

The **Build Configuration** page contains all the information needed to build, package and export the plug-in. It appears as a page in the plug-in manifest editor, but note that changes made to it will be written by PDE to the **build.properties** file of the plug-in. This file solely guides the build process.



The **Runtime Information** section lists all the libraries that you want to build. For each library, you must list the source folder(s) that will be compiled into the library.

If your plug-in declares more than one library, order them correctly using the Up and Down button, so that they get compiled in the correct order.

The **Binary Build** section is where you select all the files and folders that will make it into the packaged plug-in. In this example, you will see that we only want to the plugin.xml file, icons folder and the xyz.jar to end up in the packaged plug-in.

The **Source Build** section has a specialized purpose and is not commonly used or needed by the general population. It is needed only when you are shipping source in separate plug-ins and features than the binary plug-ins. See the [org.eclipse.pde.core.source](#) extension point for details.

If the libraries you want to build include libraries that are NOT listed on the Runtime page, and you need extra libraries on the build path for them to compile, you can add these required JARs in the **Extra Classpath Entries** section.

Source Locations

Identifier:

org.eclipse.pde.core.source

Since:

2.0

Description:

This extension point allows PDE to find source archives for libraries in Eclipse plug-ins found in an Eclipse-based product. It is used to contribute locations that contain source archives. These locations are expected to contain the same layout as the 'plugins' directory.

For each plug-in or fragment, a directory in the form {id}_{version} should exist. The content of the directory corresponds to the plug-in/fragment location. It should contain source code zip file in the form {library name}src.zip where library name is the name of the Java library that matches the source code. For JAR'd plug-ins and libraries named '.', the source zip file must be named simply `src.zip`.

In addition, it should contain any file or directory specified in the build.properties using `src.includes` variable.

Configuration Markup:

<!ELEMENT extension (location+)>

<!--ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

<!--ELEMENT location EMPTY>

<!--ATTLIST location

path CDATA #REQUIRED>

- **path** – the relative path of the directory in the contributing plug-in where source content is stored. The folder must contain one or more directories in the form {id}_{version} where `id` is a matching plug-in or fragment identifier and `version` is the matching plugin/fragment version. These directories in turn should contain source archives and any other file or folder specified using `source.includes` variable in build.properties file of the corresponding plug-in/fragment.

Examples:

The following is an example of the `source` extension:

```
<extension point =  
"org.eclipse.pde.core.source"  
>  
<location path=  
"src"  
>  
</extension>
```

In the example above, the source location `src` in the contributing plug-in has been registered.

API Information:

No Java code is required for this extension point.

Supplied Implementation:

Eclipse SDK comes with source plug-ins that contain source information for all the plug-ins and fragments in Eclipse SDK.

Copyright (c) 2004 IBM Corporation and others.

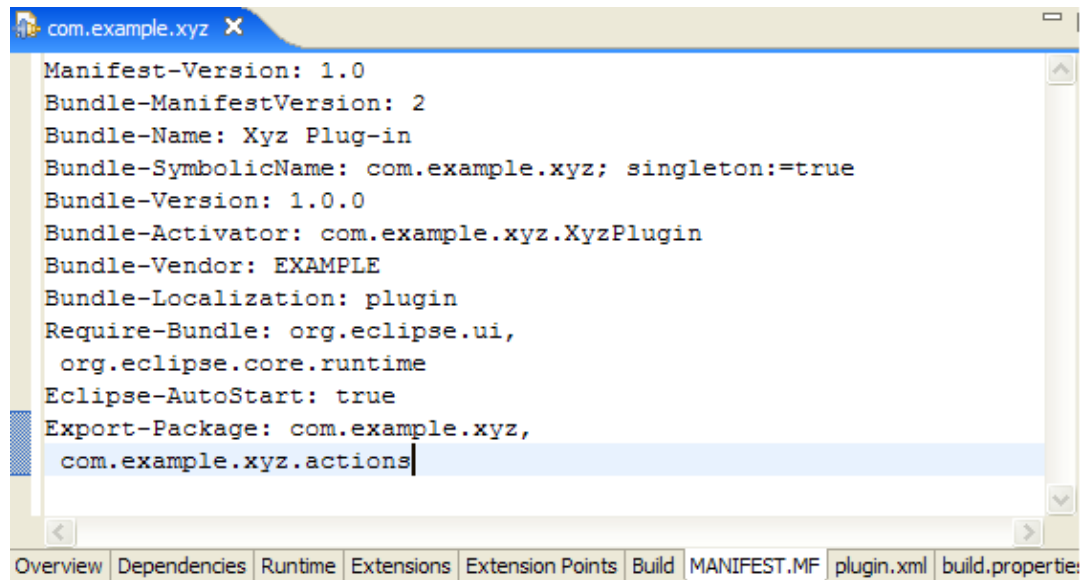
All rights reserved. This program and the accompanying materials are made available under the terms of the Eclipse Public License v1.0 which accompanies this distribution, and is available at <http://www.eclipse.org/legal/epl-v10.html>.

Source pages

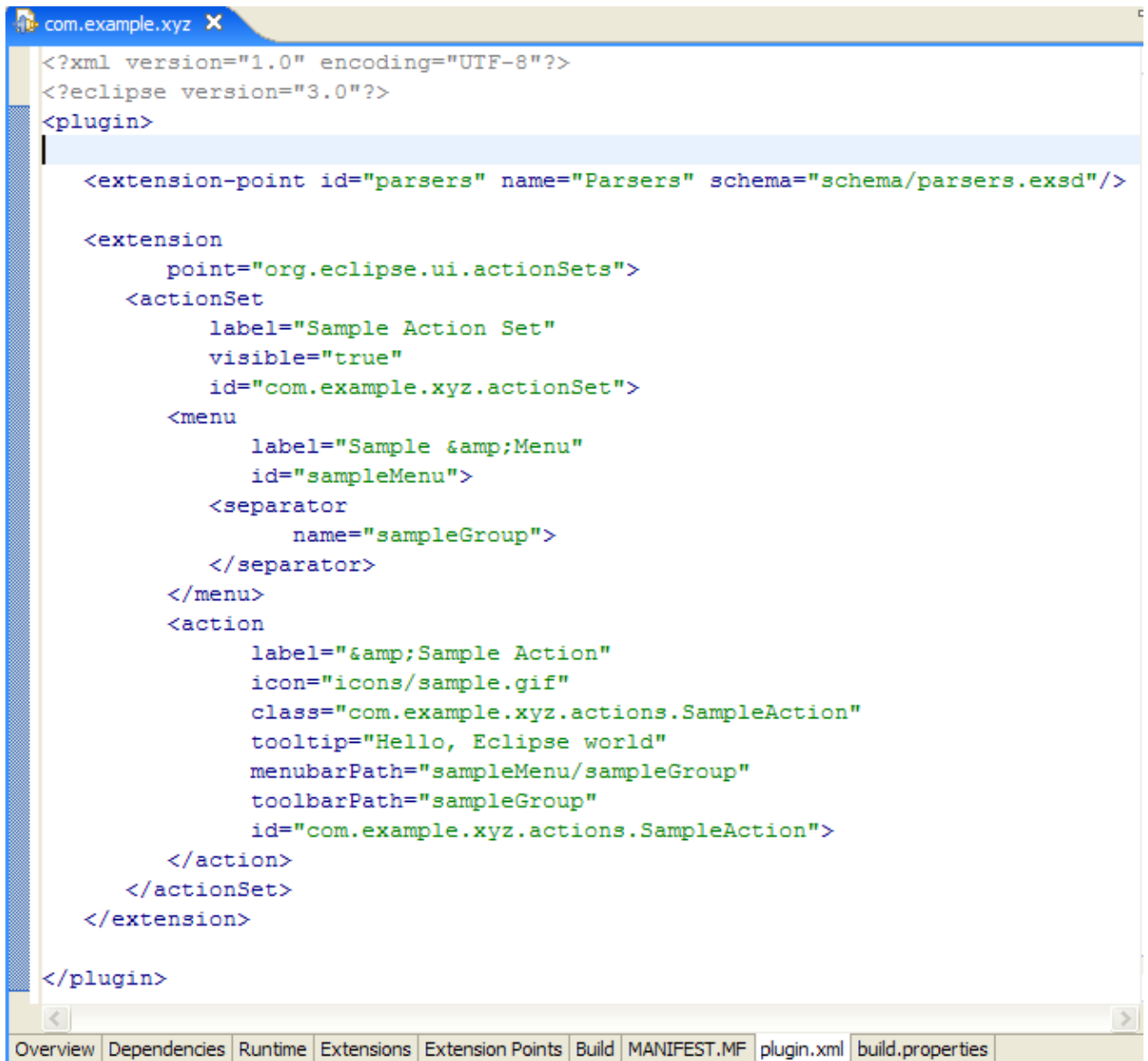
The plug-in editor manages all three plug-in files at the same time.

The `manifest.mf` file is where all the plug-in data and dependencies is stored.

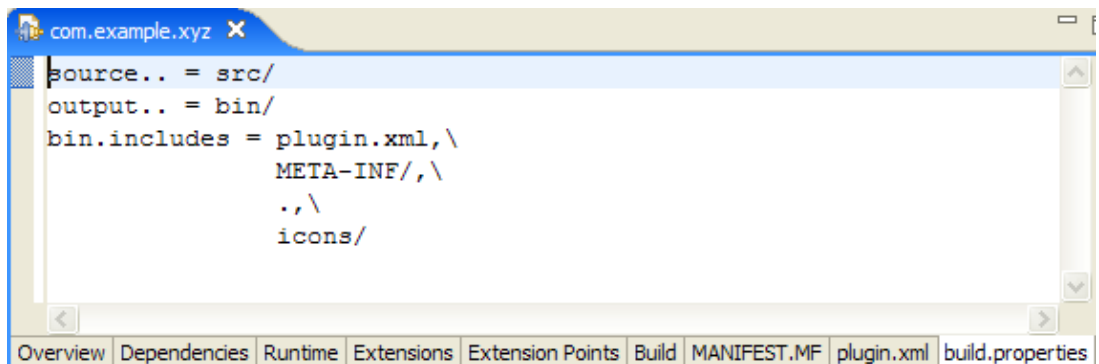
Using the Plug-in Development Environment



The plugin.xml file contains the extensions and extension points declared the plug-in.

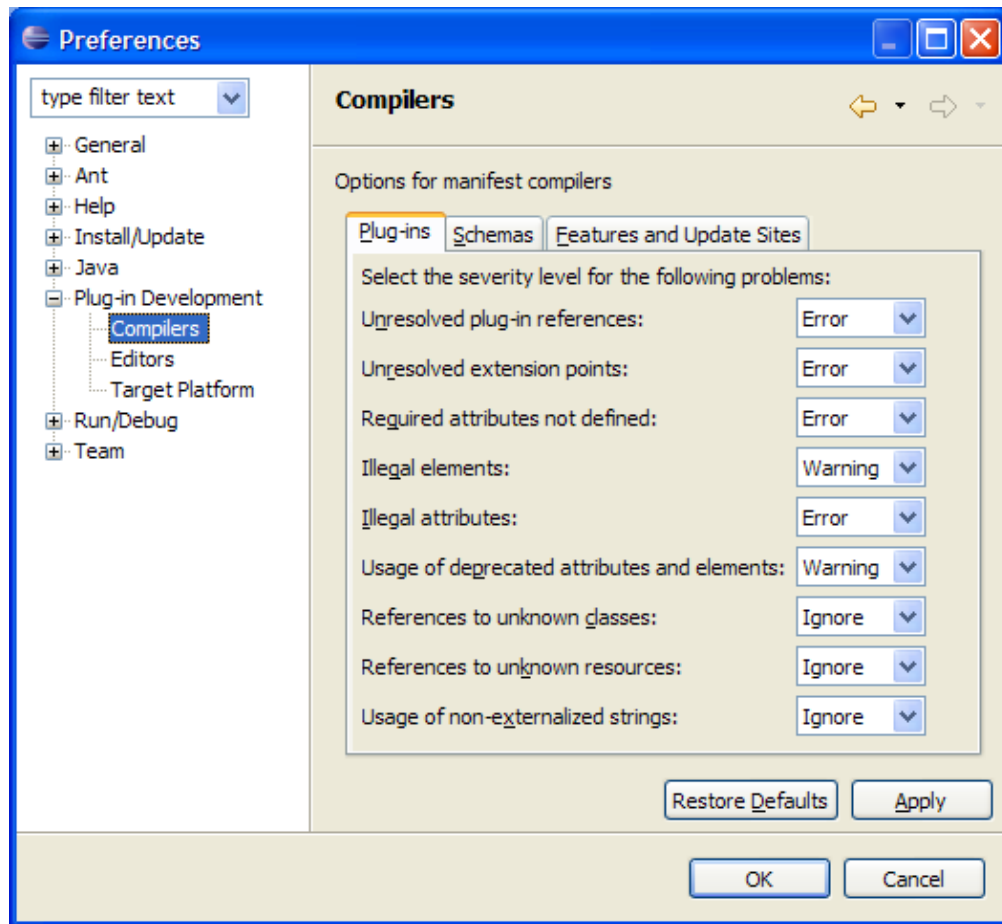


The build.properties file contains all the information necessary to build the plug-in.



PDE validates the manifest.mf and plugin.xml for semantic errors and flags violations. The list problems that PDE flags are listed on the **Plug-ins** tab of the **Plug-in Development > Compilers** preference page:

Using the Plug-in Development Environment



You can set the level of each to one of *Error*, *Warning* or *Ignore*.

Note that for PDE to be able to detect some of these problems (e.g. required attributes not defined, undefined extension attributes, etc.), extension points you are using must have a valid schema associated with them. See [Extension Point Schema](#) for more details.

Extension point schema

Extensions are the key mechanism that a plug-in uses to add new features to the platform. Extensions cannot be arbitrarily created. They are declared using a clear specification defined by an extension point.

Each extension must conform to the specification of the extension point that it is extending. Each extension point defines attributes and expected values that must be declared by an extension. This information is maintained in the platform plug-in registry. Extension point providers query these values from the registry, so it's important to ensure that your plug-in has provided the expected information.

In the most rudimentary form, an extension point declaration is very simple. It defines the id and name of the extension point. Any other information expected by the extension point is specific to that extension point and is documented elsewhere. (See the [Platform Extension Point Reference](#) for the platform extension point definitions.)

Reference documentation is useful, but it does not enable any programmatic help for validating the specification of an extension. For this reason, PDE introduces an extension point schema that describes extension points in a format fit for automated processing.

Extension point schema is a valid XML schema as defined by W3C specification. However, the full XML schema specification is very complex and mostly unnecessary for this particular use. For this reason, PDE uses only a subset of XML schema features. Each extension point schema is a valid XML schema, but PDE does not use all the available features.

The benefits of extension point schemas

There are many benefits to describing your extension point using the PDE extension point XML schema:

1. Extension point grammar allows elements, attributes, and types to be expressed formally. This information can be used by tools to validate extensions or offer assistance during creation of the extension.
2. XML schema provides for documentation annotation that is similar to Javadoc in Java source. This mechanism ties short text for valid elements and attributes to the declaration of these elements and attributes. It is much easier to keep the documentation in sync because removal of an attribute will also remove documentation for the attribute. There is no need to update the reference document.
3. Reference documentation can be generated. PDE provides a tool that tracks changes in extension point schemas and updates reference documentation on the fly.
4. You can provide additional metadata about the extension point that can be used by tools that process the schema. PDE uses this mechanism to add additional information about elements and attributes. For example, if an attribute is marked as "Java," PDE can provide assistance while setting the value of this attribute by interacting with Java platform features.

Limitations of PDE XML Schema support

PDE uses a small subset of XML schema. Using the full XML schema features set would be an overkill in this particular case. The subset allows almost 1-to-1 mapping from DTDs to schemas, but without DTD limitations. The following are the main limitations of the PDE extension point schema:

1. Only global element declarations are allowed.

Using the Plug-in Development Environment

2. Only local attribute declarations are allowed. Global attributes cannot be declared.
3. The following compositors are supported: **all**, **sequence**, **choice** and **group**.
4. There is no global type support. Types must be declared and immediately used.
5. Attributes can only have **string** and **boolean** types.
6. If an attribute is of type **string**, only the **enumeration** restriction is supported.

If you write an XML schema using these restrictions, you will notice that the resulting file looks strikingly similar to an equivalent DTD that defines the same grammar. The advantage of schema is in annotations (both documentation and metadata). An additional advantage is that the XML schema is itself written in XML, which makes its processing and reading much easier.

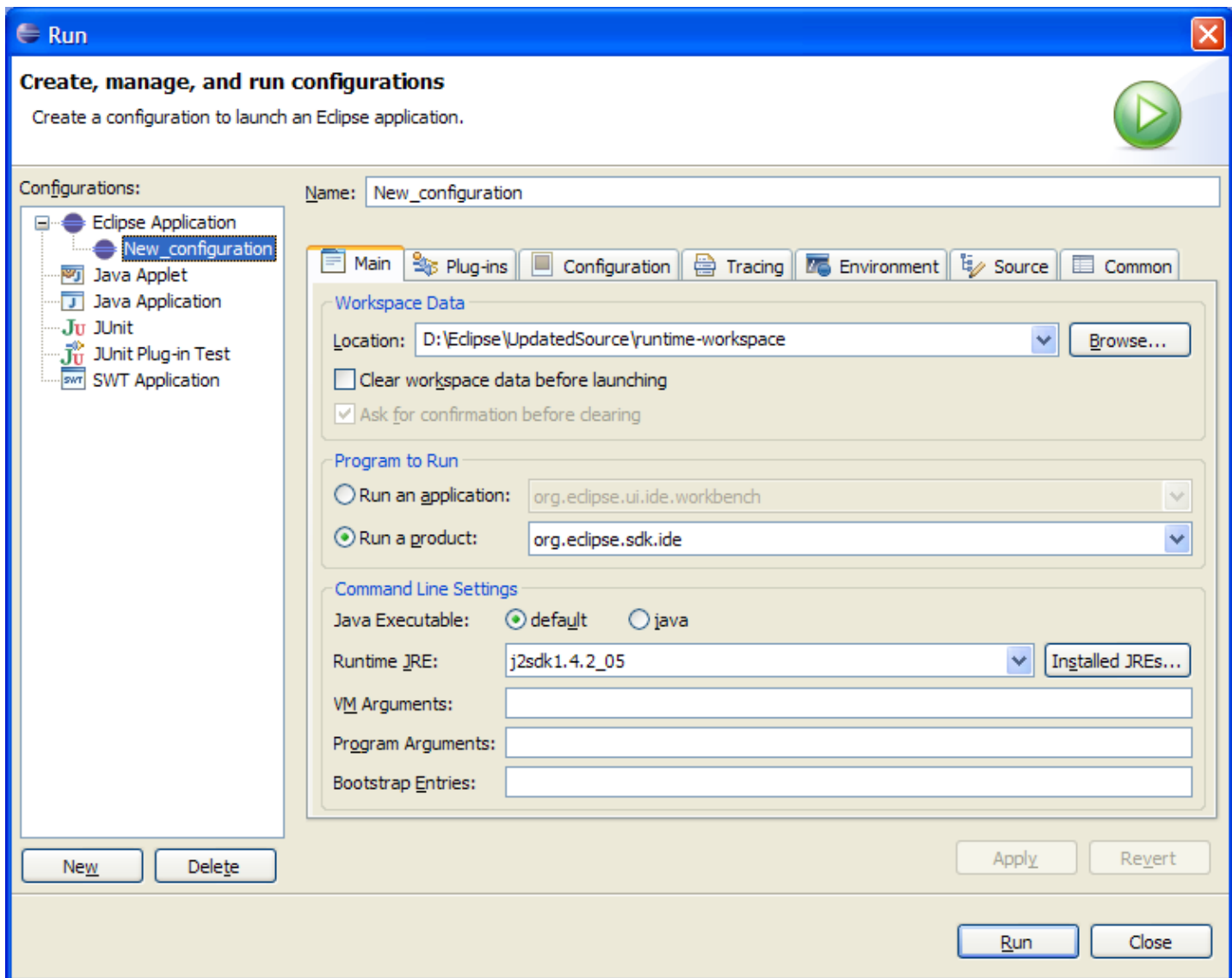
The list above is for reference only. You are most likely to define an XML schema using the PDE schema editor that will take care of generating the correct file.

Running a plug-in

As you develop your plug-in in the workspace, the incremental Java compiler will compile your Java source code and place the *.class files into the **bin** directory of your PDE project. When you are ready to test your plug-in, you can launch a separate Eclipse application instance to test your new plug-in.

The easiest way to launch an Eclipse applicaiton is via the link in the Testing section of the plug-in manifest editor's Overview page. This will immediately create a second Eclipse instance that will appear within seconds.

To gain full control over the way the run-time workbench is launched, select **Run > Run...** from the main menu bar. This will bring up the Launch Configuration Dialog.



Workspace data field defines the workspace that will be used by your application. The location of this runtime workspace must be different from the workspace of your host instance.

Using the Plug-in Development Environment

The default program to run is the *org.eclipse.sdk.ide* product. Launching it will result in a second workbench instance coming up whose constituent plug-ins are the workspace plug-ins and the plug-ins selected on the Target Platform preference page.

You can test your runtime workbench using the **JRE** of your choice and does not have to be the same one against which your plug-ins compile in the workspace. You can also specify any VM arguments that are appropriate for your testing.

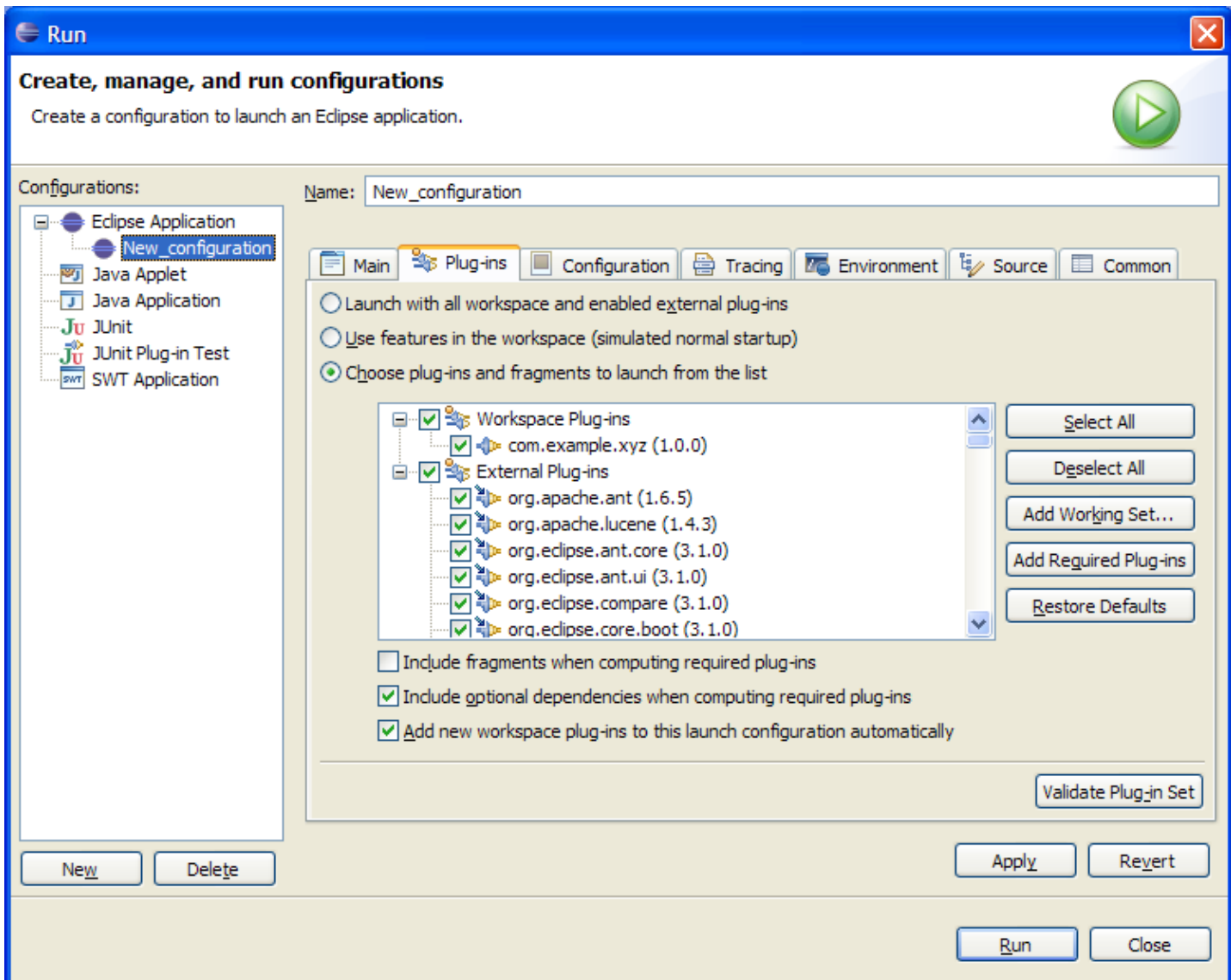
Example: Running the Sample

Press **Run**. Another platform instance should open. You will see a top menu item called "Sample Menu" with a single "Sample Action" item. Selecting it should pop up a dialog containing the phrase "Hello, world".

Choosing plug-ins to run

By default, the Eclipse application is launched with all the workspace plug-ins and all the plug-ins selected on the *Plug-in Development > Target Platform* preference page.

It is possible however to hand-select a subset of plug-ins to launch with on the *Plug-ins* tab of the launch configuration.



This gives you great flexibility, but with flexibility comes the danger of ending up with an invalid configuration. Use this feature carefully and press the **Validate Plug-in Set** often to ensure that all inter-plug-in dependencies in your subset are satisfied.

Running with tracing

The platform provides a mechanism for tracking the activity of your plug-in at runtime without full debugging. It allows you to use tracing flags that will cause tracing information to be printed on the standard output (or Console view). These flags are defined in files named **".options"** and have the following syntax:

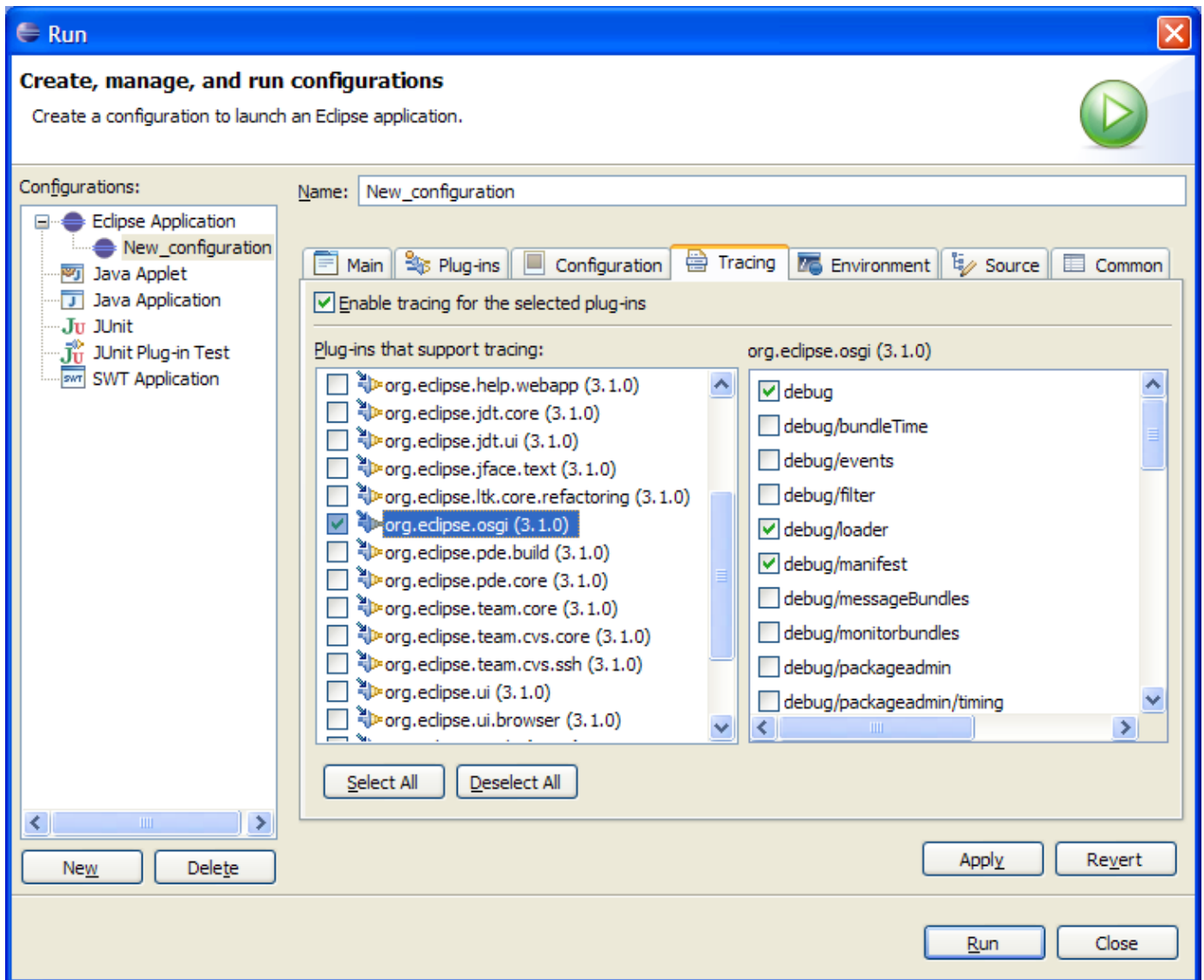
```
<plug-in Id>/debug = true/false (master switch)
<plug-in Id>/<tracing flag> = <value>
```

The first entry represents master switch for tracing your plug-in. If you call the method **isDebugging** in your plug-in class, it will return true if the value of this tracing variable is true. Other tracing flags are defined by you and their value can be obtained by using

Using the Plug-in Development Environment

```
Platform.getDebugOption(optionName);
```

Most of the platform plug-ins define tracing flags, particularly the platform core. For a new plug-in developer, the most interesting set of tracing flags are those related to class loading, because they can allow tracing of plug-in loading problems.



Example: Adding tracing support to your plug-in

If you add tracing support to the plug-in under development, your plug-ins will appear in the list of plug-ins that support tracing.

In order to allow other developers to control your plug-in's tracing flags, you need to make these options known. This is typically done by placing a **.options** file in your plug-in. The file lists all the supported flags as well as their default values.

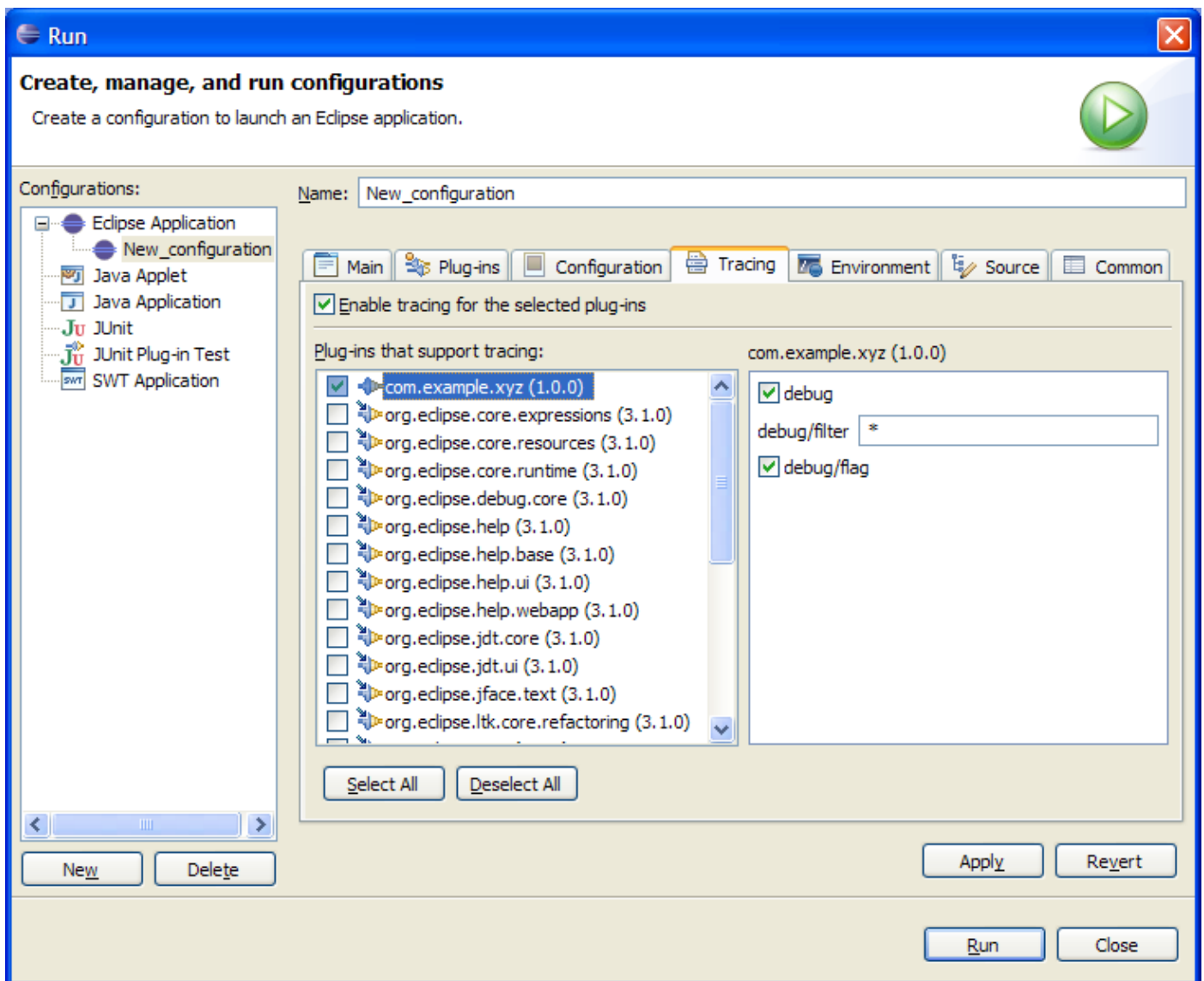
We will now define a template **.options** file with a few tracing flags for our new plug-in.

Using the Plug-in Development Environment

Select the **com.example.xyz** project created earlier and create a new file **.options**. When the default text editor opens, add the following entries:

```
com.example.xyz/debug = true
com.example.xyz/debug/flag = true
com.example.xyz/debug/filter = *
```

When this file is saved, select **Run > Run...** to open the launch dialog. Our plug-in should now show up in the list. When selected, it should show the newly defined flags with their default values.



Creating the **.options** file only defines the availability flags, allowing other plug-in developers to define the values of the tracing properties. You will still need to check the values of your tracing properties in your plug-in code using **Platform.getDebugOption()**.

Exporting a plug-in

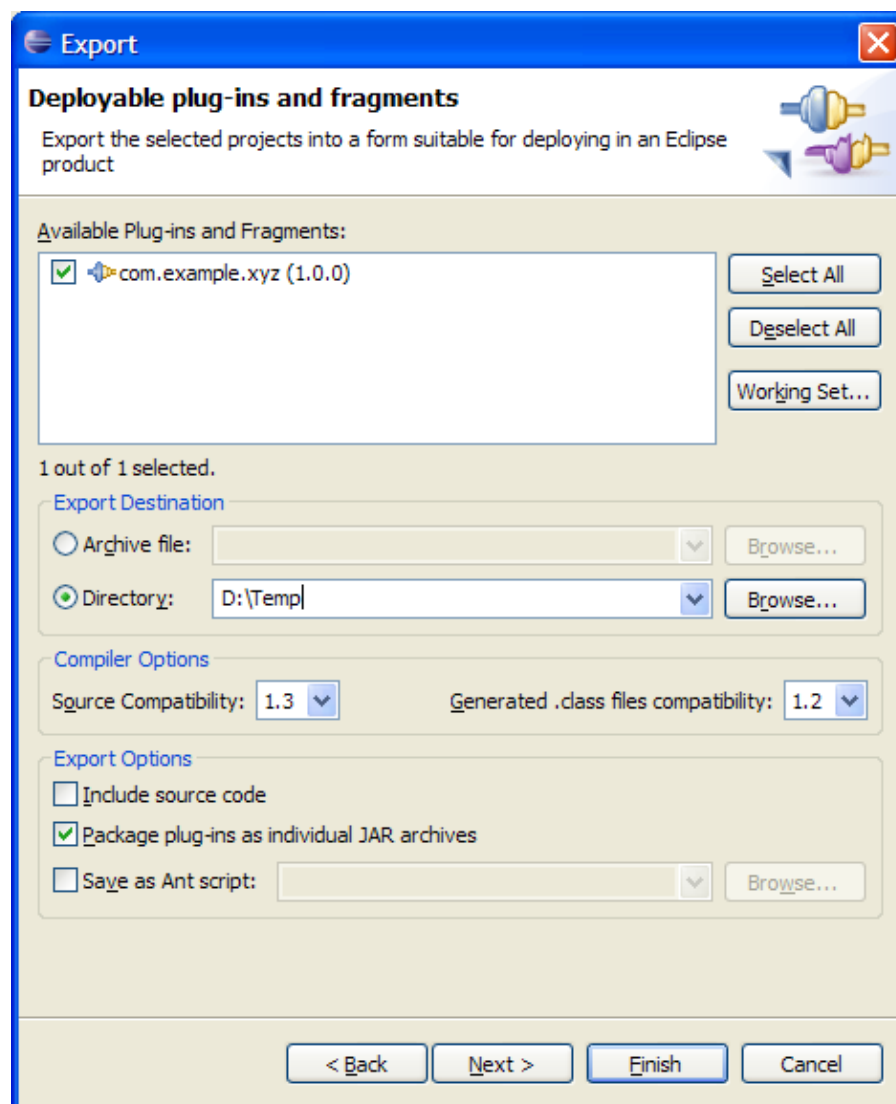
During the design phase, plug-ins and fragments in your workspace are used as-is so that you can quickly test and debug. Once you reach the stage where you are satisfied with your code, you need to publish it in a form suitable for deployment in an Eclipse product.

The easiest way to do so is through the **Export Plug-ins and Fragments Wizard**. It shields you from ant scripts and does not pollute your workspace with resources generated during the build operations:

Select **File > Export... > Deployable plug-ins and fragments**

Select the plug-ins and fragments you want to export.

The export destination can be an archive or a directory.



The recommended format for a deployable plug-in in 3.1 to be packaged as a JAR archive. Refer to the [Exporting your plug-in as a JAR](#) document for details if you have a pre-3.1 plug-in that you need to JAR.

You also have the option to save the settings of this export operation. This way you would be able to redo this export operation without having to go through the wizard all over again.

Alternatively, plug-in JARs could be built manually. Refer to the [Creating Ant Scripts from PDE](#) section.

Shipping Your Plug-in As A Single JAR

Eclipse 3.0 and previous was shipped such that each plug-in was a directory that contained code in a JAR, along with multiple other files. In order to improve the number of files that we ship along with the size of the Eclipse distributions, we have added support in Eclipse to be able to ship each plug-in as a single JAR file containing its code and other resources.

Converting A Plug-in to be Shipped as a JAR

1. Change the classpath:

- ◆ If you have a `manifest.mf` then simply delete the `Bundle-Classpath` header.
- ◆ Otherwise if you have a `plugin.xml` then change the library entry to be a dot like this:

```
<runtime>
  <library name=".">
    <export name="*" />
  </library>
</runtime>
```

2. Change the `build.properties`:

- ◆ Change all occurrences of the old jar name to simply a dot. (.)
- ◆ There should be one on the `bin.includes` line. For instance, if your `bin.includes` line used to have `core.jar`, that will be replaced with a `.`. For instance:
`bin.includes=about.html, ., META-INF/MANIFEST.MF`
- ◆ Change `source.foo.jar=` to `source..=` (that is source dot dot)
- ◆ Change `output.foo.jar=` to `output..=` (that is output dot dot)
- ◆ There may be others like `jars.compile.order`, etc
- ◆ If you newly generated a `manifest.mf`, then add `META-INF/` to the `bin.includes`.

3. Change the feature

- ◆ find all features that list your plug-in and add following to the related `<plugin>` tag:
`unpack="false"`

4. Change `about.html` linked content.

- ◆ If you have a basic `about.html` with no linked files, then you don't need to do anything.
- ◆ If you have content in your plug-in which is linked from your `about.html` file, then that content must be placed in a directory named `"about_files"` at the root of the plug-in.
- ◆ Make sure to change the links in the `about.html` to point to the new location of the files!

If you have a custom build script (`build.xml`):

- You need to ensure that your script will still work against plug-ins that are JAR'd as well as plug-ins which aren't JAR'd.

If you have other scripts (e.g. javadoc generation scripts):

- If the script assumes the layout of plug-ins and needs to add code JAR files to the classpath, then it must be modified to put the whole JAR'd plug-in on the classpath rather than the individual JARs.

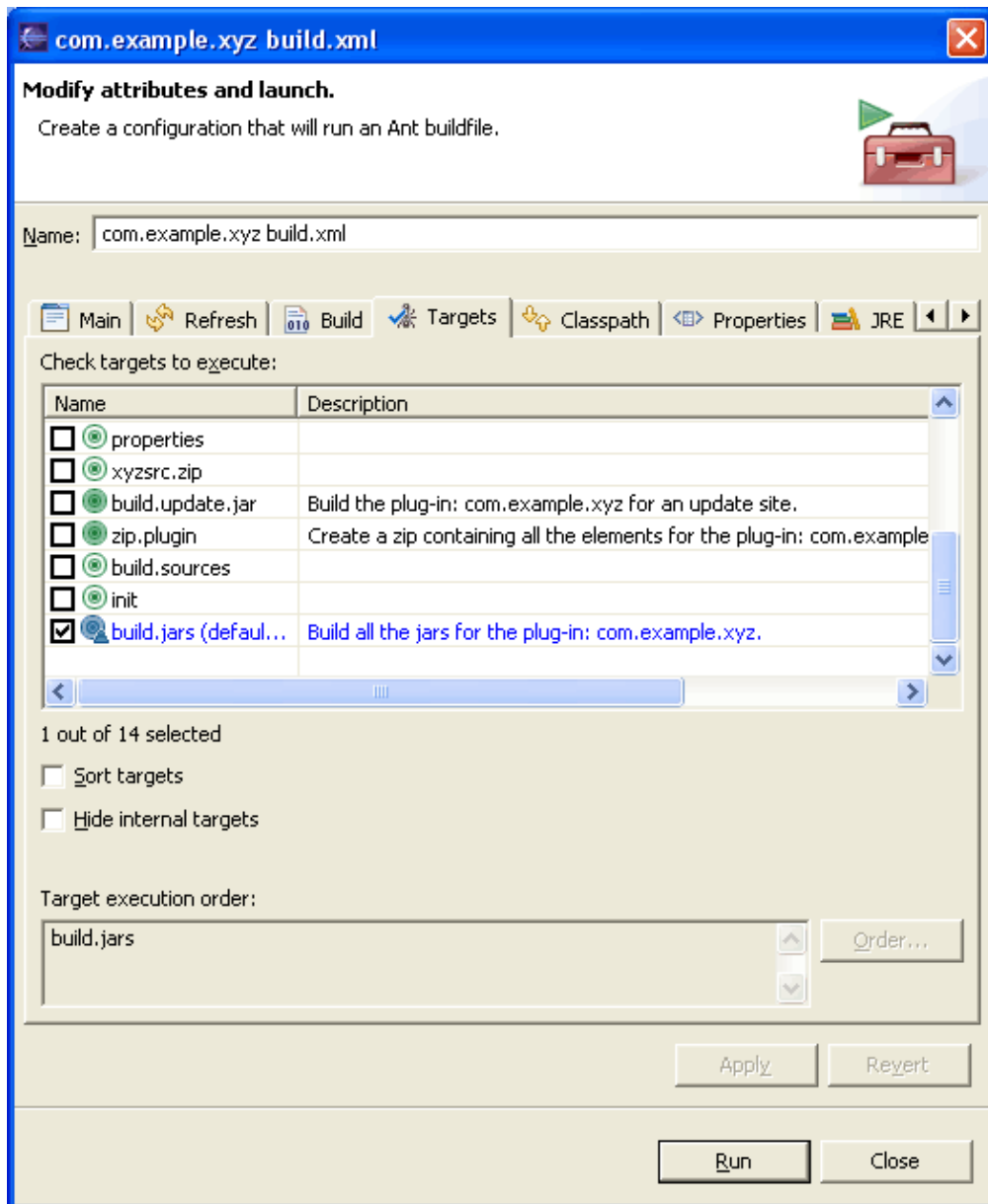
Generating Ant scripts

Ant is a simple open-source scripting engine that is capable of running scripts written in XML format. Ant is ideal for executing tasks usually found in automated builds.

The variables set in the plug-in, fragment or feature **build.properties** will be used to generate scripts for Ant. PDE generates Ant scripts for creating individual plug-in and fragment build files and one overall script for building the feature JAR. This "main" script is also responsible for running individual script files in the right order (defined by the plug-in dependency chain). Each build file has the same name (**build.xml**) and is created as a sibling of the manifest files in the corresponding projects.

Since Ant scripts use the replacement variables in **build.properties**, you can typically use them "as is", without modifying the generated scripts. If you do modify them, you must not recreate the scripts every time you want to rebuild the component.

To create scripts, you can simply select **Create Ant Build File** while a suitable manifest file (plugin.xml, fragment.xml or feature.xml) is selected in the Navigator or Package Explorer views. The command will generate the build script. After selecting **Run Ant...** from the pop-up menu while the newly generated script file is selected, the following wizard will open:



The standard Ant wizard allows customization in two ways: by providing the execution arguments and by selecting one or more build targets.

Properties

Ant arguments are typically used to provide property values that override default values and control the build process. Arguments are set using "-Dproperty=value". The following properties are recognized:

- **bootclasspath** – if set, it replaces the default boot classpath. Used when compiling cross-platform plug-ins (e.g. building a UI plug-in for Windows using Linux)
- **build.result.folder** – where the temporary files for the update JAR creation should be placed. These files are usually the plug-in library JARs.
- **plugin.destination** – where plug-in and fragment update JARs should be put. These JARs represent entire plug-ins and fragments in a format suitable for publishing on an Install/Update server and referencing by a feature. The typical layout of an Update site is to have all the plug-in and fragment

Using the Plug-in Development Environment

JARs in one place and all the features in another. This argument is useful for placing plug-ins and fragment directly into the desired directory (or the staging place on the local machine before pushing the features onto the remote server).

- **feature.destination** – where feature update JARs should be put.

To adapt the behavior of the compiler, the following properties are recognized:

- **javacFailOnError** – stop the build when an error occurs when set to true. Default is false.
- **javacDebugInfo** – compile source with debug information when set to true. Default is true.
- **javacVerbose** – produce verbose output when set to true. Default is true.
- **javacSource** – value of the `-source` command-line switch.
- **javacTarget** – generate class files for specific VM version.
- **compilerArg** – additional command line arguments for the compiler.

Targets

When executing feature build scripts, the following targets are used to call individual targets of plug-ins or fragments. In order to specify what target to execute, the property **target** should be set (e.g. `-Dtarget=refresh`). One of the **all.*** targets serves as an iterator, whereas the actual target to execute is specified via the property **target**.

- **all.plugins** – for all listed plug-ins
- **all.fragments** – for all listed fragments
- **all.children** – for all listed plug-ins and fragments
- **build.jars** – build JARs for all feature children;
- **build.sources** – build source for all feature children;
- **build.update.jar** – generate a feature JAR in the format used by the install/update mechanism. The above mentioned property **feature.destination** can be used to define where to put the JAR;
- **zip.distribution** – creates a zip file with the feature and its plug-ins and fragments in an SDK-like structure but does not include source code;
- **zip.sources** – creates a zip file with the feature and its plug-ins and fragments in an SDK-like structure which only includes the source;
- **clean** – delete everything produced by running any of the target;
- **refresh** – performs a "Refresh" action in the current project, thus making the newly generated resources visible in the Navigator or Package Explorer.
- **zip.plugin** – creates a zip file with the binary and source contents of a plug-in with the following structure:

```
id_version/  
  contents
```

where 'id' is the plug-in unique identifier and 'version' is the plug-in version. This zip file can be directly unzipped into the Eclipse installation directory as a form of a quick manual deployment.

Fragments

A plug-in **fragment** is used to provide additional plug-in functionality to an existing plug-in after it has been installed. Fragments are ideal for shipping features like language or maintenance packs that typically trail the initial products for a few months. Another frequent use of fragments is to deliver OS or windowing system-specific features.

When a fragment is detected by the platform and its parent plug-in is found, the fragment's libraries, extensions and extension points are "merged" with those of the parent plug-in.

While this merging mechanism is good from a runtime point of view, developers need to view fragments as separate entities while working on them. Fragment development is often done by different teams, on a different schedule, sometimes even on different operating systems from the original plug-in.

PDE provides full support for fragment development. Fragments can be viewed as "limited plug-ins". They have all of the capability of regular plug-ins but have no concept of life-cycle. Fragments have no top-level class with "startup" and "shutdown" methods.

Example: Writing a German fragment for XYZ Plug-in

The PDE wizards and editors that manipulate plug-ins and fragments are nearly the same. However, you must be aware of some important differences.

We start by creating a new fragment project.

On the first page of the New Fragment Project wizard, type the project name "com.example.german". Accept the default values and press **Next**.

The **Fragment Content** page has three additional fields from the plug-in creation wizard: host plug-in id, parent plug-in version, and version match rule.

Since we are writing a fragment for a specific plug-in, we can use the **Browse** button to select "com.example.xyz" in the plug-in selection dialog. Using the dialog, we could have also chosen any external plug-in.

Using the Plug-in Development Environment

New Fragment Project

Fragment Content
Enter the data required to generate the fragment.

Fragment Properties

Fragment ID:

Fragment Version:

Fragment Name:

Fragment Provider:

Classpath:

Host Plug-in

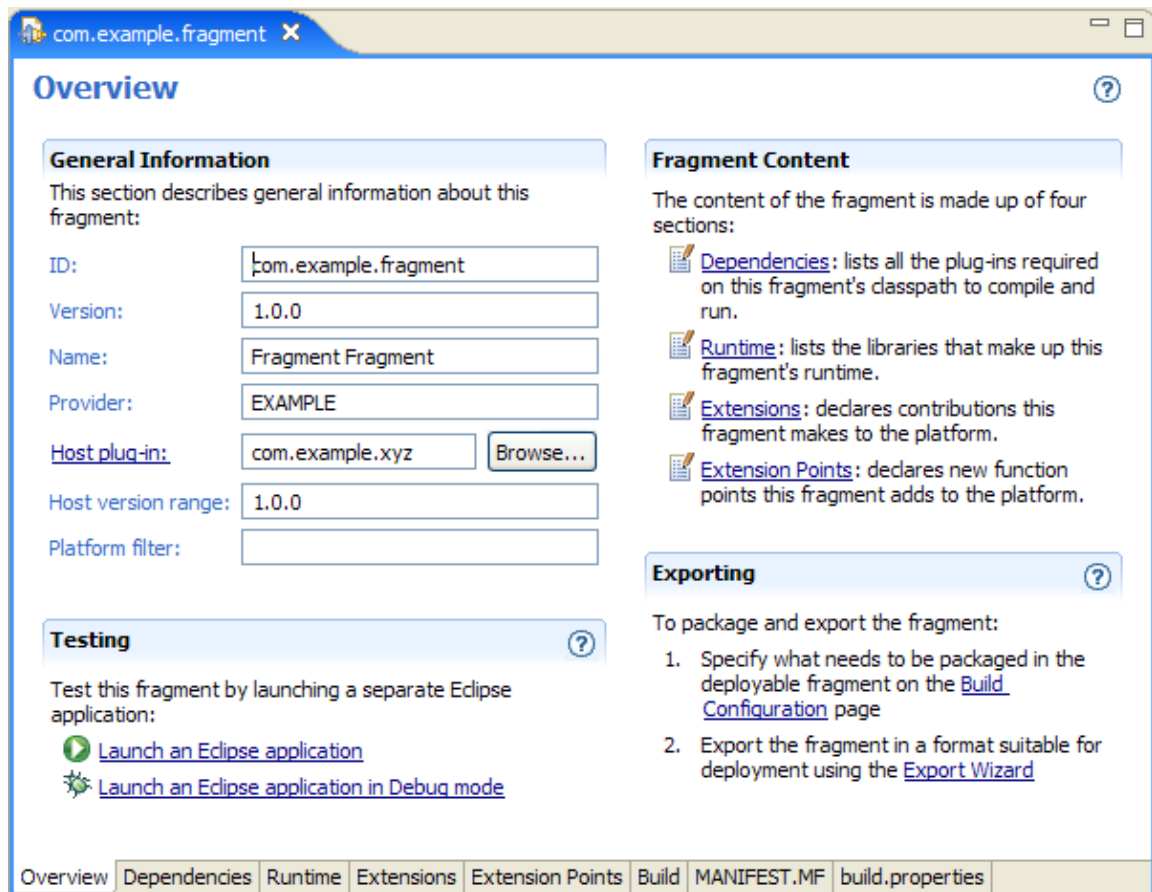
Plug-in ID:

Plug-in Version:

Match Rule:

< Back Next > Finish Cancel

Once the project is created, it opens the fragment manifest editor.



As opposed to a plug-in, a fragment does not have a plug-in class since they follow the life cycle of their host plug-in.

We will add a similar action set as in the plug-in example, but this time in German.

1. Go to the Extensions page in the fragment manifest editor. Press **Add** to launch the Extension wizard.
2. Select "org.eclipse.ui.actionSets" from the list of extension points. Press **Finish**.
3. Select the new action set. Select **New->actionSet** from the popup menu.
4. In the Extension Element Details section, change the **label** property to "Deutsche Aktionsmenge."
5. In the All Extensions section, right-click on the new action set and select **New->menu** from the popup.
6. Change the **label** property of the menu to "Beispiel Menu" and the **id** property to "beispielMenu."
7. Select the menu element again and choose **New->separator** from the popup menu. Change its name to "beispielGruppe" and save it.
8. Create a new "action" element (similar to step 6). Set the **label** property to "Beispiel Aktion." Set the **menubarPath** to "beispielMenu/beispielGruppe."
9. Click on the **class** property hyperlink to generate a new class for your action. Use "com.example.german/src" as your source folder and leave the package name blank (uses the default package). Change the class name to "DeutscheBeispielAktion". Press **Finish**.
10. When the Java editor with the new class opens, find the "run" method and add the following:

```
System.out.println("Hallo, PDE welt!");
```

11. Save and close the Java editor and fragment manifest editor.

Using the Plug-in Development Environment

When you run the fragment using the "Run" tool bar button, the run-time platform instance should have the "Deutsche Aktionsmenge" action set available. (Use **Window->Customize Perspective...->Other** to get to the list of action sets). When you activate the action set, the "Beispiel Menu" menu should appear on the tool bar. When you select its menu item, you should see "Hallo, PDE welt!" in the Console. The runtime platform didn't see the German fragment directly. Instead, its plug-in registry resolved fragment references in such a way that the fragment's action set appeared to the platform as though it came directly from the XYZ Plug-in.

Features

The platform is designed to accept updates and additions to the initial installation. The platform **Update Manager** handles this task by connecting to sites where updates are posted. (See the Workbench User Guide and [Platform Installation and Update](#) for more information about features and the Update Manager.)

You need to package your work in a form that will be accepted by the Update Manager. When you deliver an update to the platform, you are contributing a **feature**.

Features have a manifest that provides basic information about the feature and its content. Content may include plug-ins, fragments and any other files that are important for the feature. The delivery format for a feature is a JAR.

In PDE, your typical development process looks like this:

1. Projects for plug-ins and fragments are created.
2. Code for plug-ins and fragments is created, tested and debugged.
3. When you want to make your code available to others, you create a new feature project.
4. Individual build properties for each plug-in and fragment are tailored to control what files are included and excluded from the packaging.
5. Versions are synchronized with previous versions of the feature, so that the Update Manager will know that a feature is a newer version of an already installed feature.
6. The feature JAR is built.
7. The feature is published on the update server and made available for download.

Setting up a feature project

Similar to plug-ins and fragments, PDE treats platform features as projects. PDE attaches a special "feature" capability to these projects to be able to run nature-specific builders. The project must have a feature manifest.

PDE provides a wizard for setting up a feature project. Typically, you use this wizard to set up a feature once you are done developing plug-ins and fragments. However, you can create the feature at any stage of development and add new plug-ins later.

Example: Setting up a feature for plug-ins and fragments

Assuming that you have followed the previous examples, you should have "XYZ Plug-in" and "German Fragment" in your workspace already. We will create a sample feature and package these artifacts to be ready for delivery.

1. Bring up the feature wizard (**New->Project->Plug-in Development->Feature Project**)

Using the Plug-in Development Environment

2. Set the name of the project to "com.example.feature" and press **Next**.
3. Set the feature name to "Sample Feature" and the feature version to "1.2.2". Set the provider to "Example".
4. In the following page, check the plug-in (XYZ Plug-in) and the fragment (German Fragment).
5. Press **Finish**.

You should now have the "com.example.feature" project in your workspace. The project should have "feature.xml" file and feature manifest editor will open for editing.

Feature manifest editor

The feature manifest editor uses the same concepts seen in the other PDE editors.

The screenshot shows the 'Feature Manifest Editor' for the project 'com.example.feature'. The interface is divided into several sections:

- General Information:** This section describes general information about the feature. It contains input fields for ID (com.example.feature), Version (1.0.0), Name (com.example.feature), Provider (EXAMPLE), Branding Plug-in (with a 'Browse...' button), Update Site URL, and Update Site Name.
- Feature Content:** This section explains the content of the feature, which is made up of five sections:
 - Information:** holds information about this feature, such as description and license.
 - Plug-ins:** lists the plug-ins that make up this feature.
 - Included Features:** lists the features that are included in this feature.
 - Dependencies:** lists other features and plug-ins required by this feature when installed.
 - Installation:** sets advanced installation options, declares an optional install handler and non-plugin data in a feature.
- Exporting:** This section provides instructions on how to export the feature:
 - 1. **Synchronize** versions of contained plug-ins and fragments with their version in the workspace.
 - 2. Specify what needs to be packaged in the feature archive on the **Build Configuration** page.
 - 3. Export the feature in a format suitable for deployment using the **Export Wizard**.
- Publishing:** This section provides instructions on how to publish the feature on an update site:
 - 1. Create an **Update Site Project**.
 - 2. Use the site editor to add the feature to the site, and build the site.
- Supported Environments:** This section allows specifying environment combinations in which the feature can be installed. It includes input fields for Operating Systems, Window Systems, Languages, and Architecture, each with a 'Browse...' button.

At the bottom, there is a tabbed interface with the following tabs: Overview, Information, Plug-ins, Included Features, Dependencies, Installation, Build, feature.xml, and build.properties. The 'Overview' tab is currently selected.

Information that is entered during the feature project setup can be changed on the Overview page. In addition, you can provide an **update site** URL to be used by the Update Manager when searching for new updates.

Branding information for primary features is stored in a branding plug-in. If not explicitly set, Eclipse will assume that the branding plug-in has the same identifier as the feature.

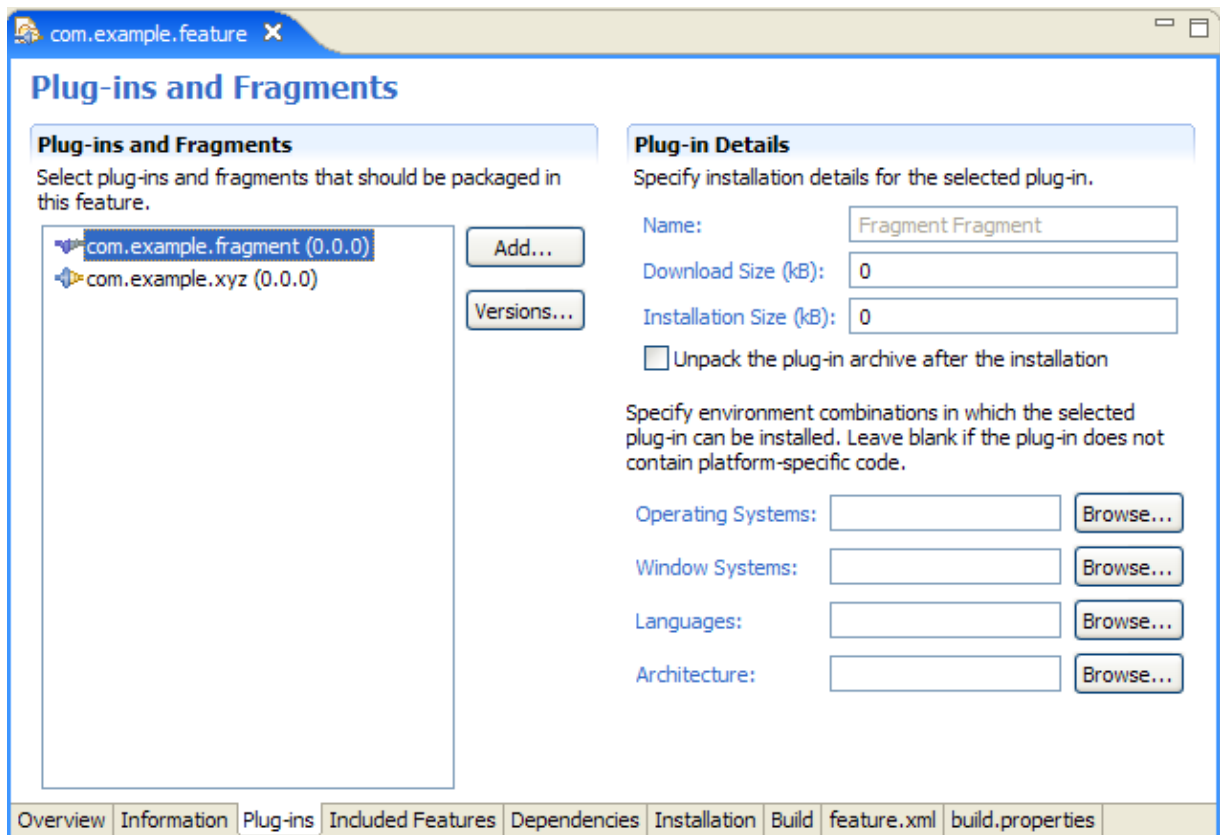
Using the Plug-in Development Environment

By default, your feature is treated as universally portable. You can add constraints by providing supported operating/windowing systems, languages and/or system architectures. This information will be used to ensure that your feature is not installed or shown in the context that does not match these constraints.

The screenshot shows a window titled 'com.example.feature' with a tab labeled 'Information'. The main content area has a heading 'Information' and a subheading 'Enter description, license and copyright information. Optionally, provide links to update sites for installing additional features.' Below this is a tabbed interface with four tabs: 'Feature Description' (selected), 'Copyright Notice', 'License Agreement', and 'Sites to Visit'. Under the 'Feature Description' tab, there is an 'Optional URL:' field containing 'http://www.example.com/description' and a 'Text:' label next to a large text area containing '[Enter Feature Description here.]'. At the bottom of the window is a navigation bar with tabs: 'Overview', 'Information' (selected), 'Plug-ins', 'Included Features', 'Dependencies', 'Installation', 'Build', 'feature.xml', and a button with a double quote and '1'.

Features are required to provide description, license and copyright information. This information can be edited on the Information page. Each of these three categories can be represented as either text or a URL that points to a valid HTML page. Although URL can be absolute, HTML pages are typically provided with the feature and URLs are relative to the project root.

The *Sites To Visit* tab lists URLs that are used to point users to other interesting features and/or sites.



The plug-ins and fragments to be packaged in this feature are listed on the **Plug-ins** page. If a plug-in and/or fragment contains platform-specific code, then environment conditions should be specified and associated with that plug-in or fragment.

Included Features

Create a composite feature by including references to other features.

org.eclipse.rcp (0.0.0) Add...

Included Feature Details

Specify a name of the included feature, displayed when the feature is not installed. Indicate if the included feature is optional.

Feature Name:

☐ The feature is optional

When searching for patches to this included feature, contact

☒ the update site for the parent feature

☐ the update site for the included feature

☐ both

Specify environment combinations in which the included feature can be installed. Leave blank if the feature does not contain platform-specific code.

Operating Systems: Browse...

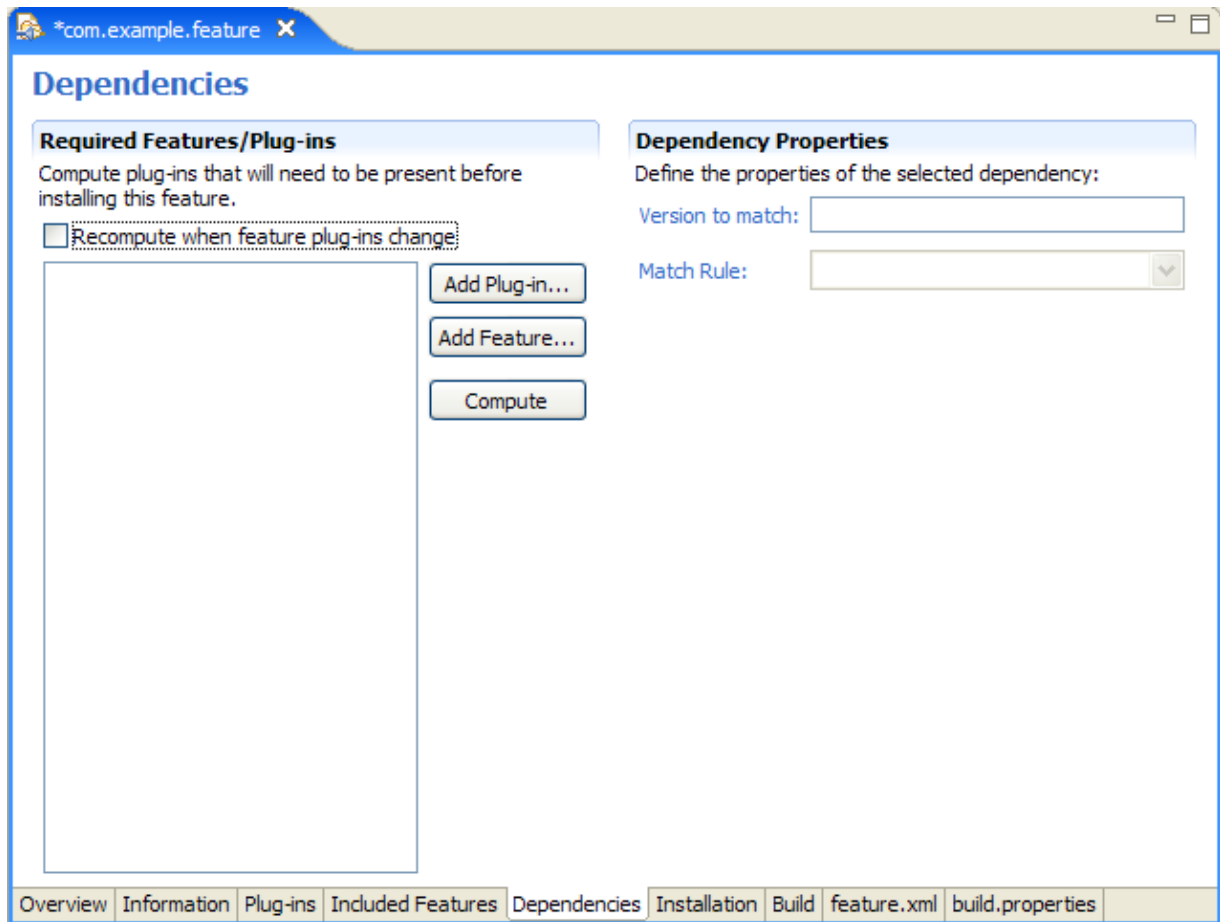
Window Systems: Browse...

Languages: Browse...

Architecture: Browse...

Overview Information Plug-ins **Included Features** Dependencies Installation Build feature.xml build.properties

A feature may include other features, thus creating a feature hierarchy. When the feature is built, all included features are recursively built and packaged.



This section lists all features and plug-ins that must be present in the product before the Update Manager installs this feature. If any of these pre-requisites is missing, the feature will not be installed. The requirement can be based solely on plug-in IDs, or further constrained using expected versions and match rules.

Installation Details

Installation Options
Specify if the feature cannot be installed simultaneously with other features, or must be installed in the same directory as another feature.

☐ This feature requires exclusive installation.

To colocate this feature with another feature, specify the reference feature ID.

Feature ID:

Install Handler
Specify an optional install handler that will be called during the installation.

Library:

Handler:

Feature Data
Select non-plug-in data archives that should be packaged in this feature.

Data Archive Details
Specify size (in kB) for the non-plug-in data archives:

Download Size:

Installation Size:

Specify environment combinations in which the selected archive can be installed. Leave blank if the archive does not contain platform-specific code.

Operating Systems:

Window Systems:

Languages:

Architecture:

Overview | Information | Plug-ins | Included Features | Dependencies | **Installation** | Build | feature.xml | build.properties

In addition to plug-ins, opaque data entries can be specified to carry custom feature information. These entries usually come together with custom install handlers. Install handlers can be used to perform non-standard install tasks and manipulate data entries once they are downloaded by the Update Manager. You can read more about this and other feature issues in Platform Install and Update guide.

Synchronizing versions

Automatic synchronization at build time (Recommended)

If the plug-in version changes frequently and/or developers do not have access to the feature, the versions of plug-ins, fragments and included features can be set to the special value **0.0.0** that will be replaced when exporting the feature. This is especially convenient when plug-in version are automatically upgraded using the qualifier tag.

UI driven synchronization

The versions of plug-ins and fragments should be synchronized with the version of the packaged feature so that you can manage plug-in, fragment, and feature versions consistently. Developers typically ignore individual manifest versions until it is time to deploy their features. The Update Manager uses feature versions to determine whether a plug-in is older or newer than one already installed. Plug-ins and fragments

Using the Plug-in Development Environment

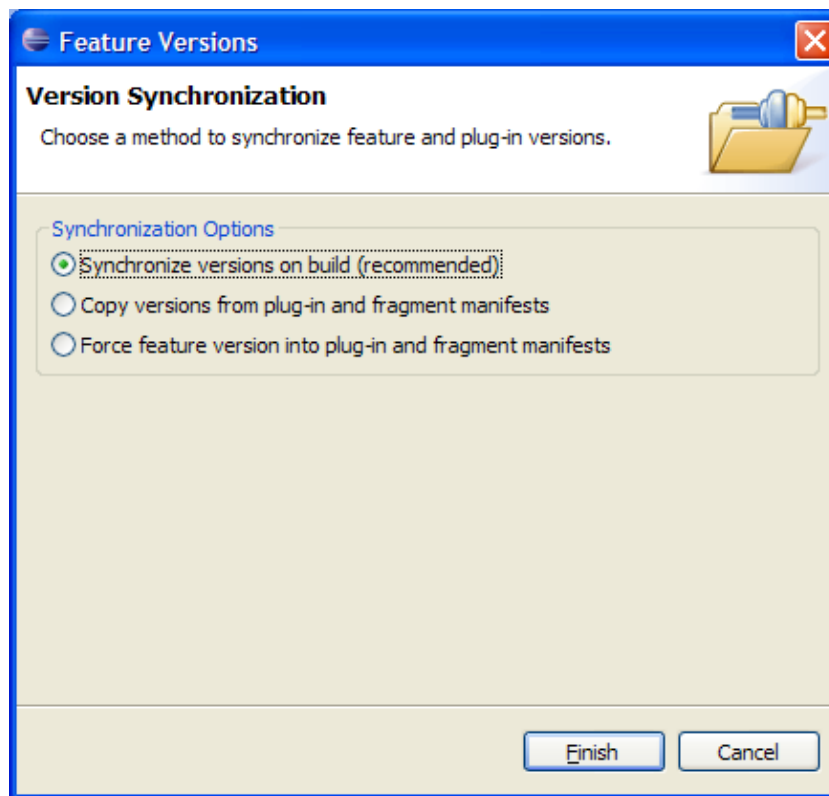
need to follow the same version number conventions so there is no confusion about which plug-in version belongs to which feature version.

The most convenient way to synchronize versions is to pick the version of the feature and force it into all the plug-ins and fragments that the feature references. This operation updates manifest files, so you are required to close all the manifest editors before proceeding.

Example: Synchronizing versions in the feature editor

We will take our ongoing example and force the feature version (1.2.2) into "XYZ Plug-in" and "German Fragment."

1. Open the component manifest editor.
2. Select **Synchronize versions...** from the popup menu. A wizard will open.
3. Select the first radio button ("Force feature version..."). Press **Finish**.
4. Switch to the Content page verify that the versions are now 1.2.2.
5. Double-click on "XYZ Plug-in" and "German Fragment" objects and verify versions in their corresponding manifest editors.



Exporting a Feature

First you have to set up the build configuration. The build configuration includes information about the files and directories that should be included in the feature for each individual plug-in and fragment. There may be some design-time files and directories that should not be shipped. See [Build Configuration](#) for details.

Using the Plug-in Development Environment

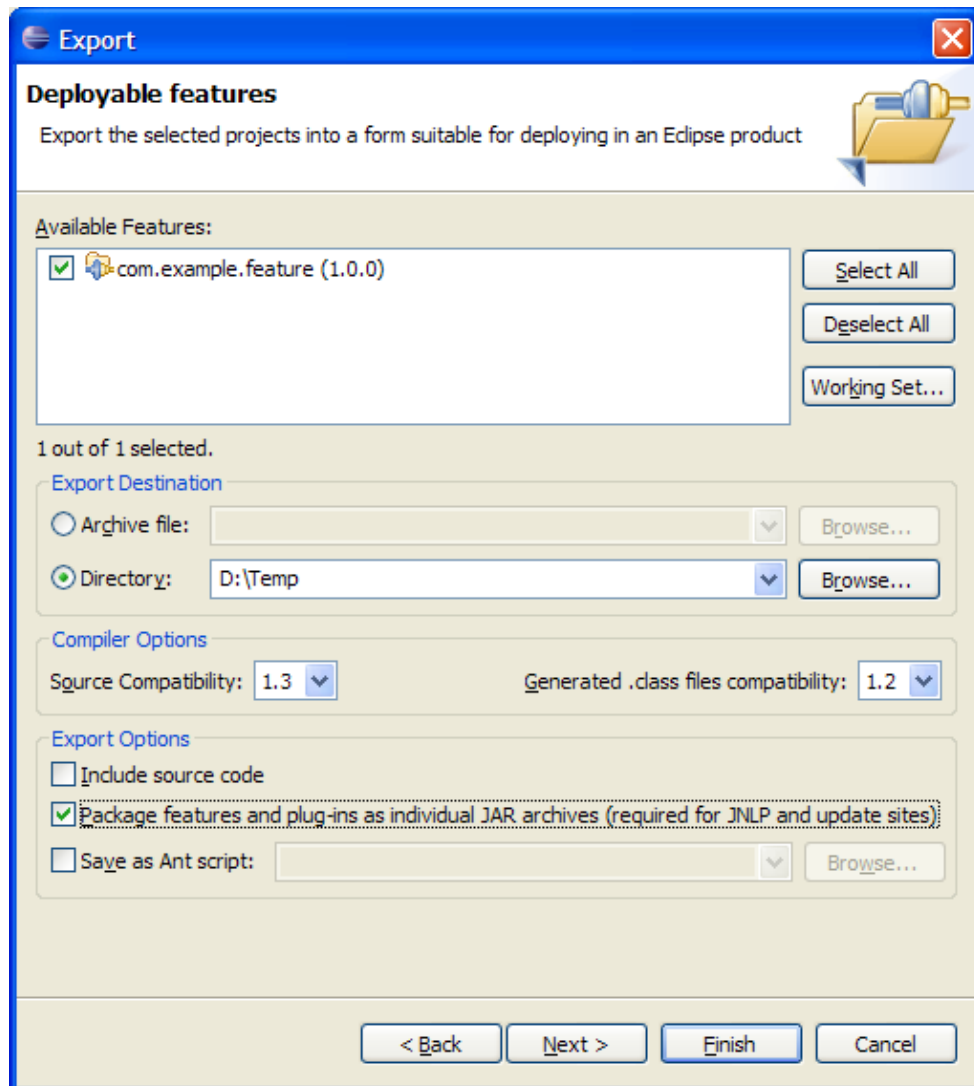
Then you can use PDE's Export Deployable Features wizard to build and export the feature. This way you are shielded from Ant scripts and your workspace is never polluted with build by-products.

Select **File > Export... > Deployable Features**.

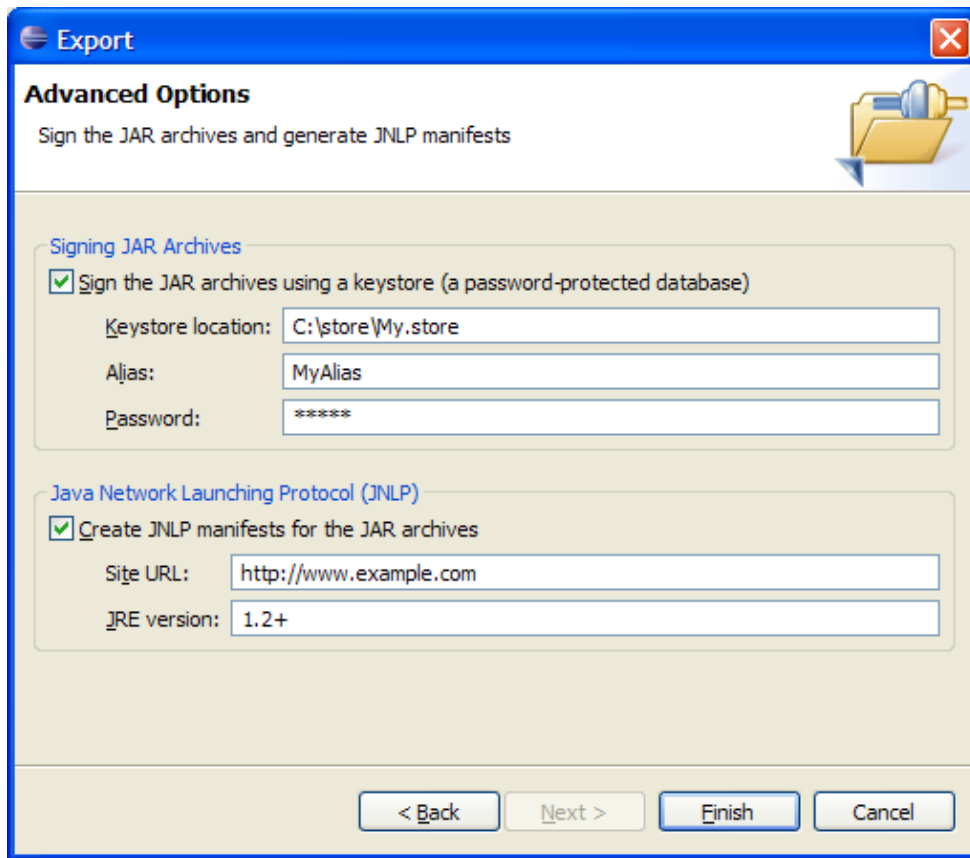
Select the feature(s) you want to export.

A feature can be exported to an archive or a directory.

To publish a feature to an update site, the feature and its includes plug-ins must be packaged as individual JAR archives.



Press *Next*.



You have the option to sign the JARs for added security. You also have the option to generate JNLP manifests for [Java Web Start deployment](#).

The wizard will build all the features selected and their included plug-ins and fragments. Features go into a features/ subdirectory and plug-ins go into a plugins/ subdirectory.

The alternative would be to build features manually:

1. Right-click on the feature.xml file for your feature project and choose 'Create Ant Build File'. This will generate a build.xml file.
2. Select the build.xml file and choose 'Run Ant...'
3. In the Ant build script wizard, select the target(s) you want to run. See [Generating Ant Scripts](#) for details.

Build configuration

The build mechanism is driven by a build configuration. The build configuration for an individual plug-in, fragment, or feature is found in a **build.properties** file for the corresponding element.

PDE project creation wizards generate the **build.properties** file when plug-in projects are created. The file contains information on how to compile source folders into JARs. This information can be indirectly updated in the Runtime page of the manifest editor. It can also be directly modified using the appropriate editor.

Using the Plug-in Development Environment

PDE provides a simple editor for the **build.properties** that has form and source views. The file itself follows the Java properties format. You need to provide a number of keys and their corresponding values. Multiple values are separated using a comma as the delimiter.

Common properties

- **bin.includes** – lists files that will be included in the binary version of the plug-in being built;
- **bin.excludes** – lists files to exclude from the binary build;
- **qualifier** – when the element version number ends with **.qualifier** this indicates by which value ".qualifier" must be replaced. The value of the property can either be **context**, **<value>** or **none**. Context will generate a date according to the system date, or use the CVS tags when the build is automated. Value is an actual value. None will remove ".qualifier". If the property is omitted, context is used.
- **custom=true** – indicates that the build script is hand-crafted as opposed to automatically generated. Therefore no other value is consulted.

Plug-in specific properties

- **source.<library>** – lists source folders that will be compiled (e.g. **source.xyz.jar=src/, src-ant/**). If the library is specified in your plugin.xml or manifest.mf, the value should match it;
- **output.<library>** – lists the output folder receiving the result of the compilation;
- **source.<library>** – lists the files that should not be copied by in the library by the compiler;
- **extra.<library>** – extra classpaths used to perform automated build. Classpath can either be relative paths, or platform urls referring to plug-ins and fragments of your development environment (e.g. **../someplugin/xyz.jar, platform:/plugins/org.apache.ant/ant.jar**). Platform urls are recommended over relative paths;
- **manifest.<library>** – indicate the file that will be used as a manifest for the library. The file must be located in one of the source folder being used as input of the jar.
- **src.includes** – lists files to include in the source build;
- **src.excludes** – lists files to exclude from the source build;
- **jars.extra.classpath** – (**deprecated**) same effect than extra.<library> except that the entries are applied to all libraries;
- **jars.compile.order** – defines the order in which jars should be compiled (in case there are multiple libraries).

The values defined for these keys ending with "includes" or "excludes" are expressed as Ant "patterns". Standard formats give the expected results. For example, **"*.jar"** indicates all jar files in the top level directory. The patterns are not deep by default. If you want to describe all Java files for example, you should use the pattern **"**/*.java"**. The pattern **"**"** matches any number of directory levels. Similarly, to describe whole sub-trees, use **"xyz/"**.

Feature specific properties

- **root** – list the files and folders that must be included in the root of the product. The different values supported are:
 - ◆ `<folderName>` – a relative path to a folder to be copied;
 - ◆ `file:<fileName>` – a relative path to a file to be copied;
 - ◆ `absolute:<folderName>` – an absolute path to a folder to be copied;
 - ◆ `absolute:file:<fileName>` – an absolute path to a file to be copied;
- **root.<config>** – list the files and folders that must be included in the root of the product when it is built for the specified configuration. config is composed of the three (3) segments of a configuration separated with a dot;
- **root.permissions.<permissionValue>** – list the files and folders to chmod to the given value. Reference to folders must ends with a '/';
- **root.permissions.<config>.<permissionValue>** – list the files and folders to chmod to the given value for a specific configuration. Reference to folders must ends with '/';
- **root.link** – list by pairs (separated by a comma) the files and folders that need to be symbolically linked. The first entry indicate the source (target in the unix terminology) and the second entry the link name;
- **root.link.<config>** – a comma separated list of pairs of files and folders that need to be symbolically linked for a specific configuration. The first entry indicate the source (target in the unix terminology) and the second entry the link name;
- **generate.feature@<featureId>** – indicates that the source feature **featureId** will be the source feature for the feature indicated as value of this property. The values listed after the first comma indicates elements to be fetched from the repository;
- **generate.plugin@<pluginId>** – indicates that the source plug-in **pluginId** will be the source plug-in for the indicated as value of this property.

The following example has been extracted from the build.properties of the org.eclipse.platform feature.

```
bin.includes=ep1-v10.html,eclipse_update_120.jpg,feature.xml,feature.properties,license.html

root=rootfiles,file:../../plugins/org.eclipse.platform/startup.jar,configuration/
root.permissions.755=eclipse

root.linux.motif.x86=../../plugins/platform-launcher/bin/linux/motif,linux.motif
root.linux.motif.x86.link=libXm.so.2.1,libXm.so.2,libXm.so.2.1,libXm.so
root.linux.motif.x86.permissions.755=*.so*
```

Generating Ant scripts from the command line

Ant scripts are typically generated using the Plug-in Development Environment (PDE), but it is also possible to generate them by hand or from other scripts.

Indeed PDE exposes Ant tasks to generate the various build scripts. Build script generation facilities reside in the following tasks. Arguments are also listed for each task.

Using the Plug-in Development Environment

- **eclipse.fetch:** generates an Ant script that fetches content from a CVS repository. The eclipse fetch is driven by a file whose format is described below (see [Directory file format](#)).

elements : the entry that will be fetched. The format expected is of the form type@id as specified in the directory file format;

buildDirectory : the directory into which fetch scripts will be generated and into which features and plug-in projects will be checked out;

directory : the path to a directory file;

children : optional, specifies whether the script generation for contained plug-ins and fragments should be invoked. Default is set to true;

cvspassfile : optional, the name of a CVS password file;

fetchTag : optional, overrides the tag provided in directory file by the given value;

configInfo : optional, an ampersand separated list of configuration indicating the targeted configuration. The default is set to be platform independent;

recursiveGeneration : optional, specify whether or not fetch scripts should be generated for nested features. The default is set to true.

- **eclipse.buildScript:** generates a build.xml file for the given elements.

elements : the entry to be fetched from the repository. Entry is expected to be of the form type@id as specified in the directory file format;

buildDirectory : the directory where the features and plug-ins to build are located;

children : optional, specifies whether the script generation for contained plug-ins and fragments should be invoked. Default is set to true;

recursiveGeneration : optional, specified whether the script generation for contained features should be invoked. Default is set to true;

devEntries : optional, a comma separated list of directories to be given to the compile classpath;

buildingOSGi : optional, indicates if the target is 3.x. or 2.1;

baseLocation : optional, indicates a folder which contains installed features and folders;

configInfo : optional, an ampersand separated list of configuration indicates the targeted configuration. The default is set to be platform independent;

pluginPath : optional, a comma separated list of URLs pointing to installed plug-ins. If specified, this list must include the whole list of plug-ins to be compiled;

archivesFormat : optional, an ampersand separated list of configs and the expected output format for each of those. The format is separated by a dash (–) from the configuration. The values supported are: folder, tar, zip, antZip, respectively meaning don't archive, use tar to create the archive, use the

Using the Plug-in Development Environment

version of info zip available on the platform, use ant zip . The default value is antZip.

product : optional, '/' separated path to the location of an RCP product being built. The first segment of the path must refer to the plug-in id of a plug-in containing the .product file.

signJars : optional, indicates if the scripts generated must sign jars for features and plug-ins. The default value is false. The parameters to the sign task are controlled by the following ant properties: sign.alias, sign.keystore and sign.storepass respectively being passed to the alias, keystore and storepass parameters from the ant signJar task. The default value is false.

generateJnlp : optional, indicates if a jnlp file should be generated for all the features being built.

outputUpdateJars : optional, generates plug-ins and features in the update site format when set. The default value is false. Note that the site.xml is not generated nor updated.

forceContextQualifier : optional, uses the given value to replace the .qualifier being by plug-ins and features.

Examples

```
<eclipse.fetch elements="bundle@org.eclipse.osgi"
    buildDirectory="c:\toBuild"
    directory="directory.txt"
    configInfo="win32,win32,x86 & linux, motif, x86"
/>

<eclipse.buildScript elements="bundle@org.eclipse.osgi"
    buildDirectory="c:\toBuild"
    archivesFormat="macosx, carbon, ppc - tar"/>
```

Directory file format

Directory files are used to indicate where the plug-ins and features are located, as well as indicating which version should be fetched. It is a Java property file whose line format is "type@id=version, repositoryLocation, password,path".

- *type*: a string describing the type of the element. It must be one of the following: plugin, fragment, feature, bundle;
- *id*: the name of the CVS module where the element is located. Note that the feature/plugin/fragment.xml must be in the root of this module;
- *version*: an existing version tag in the repository;
- *repositoryLocation*: a CVS repository location;
- *password*: optional, a password to connect to this repository;
- *path*: optional, the cvs module name and path to the element manifest.

Example of a directory file

```
plugin@org.eclipse.pde.build=v20040622,:pserver:anonymous@dev.eclipse.org:/home/
feature@org.eclipse.pde.builder=v20040622,:pserver:anonymous@dev.eclipse.org:/h
plugin@org.eclipse.osgi=v20040617a,:pserver:anonymous@dev.eclipse.org:/home/ecl
```

Using the targets

The tasks previously described only work if Eclipse is running. In the particular scenario of executing Ant scripts using Eclipse tasks, the scripts must be run using the Eclipse Ant Runner application. The command line for this particular case is the following:

```
java -cp startup.jar org.eclipse.core.launcher.Main -application org.eclipse.ant.core.antRunner
```

Note that the parameters appearing after the application are the parameters that are passed to Ant.

Working with update sites

Eclipse is capable of installing or updating features placed on the remote servers. The features and plug-ins must be packaged in JARs and have a manifest (site.xml) file that links them together. These files collectively form an Eclipse **update site**.

PDE provides support for building update sites directly in the workspace. Normally, update sites are placed on remote HTTP servers, but sites in the local file systems are also valid and can be viewed in update manager. PDE uses this property to provide for building and previewing update sites directly in the workspace.

Setting up an update site project

An update site project is represented in the workspace with a project that has a site.xml file at its root.

PDE provides a wizard for setting up an update site project. Typically, you use this wizard to set up an update site once you are done developing plug-ins, fragments and features. However, you can create the site project at any stage of development.

By default, an update site project is created local to your workspace. Since you may want to host multiple versions of your plug-ins and features in this site, you may want to keep the update site outside the workspace location. In that case, change the location of the update site project during creation. If the location you specified already contains the files the wizard is about to create, it will keep them instead. This will allow you to create this project in more than one workspace and point at the same shared location.

Example: Setting up an update site project

Assuming that you have followed the previous examples, you should have "XYZ Plug-in" and "German Fragment" in your workspace already, as well as "Sample Feature" feature project. We will create an update site that can serve "Sample Feature" to the update manager.

1. Bring up the update site wizard via **New > Project > Plug-in Development > Update Site Project**.
2. Set the name of the project to "Update Site".
3. Set the update location to a non-default location.
4. Select the **Generate a web page** listing option. This will help advertise your features and plug-ins to the world. Click **Finish**.

New Update Site

Update Site Project
Create a new update site project

Project name:

Project contents

☐ Use default

Directory:

Web Resources

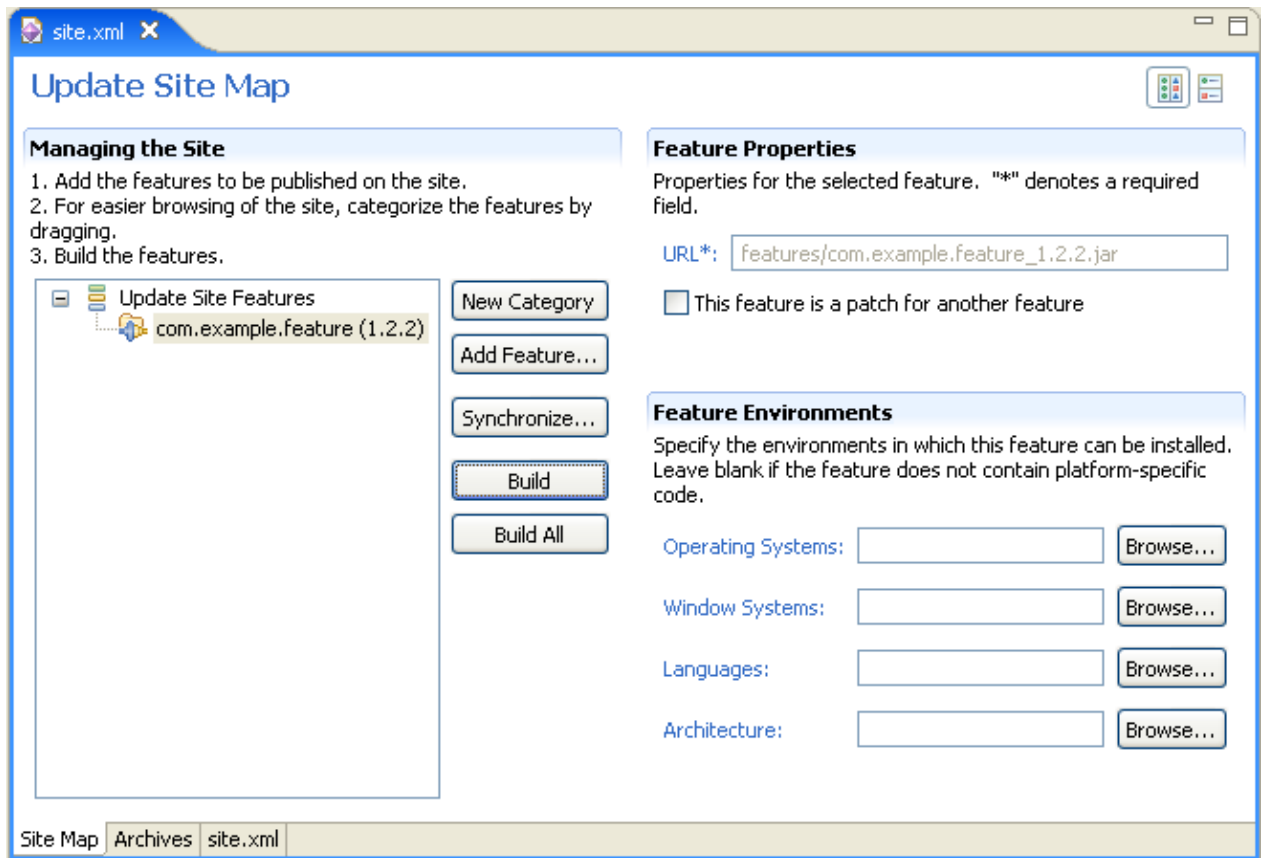
☒ Generate a web page listing all available features within the site

Web resources location:

< Back Next > **Finish** Cancel

Building plug-ins, fragments and features using update site editor

Building an update site is a relatively simple task and most of the work is done on the **Features** page of the update site editor.



Features added to the **Managing the Site** section will be built recursively when you press **Build All**. This means that the features and all the plug-ins and fragments they include will be built in one batch operation. The feature JARs end up in the **features/** folder of your site project and the plug-in JARs are placed in the **plugins/** folder of your site project.

If you would like to build one feature, select a feature and press the **Build** button..

If you are dealing with features that include other features, only the root feature needs to be listed, since its child features will automatically be built.

Features that have environment constraints need to specify environments in the **Feature Environments** section. Select a feature and fill in the values, or press **Synchronize...** button and let the environment be copied from feature manifest to the site.

For easy browsing of your features on your update site, you can create categories and organize your published features in these categories. A feature may appear in ≥ 0 categories.

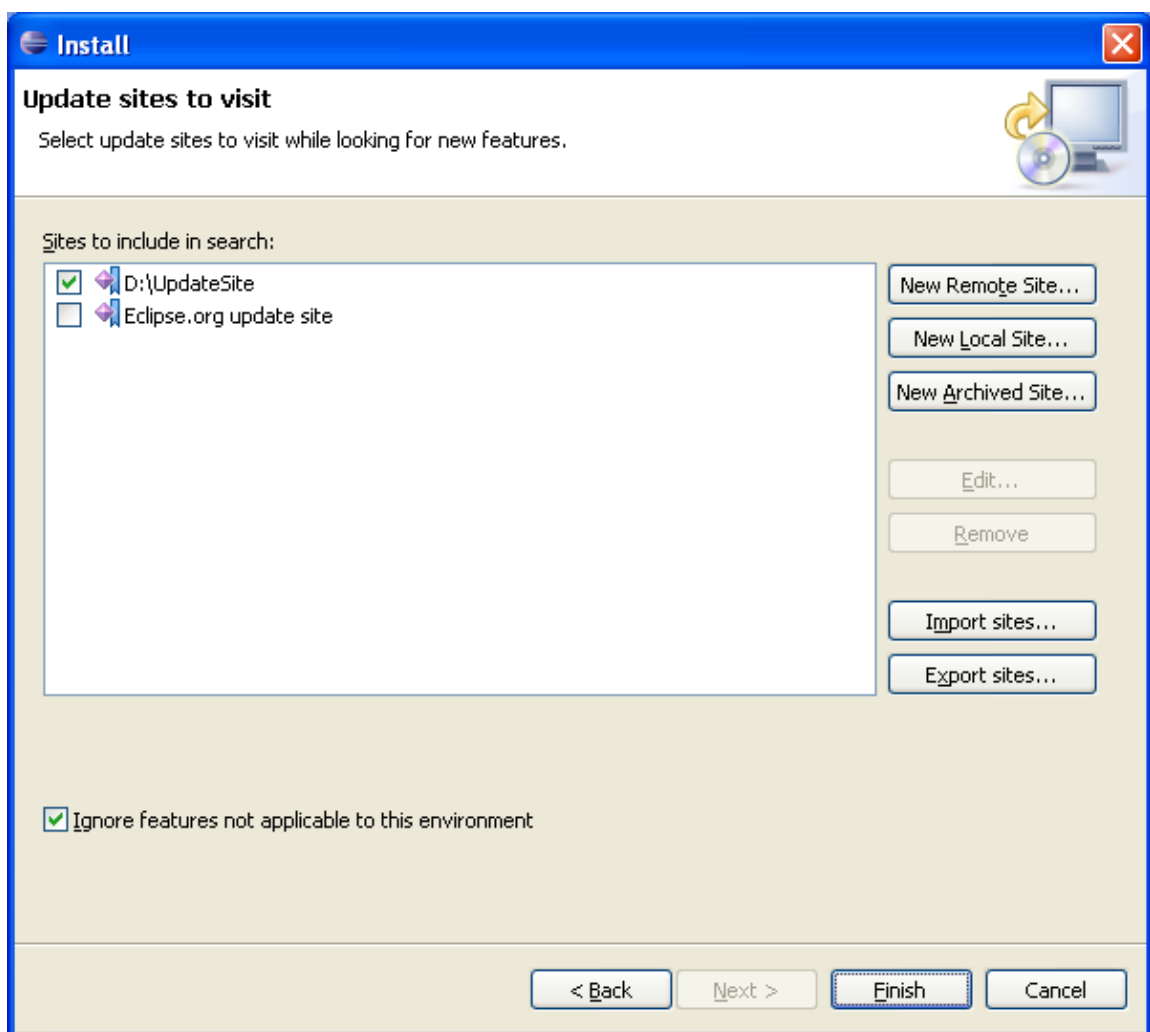
To preview on what your web site would look like, save the site.xml file and open the index.html file at the root of your site project in a browser.

Previewing an update site

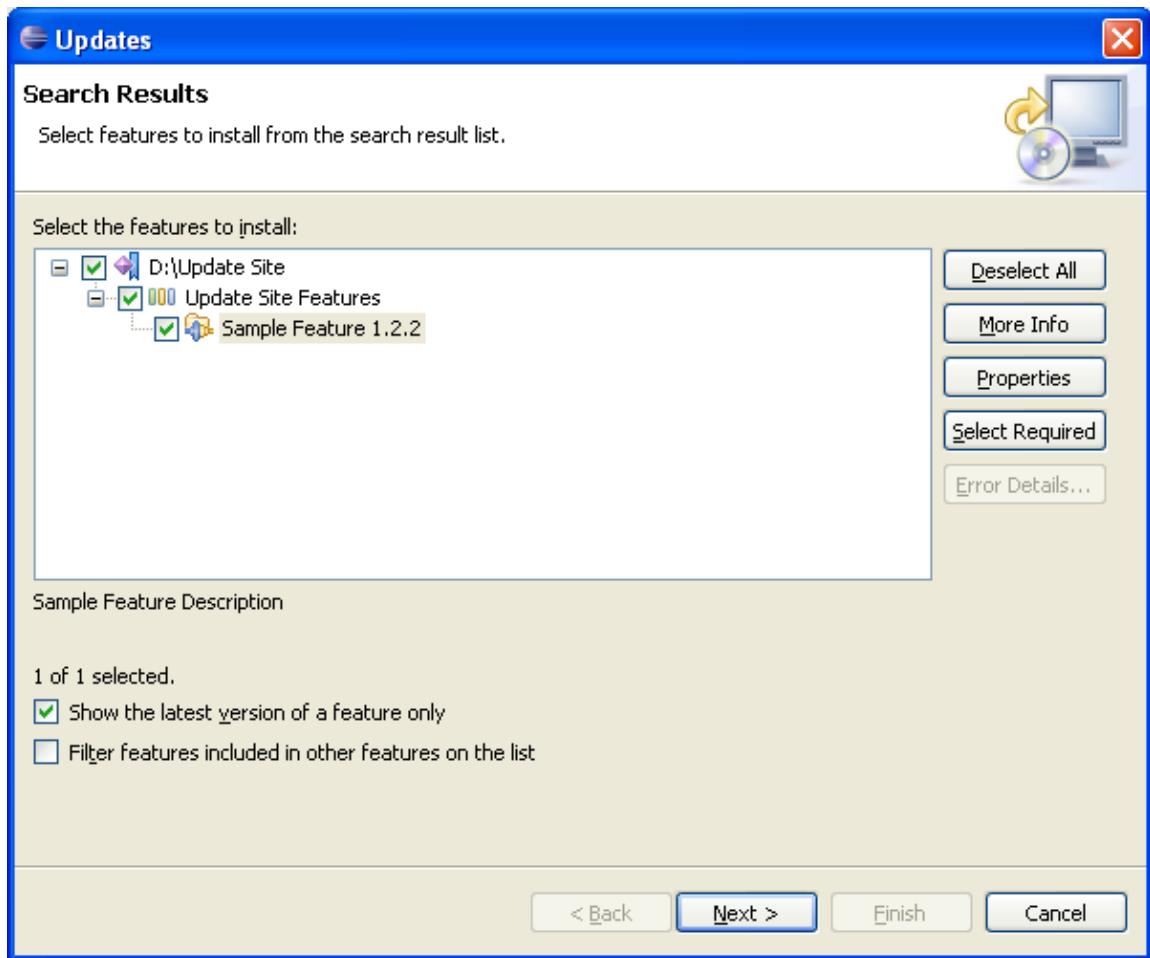
After you have built your feature using the update site editor and all the archives are in the right places, you can preview it in-place using the Update Manager.

Example: building and previewing the sample update site

1. Open the New Install Wizard via **Help > Software Updates > Find and Install...**
2. Select **Search for new features to install**.
3. On the **Update sites to visit** page, use the **New Local Site...** button to locate the update site you have created. Select the new site and click **Finish**.



4. Drill down, and verify that the feature you created has been located by the Update Manager.



5. Select the feature, and click **Next**. Verify the license information. Proceeding beyond that point in the Install Wizard will go through the steps of the actual installation, which is not the focus here.

Create an RCP Template

The first step in creating a fully-branded RCP product is to create a plug-in project with an RCP template

Select **File > New... > Project > Plug-in Project**. Click **Next**.

Using the Plug-in Development Environment

New Plug-in Project

Plug-in Project
Create a new plug-in project

Project name:

Project contents
☒ Use default
Directory:

Project Settings
☒ Create a Java project
Source Folder Name:
Output Folder Name:

Plug-in Format
What version of Eclipse is this plug-in targeted for?
☒ Create an OSGi bundle manifest

Enter *com.example.xyz* for the name of the project. Click *Next*.

Using the Plug-in Development Environment

New Plug-in Project

Plug-in Content
Enter the data required to generate the plug-in.

Plug-in Properties

Plug-in ID:

Plug-in Version:

Plug-in Name:

Plug-in Provider:

Classpath:

Plug-in Class

☒ Generate the Java class that controls the plug-in's life cycle

Class Name:

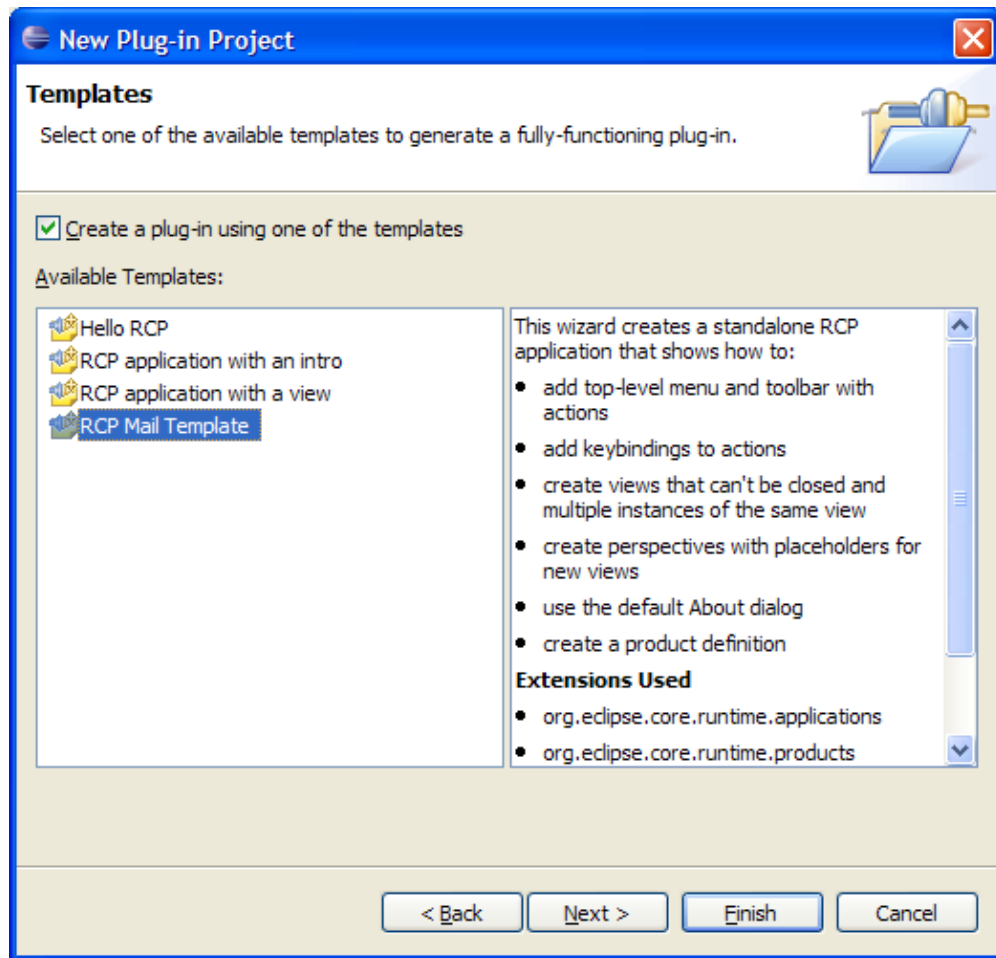
☒ This plug-in will make contributions to the UI

Rich Client Application

Would you like to create a rich client application? ☒ Yes ☐ No

< Back Next > Finish Cancel

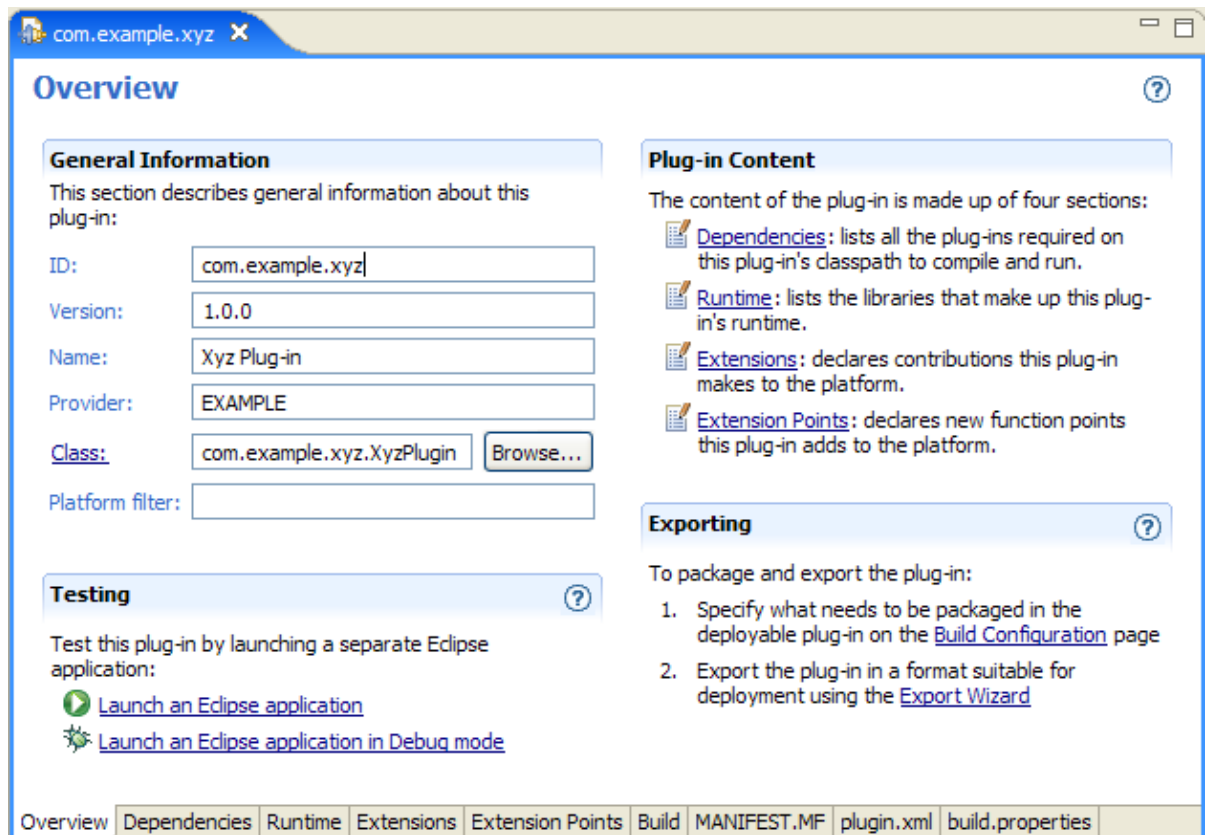
Choose *Yes* in the Rich Client Application section. Click *Next*.



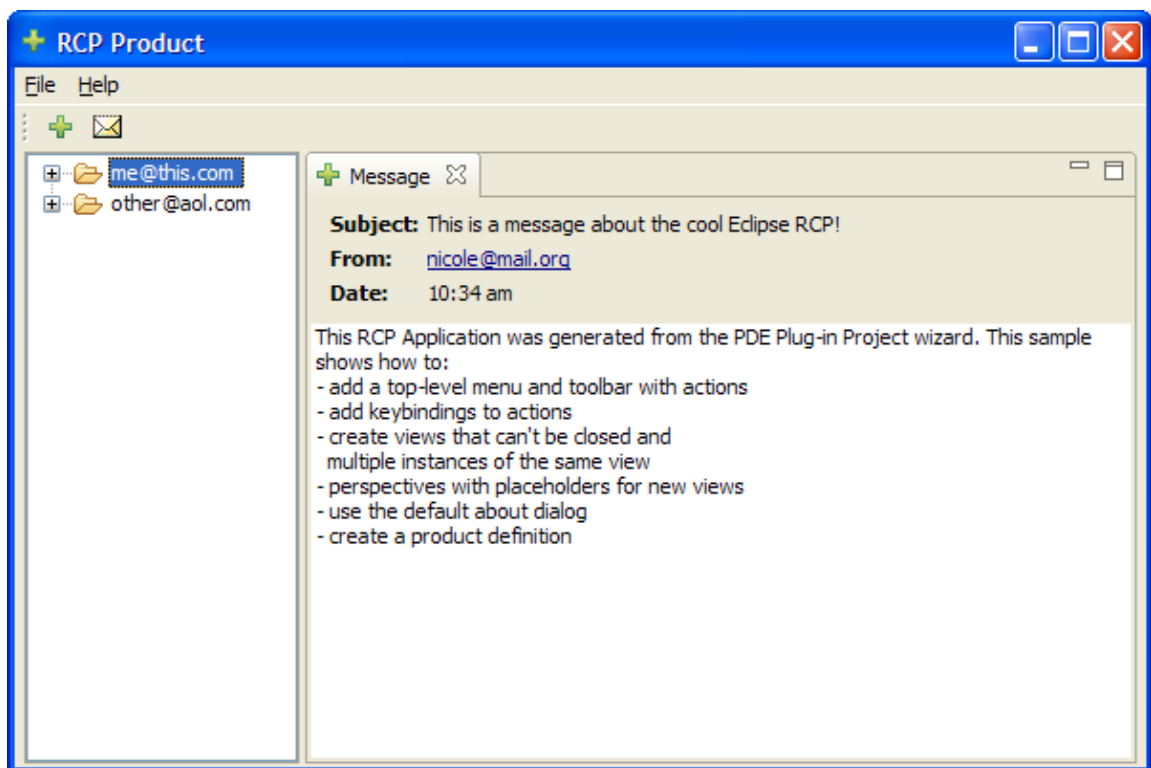
Press ***Finish***.

A new plug-in project will be created and the manifest editor will open.

Using the Plug-in Development Environment



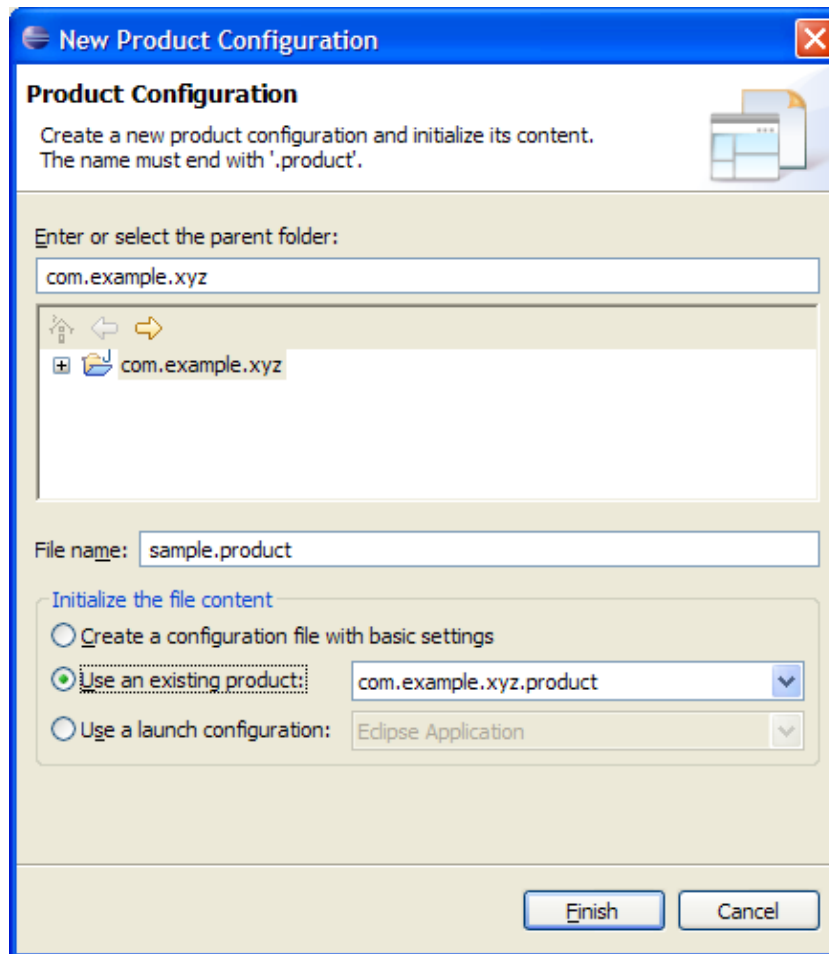
Click on the *Launch an Eclipse application* link in the **Testing** section. A splash screen will appear and go away. The fully-branded RCP mail application subsequently appears.



Product Configuration

The RCP product export story is based on a .product file. This file contains all the data necessary to build a plugin-based or a feature-based product. Note that this file is a development-time artefact used by PDE only. It is NOT read or interpreted by the runtime and must/should not be included in the product being built.

To create a product configuration, select **File > New... > File > Product Configuration**.



The only restriction is that the file extension be *.product*. Otherwise, the product configuration could be placed in any folder, in any project in the workspace.

To initialize the configuration, you have three options:

1. Create a configuration file with basic settings: This option should be chosen when the user does not even know what a product is or if the user has not yet declared a product extension in his plug-in.
2. Use an existing product to initialize the file: This option should be chosen if you already have a plugin declaring a product extension. With this option, PDE will pre-populate the product configuration with as much data as possible given your product declaration.
3. Use a launch configuration: This option should be chosen if you have been testing your plug-in with an Eclipse application launch configuration that is filled with custom settings, and you want all those

Using the Plug-in Development Environment

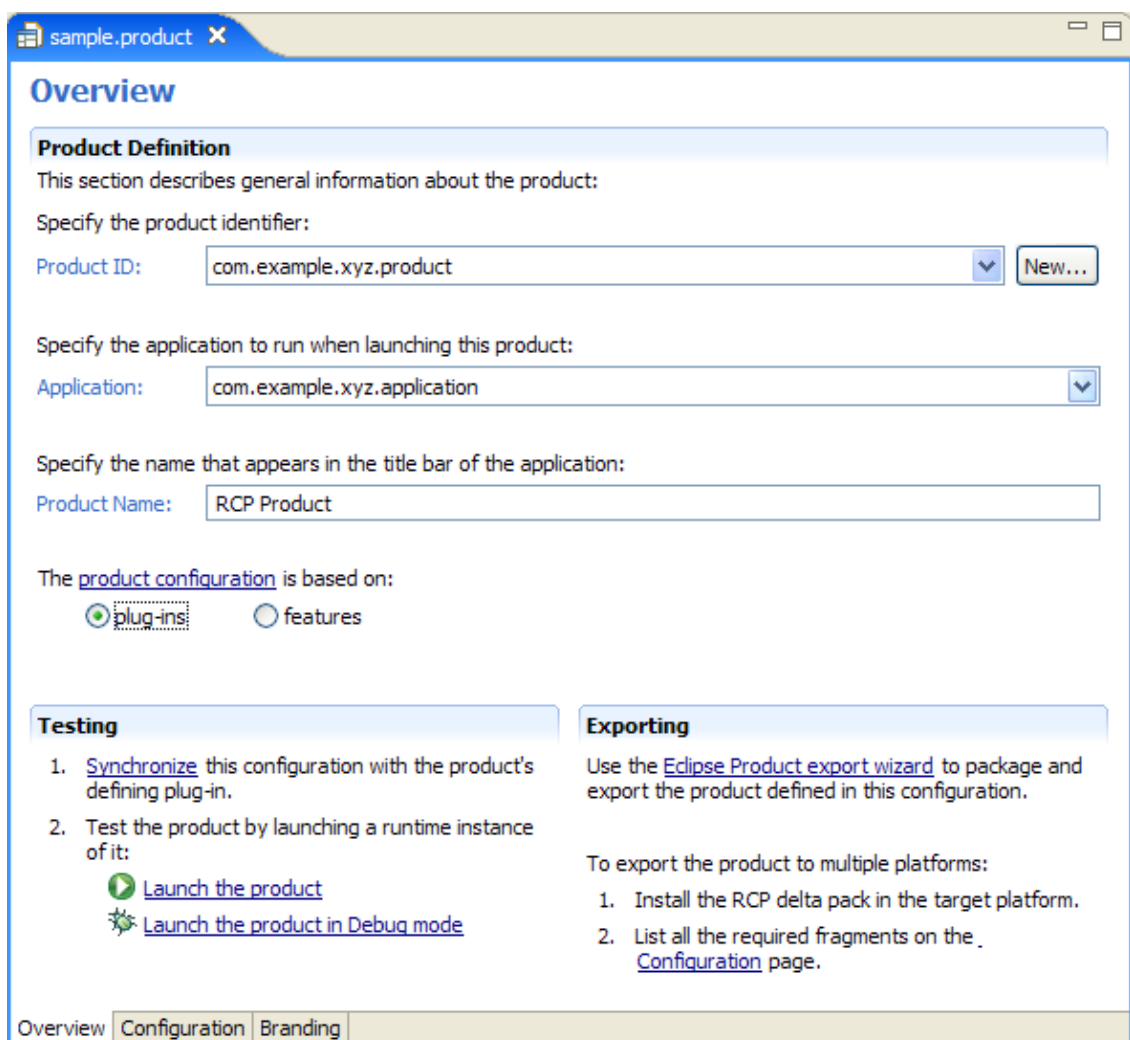
settings to get carried over to the product configuration.

In our RCP mail example, we already have a product defined: *com.example.xyz.product*. So the second option is best. Select it and press **Finish**.

Product Editor

The product editor is the one-stop shop where you can define all aspects of your product from basic definition to branding.

Because we just created a product configuration based on the RCP template, most of the product information has already been filled out by PDE based on the plugin.xml of the *com.example.xyz* plug-in.



The Product Definition section is where the product ID and name are specified, as well as the application that will run when the product is launched.

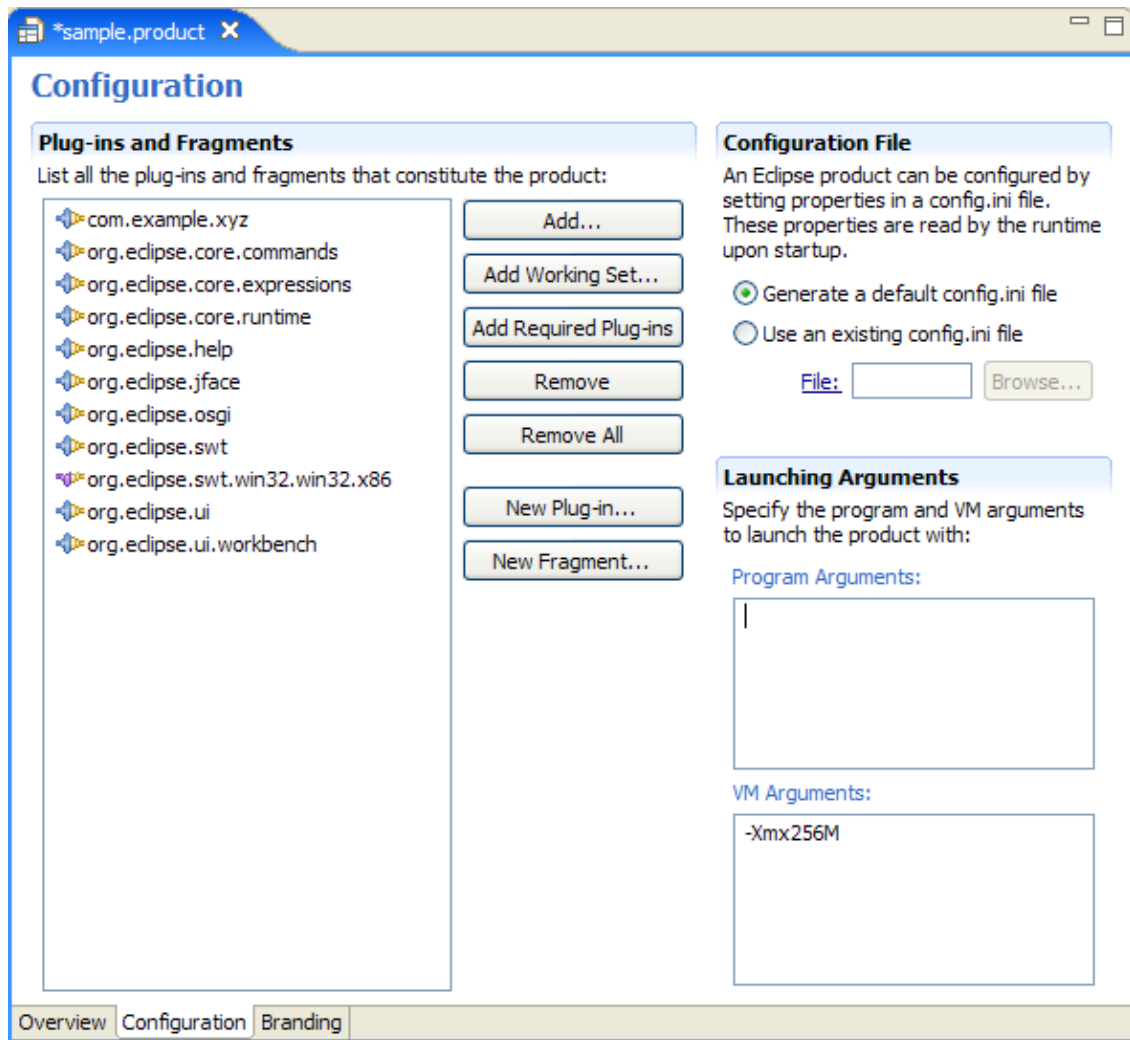
In this example, we will create a plugin-based product.

Using the Plug-in Development Environment

The **Testing** and **Exporting** sections should look familiar to the user as they are very similar to sections on the **Overview** page of the plug-in editor.

A very noticeable difference however is the first step in the Testing section: *Synchronize this configuration with the product's defining plug-in*. What does this mean?

Remember that the product configuration is for PDE use only and is not interpreted nor read by the runtime. Some of the data that enter in this file (e.g. product ID, application, window images, about image, etc.) must be copied to the plugin.xml file of your branding plug-in for these changes to take **real** effect at runtime. That's what the synchronize action does. It makes sure your plugin.xml contains up-to-date data and in sync with the product configuration.



The Configuration page is where you list all the plug-ins and fragments that constitute the product.

A configuration file is a property file containing system properties read by the runtime upon startup. It is recommended to let PDE generate a default config.ini file for you.

You can also specify the program arguments and VM arguments with which your product is to be launched.

For a list of program and VM properties, refer to the [Runtime options](#) document.

Branding

Program Launcher
Customize the executable that is used to launch the product:

Launcher Name:

Customizing the launcher icons varies per platform:

- ▶ linux
- ▶ macosx
- ▶ solaris
- ▶ win32

Splash Screen
The splash screen appears when the product launches. If its location is not specified, the 'splash.bmp' file is assumed to be in the product's defining plug-in.

Specify the plug-in in which the splash screen is located:

Plug-in:

Window Images
Specify the images that will be associated with the application window. These GIF images are typically located in the product's defining plug-in.

16x16 Image:

32x32 Image:

About Dialog
Customize the text and image of the About dialog. The GIF image is typically located in the product's defining plug-in and its size must not exceed 500x330 pixels. The text is not shown if the image size exceeds 250x330 pixels.

Image:

Text:

Overview Configuration **Branding**

The **Branding** page is where the product is given its identity: launcher name, custom images, custom launcher icons, splash screen and About dialog.

Name the executable that will launch your product *rcpmail*.

Customizing the launcher icons varies per platform. In this example, we will leave the default Eclipse icon as-is.

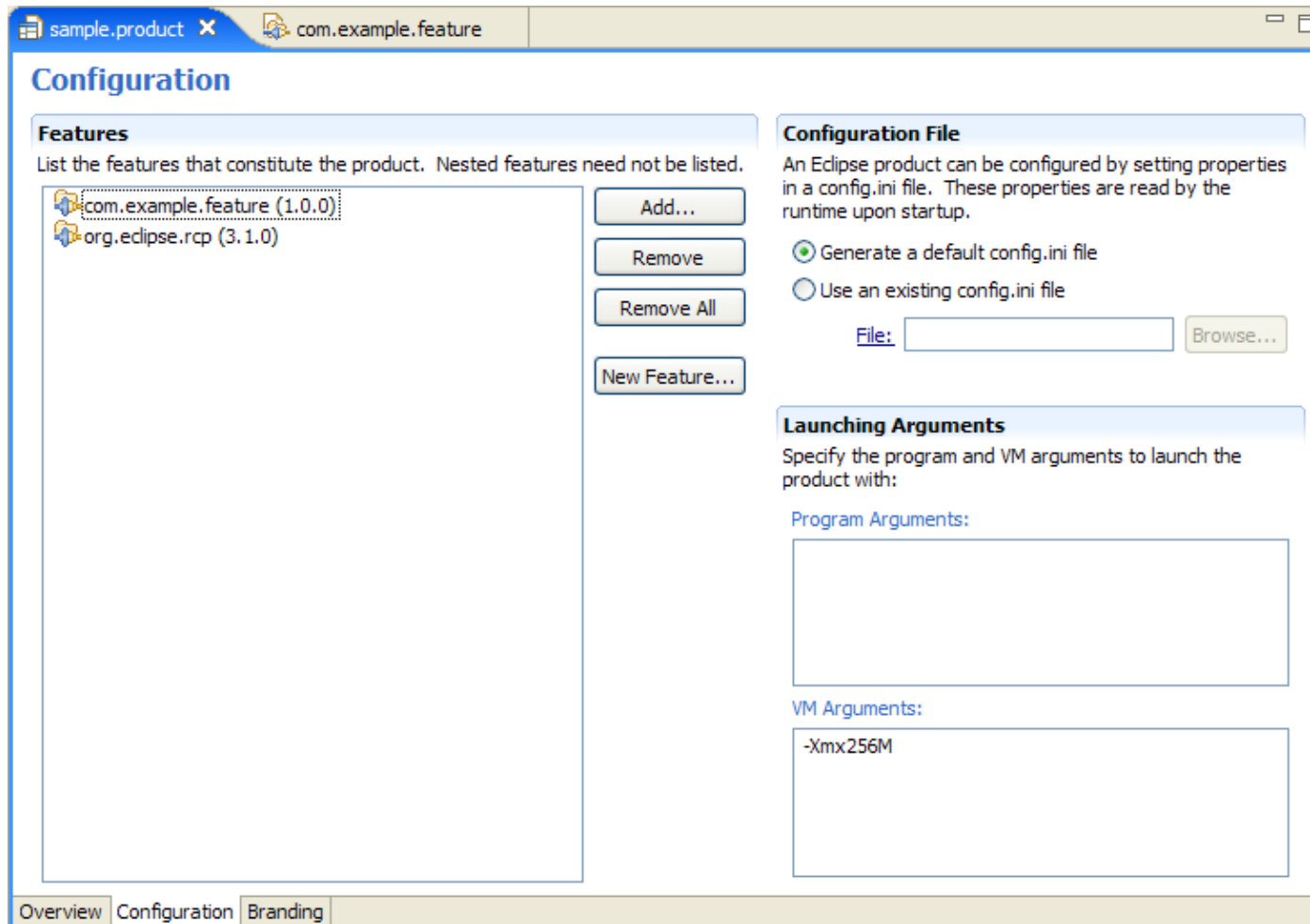
The splash screen must be named 'spash.bmp' and is assumed to be at the root of the product's defining plug-in if its location is not specified. In our example, the RCP mail plug-in project does have a splash.bmp at the right location, so the field can remain empty.

More customization can be done to the window images and the About dialog. All that data is already filled out by PDE using the plugin.xml of the *com.example.xyz* plug-in.

Feature-Based Product

To be able to take advantage of the Update Manager functionality and be able to publish upgrades and patches to your product, your product has to be feature-based.

Creating a feature-based product using the product editor is almost the same as a plugin-based product, with the only difference being the settings on the Configuration page.



Using the **New Feature...** button on the **Configuration** page of the product editor, create a feature named *com.example.feature* that contains plug-in *com.example.xyz*, as described in the [Feature section](#) of the guide.

On the [Overview page of the feature editor](#), the feature must declare the *com.example.xyz* plug-in as its **branding** plug-in.

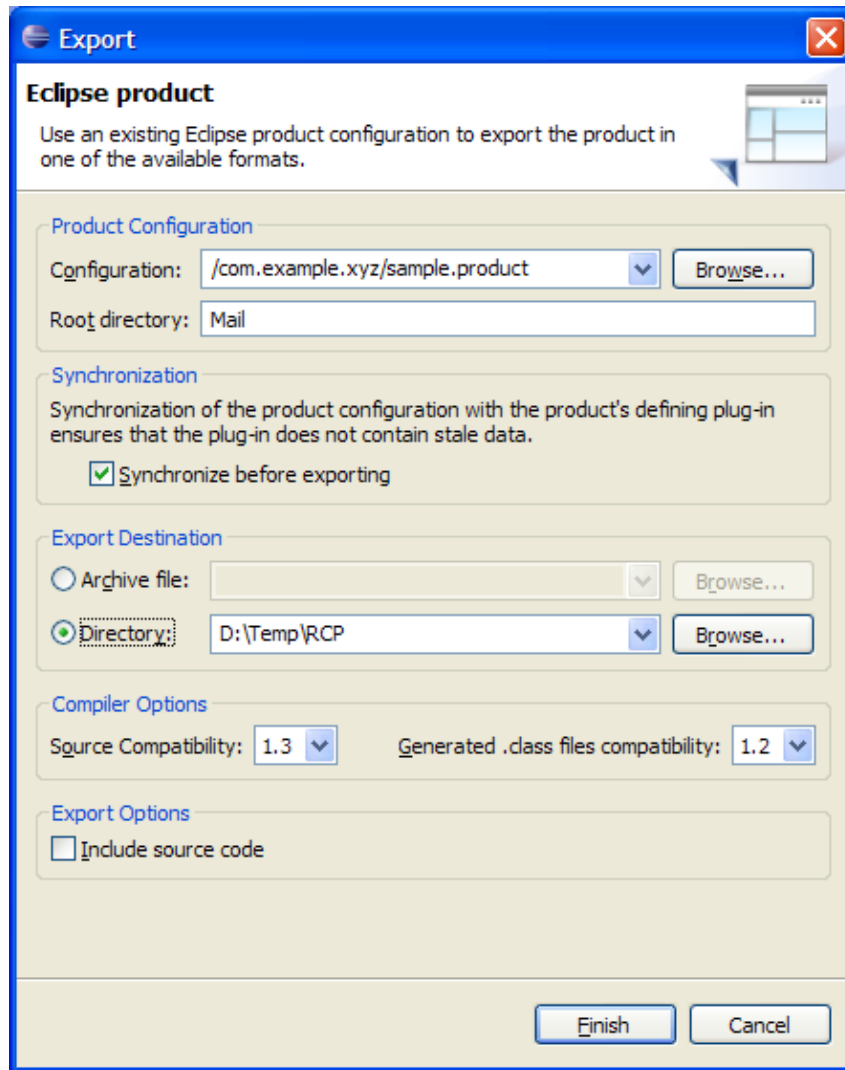
Add the *org.eclipse.rcp* feature to the list of features to be included in your product.

That is it!

Proceed to the Exporting a Product section as you normally would with a plugin-based configuration.

Exporting a Product

From the Exporting section of the product editor, click on the *Eclipse product export wizard* to bring up the wizard.



The root directory refers the name of the top-level directory of your product. You can name it anything you like and can be more than one segment.

The synchronization option also appears in the wizard and is checked by default to make sure that your plugin.xml does not contain stale data and is updated with the latest content from the product configuration.

A product can be exported to an archive or a directory. Select the **directory** option. Enter the name of a directory and Press **Finish**.

When the operation completes, go to that directory and launch your new customized product.


Simple as that!

Using extension point schema

Extension points defined by the plug-ins in your workspace are readily available to your own plug-in and other plug-ins. If an extension point schema has been defined for it, PDE can provide assistance when creating new extensions. This assistance includes:

- Providing choices for the **New** pop-up menu so that only valid child elements are added
- Providing attribute information for the property sheet so that only valid attributes are set
- Providing correct attribute property editors that match the attribute types (boolean, string, and enumeration).
- Providing additional support for special attribute types ("java" and "resource").
- Using the status line to show the first sentence of the documentation snippet for attributes when selected in the property sheet.

Example: Using the "Sample Parsers" extension point

Before trying to use the extension point we defined before, we still need to define the expected interface. Select the **com.example.xyz** project in the Navigator and press the  tool bar button to create a new Java interface. Be sure to set the package name to **com.example.xyz** and interface name to **IParser** before pressing **Finish**. Edit the interface to look like this:

```
package com.example.xyz;

public interface IParser {
    /**
     * Run the parser using the provided mode
     */
    public void parse(int mode);
}
```

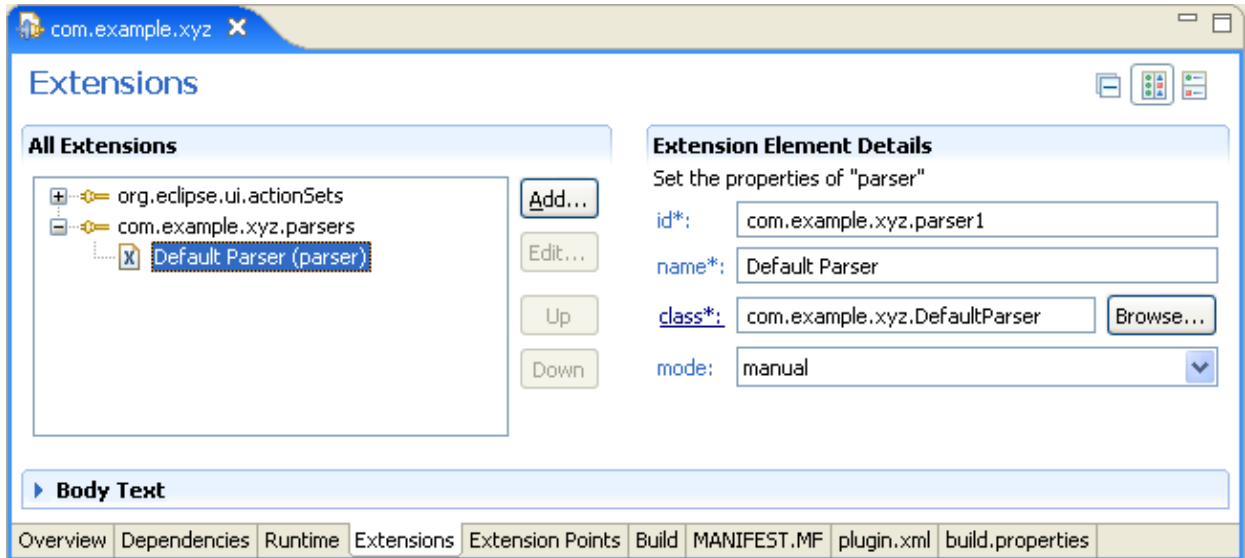
We now have the extension point, an XML schema for it, and the required interface. Be sure to save all of your open editors. Everything is now ready for our own plug-in or other plug-ins to contribute to the extension point.

1. Open the manifest editor for the **com.example.xyz** plug-in.
2. Switch to the Extensions page and press **New-> Extension**.
3. You should have "com.example.xyz.parsers" available as a choice. Select it and press **Finish**.
4. Select the newly added "com.example.xyz.parsers" element and popup the **New->parser** menu. (We specified that our extension point can accommodate any number of "parser" elements.)
5. Select the new parser element. The Extension Element Details section should show four attributes: **id**, **name**, **class** and **mode**. Note how the status line shows the short information about attributes as you select them. This information comes directly from the extension point schema.
6. Change the **name** to "Default Parser". Change the **mode** to "manual."
7. Click on the **class** hyperlink in the Extension Element Details section. Here you will see that PDE is seamlessly integrated with JDT's "New Java Class" wizard and utilizes schema attributes to automatically implement your IParser interface. Create your class with "com.example.xyz/src" as your source folder, "com.example.xyz" as the package, and **DefaultParser** as the class name. Press **Finish**.
8. You should now be in the Java editor for the **DefaultParser** class. Notice how it has implemented the right interface (**IParser**) and already has the stub implementation of the "parse" method. Note that if you close the editor and click on the **class** hyperlink again, the editor will reopen the

Using the Plug-in Development Environment

DefaultParser class. The "New Java Class" wizard will only appear when the class specified in the class attribute text field cannot be found; otherwise, the link will open the class in an editor.

As you can see, when you provide a complete XML schema for your extension point, it will help all your potential users by letting PDE to assist them and prevent them from making errors.



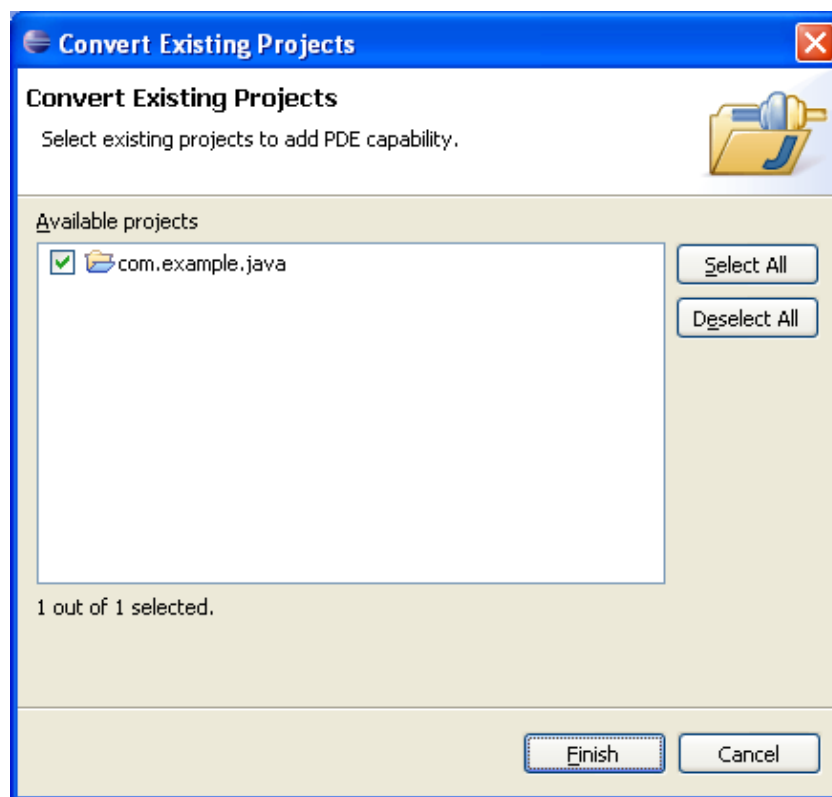
Converting existing projects into PDE projects

A PDE project is a project that "knows" it hosts a plug-in and is eligible for plug-in-specific operations.

PDE is fully capable of working on plug-ins in the workspace represented as ordinary projects. However, it cannot offer incremental file checking, capability-based filtering and other similar features if a project does not possess full PDE capabilities.

It is possible to convert a regular project into a PDE project at any point. For example, you may have some Java classes that you want to package into a library and share with others as a plug-in. Alternatively, you may want to get full support for manifest syntax checking and reporting that only PDE projects have.

To convert projects into PDE projects, select a project in the Navigator view. Select **PDE Tools > Convert Projects to Plug-in Projects...** from the context menu.



The wizard will list all projects that do not have PDE capability. Candidate projects do not need to have manifest files. If they are missing, PDE will create generic ones you can use as a starting point. Files that already exist will be left intact. Select one or more projects, click **Finish**.

PDE Extension Points

The following extension points can be used to extend the capabilities of the PDE infrastructure:

- [org.eclipse.pde.core.source](#)
- [org.eclipse.pde.ui.newExtension](#)
- [org.eclipse.pde.ui.pluginContent](#)
- [org.eclipse.pde.ui.templates](#)

Extension Wizards

Identifier:

org.eclipse.pde.ui.newExtension

Description:

This extension point should be used to contribute wizards that will be used to create and edit new extensions in PDE plug-in manifest editor. Wizards can create one or more extensions at the same time, as well as the code needed to implement those extensions. If a contributed wizard is specifically created for a particular extension point, it is advisable to also register a matching editor wizard. This wizard will be used to edit the extension point in the manifest editor after it has been created in the manifest file.

Configuration Markup:

```
<!ELEMENT extension (wizard | category | editorWizard)*>
```

```
<!--ATTLIST extension
```

```
point CDATA #REQUIRED
```

```
id CDATA #IMPLIED
```

```
name CDATA #IMPLIED>
```

- **point** – a fully qualified identifier of the target extension point
- **id** – an optional identifier of the extension instance
- **name** – an optional name of the extension instance

```
<!ELEMENT wizard (description?)>
```

```
<!--ATTLIST wizard
```

```
id CDATA #REQUIRED
```

```
name CDATA #REQUIRED
```

```
icon CDATA #IMPLIED
```

```
class CDATA #IMPLIED
```

```
availableAsShortcut (true | false)
```

```
category CDATA #IMPLIED
```

```
template CDATA #IMPLIED>
```

a wizard that can be used to create a new extension from within the plug-in manifest editor

Using the Plug-in Development Environment

- **id** – a unique name that will be used to identify this wizard.
- **name** – a translatable name that will be used in UI representation of this wizard.
- **icon** – a relative path of an icon that will be used to visually represent the wizard.
- **class** – a fully qualified name of a class which implements `org.eclipse.pde.ui.IExtensionWizard` interface. This attribute is mutually exclusive with the `template` attribute.
- **availableAsShortcut** – If `true`, this wizard will appear in the short cut menu on the menu bar and the tool bar.
- **category** – an optional id that makes this wizard a member of the previously defined category. If category is hierarchical, full path to the parent category should be specified using '/' as a delimiter.
- **template** – an identifier of a template declared elsewhere using the extension point `org.eclipse.pde.ui.templates`. If defined, the template with the specified id will be located and the extension wizard will be created using the template. This attribute is mutually exclusive with the `class` attribute.

<!ELEMENT editorWizard (description?)>

<!ATTLIST editorWizard

id CDATA #REQUIRED

name CDATA #REQUIRED

icon CDATA #IMPLIED

class CDATA #REQUIRED

point CDATA #REQUIRED>

a wizard that can be used to edit an existing extension from within the plug-in manifest editor

- **id** – a unique name that will be used to identify this wizard.
- **name** – a translatable name that will be used in UI representation of this wizard.
- **icon** – a relative path of an icon that will be used to visually represent the wizard.
- **class** – a fully qualified name of a class which implements `org.eclipse.pde.ui.IExtensionEditorWizard` interface.
- **point** – a fully qualified identifier of the extension point that this wizard is capable of editing

<!ELEMENT category EMPTY>

<!ATTLIST category

id CDATA #REQUIRED

Description:

Using the Plug-in Development Environment

name CDATA #REQUIRED

parentCategory CDATA #IMPLIED>

- **id** – a unique name that will be used to reference this category
- **name** – a translatable name that will be used for UI presentation of this category
- **parentCategory** – an optional attribute that can be used to create category hierarchy

<!ELEMENT description (#PCDATA)>

A short description of this wizard.

Examples:

The following is an example of the extension:

<extension point=

"org.eclipse.pde.ui.newExtension"

>

<category name=

"Custom Extensions"

id=

"custom"

>

</category>

<wizard availableAsShortcut=

"true"

name=

"Simple Java Editor Extension"

icon=

"icons/java_edit.gif"

Description:

Using the Plug-in Development Environment

```
category=  
  
"generic"  
  
class=  
  
"com.example.xyz.SimpleJavaEditorExtension"  
  
id=  
  
"com.example.xyz.simple"  
  
>
```

<description>

This wizard creates a simple Java editor with all the required classes and manifest markup.

</description>

</wizard>

</extension>

API Information:

This extension point requires that a class that implements `org.eclipse.pde.ui.IExtensionWizard` interface.

Supplied Implementation:

PDE provides a generic wizard that creates extension points based on the extension point schema information. In addition, all templates registered using `org.eclipse.pde.ui.templates` extension point in PDE UI are also hooked as individual extension wizards.

Copyright (c) 2004 IBM Corporation and others.

All rights reserved. This program and the accompanying materials are made available under the terms of the Eclipse Public License v1.0 which accompanies this distribution, and is available at <http://www.eclipse.org/legal/epl-v10.html>.

Plug-in Content Wizards

Identifier:

org.eclipse.pde.ui.pluginContent

Description:

This extension point provides for contributing wizards that create additional content of the PDE plug-in projects. After the plug-in manifest and key files have been created, these wizards can be used to add more files and extensions to the initial structure. A typical implementation of this wizard would add content based on a parametrized template customized based on the user choices in the wizard. The goal is to arrive at a plug-in that is does something useful right after the creation (e.g. contributes a view, an editor etc.).

Configuration Markup:

<!ELEMENT extension (wizard*)>

<!--ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point** – a fully qualified identifier of the target extension point
- **id** – an optional identifier of the extension instance
- **name** – an optional name of the extension instance

<!ELEMENT wizard (description?)>

<!--ATTLIST wizard

id CDATA #REQUIRED

name CDATA #REQUIRED

icon CDATA #IMPLIED

class CDATA #REQUIRED

category CDATA #IMPLIED

ui-content (true | false) "true"

java (true | false) "true"

rcp (true | false) "false">

Using the Plug-in Development Environment

- **id** – a unique name that will be used to identify this wizard.
- **name** – a translatable name that will be used in UI representation of this wizard.
- **icon** – a relative path of an icon that will be used to visually represent the wizard.
- **class** – a fully qualified name of a class which implements `org.eclipse.pde.ui.IPluginContentWizard`.
- **category** – an optional tag that can be used to associate content wizards with different target projects.
- **ui-content** – a flag that indicates if the wizard will contribute code with user interface content. This flag will affect which plug-in class will be picked since (UI plug-ins extend `AbstractUIPlugin` class, while non-UI plug-ins extends `Plugin` base class). Since many contributions to Eclipse have UI content, this attribute is `true` by default.
- **java** – a flag that indicates that the wizard will contribute Java content. Since most of the Eclipse plug-ins have Java code, the attribute is `true` by default. Set it to `false` if the plug-in will not have Java code (for example, documentation files only).
- **rcp** – Since 3.1. A boolean flag indicating whether the wizard contributes a standalone fully-functioning rich client application. If set to `true`, the wizard will appear in the New Plug-in Project wizard only when the user chooses the Rich Client Application option.

<!ELEMENT description (#PCDATA)>

Short description of this wizard.

Examples:

The following is an example of this extension point:

```
<extension point=
"org.eclipse.pde.ui.pluginContent"
>
<wizard name=
"Example Plug-in Content Generator"
icon=
"icons/content_wizard.gif"
class=
"com.example.xyz.ContentGeneratorWizard"
id=
```

Description:

Using the Plug-in Development Environment

```
"com.example.xyz.ExampleContentGenerator"
```

```
>
```

```
<description>
```

Adds a view and a preference page.

```
</description>
```

```
</wizard>
```

```
</extension>
```

API Information:

Wizards that plug into this extension point must implement `org.eclipse.pde.ui.IPluginContentWizard` interface and is expected to extend `org.eclipse.jface.wizard.Wizard`.

Supplied Implementation:

PDE provides APIs for contributing content wizards based on customizable templates. A number of concrete wizards based on these templates is contributed by PDE UI itself.

Copyright (c) 2004 IBM Corporation and others.

All rights reserved. This program and the accompanying materials are made available under the terms of the Eclipse Public License v1.0 which accompanies this distribution, and is available at <http://www.eclipse.org/legal/epl-v10.html>.

Extension Templates

Identifier:

org.eclipse.pde.ui.templates

Since:

2.0

Description:

This extension point registers plug-in project content templates that are used to generate code for the new extensions. Templates are used in two contexts:

- One or more templates are combined in a wizard that is contributed as plug-in content wizard using `org.eclipse.pde.ui.pluginContent` extension point. These templates create interesting content for newly created plug-in projects. In addition, all the templates contributed using this extension point can be seen in a special version of the plug-in content wizard that lists the templates and allows users to freely combine the templates by checking them in the list.
- New extension can be added to an existing plug-in using a template.

Configuration Markup:

<!ELEMENT extension (template+)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point** – a fully qualified identifier of the target extension point
- **id** – an optional identifier of the extension instance
- **name** – an optional name of the extension instance

<!ELEMENT template EMPTY>

<!ATTLIST template

id CDATA #REQUIRED

name CDATA #REQUIRED

icon CDATA #IMPLIED

class CDATA #REQUIRED

Using the Plug-in Development Environment

contributingId CDATA #REQUIRED>

- **id** – a unique name that will be used to identify this template.
- **name** – a translatable name that will be used in UI representation of this template.
- **icon** – a relative path of an icon that will be used to visually represent the template.
- **class** – a fully qualified name of the class that implements `org.eclipse.pde.ui.templates.ITemplateSection` interface.
- **contributingId** – the identifier of the extension point that this template will contribute into.

Examples:

The following is an example of the template registration:

```
<extension point=
"org.eclipse.pde.ui.templates"
>
<template contributingId=
"org.eclipse.ui.actionSets"
name=
"XYZ Action Set Generator"
class=
"com.example.xyz.XYZActionSetTemplate"
id=
"com.example.xyz.ActionSetTemplate"
>
</template>
</extension>
```

API Information:

Each template must provide a class that implements `org.eclipse.pde.ui.templates.ITemplateSection` interface. However, abstract classes that implement the interface and can be extended are available.

Since:

Using the Plug-in Development Environment

Supplied Implementation:

PDE UI contributes a number of templates that create extensions for the most popular extension points like editors, views, preferences etc.

Copyright (c) 2004 IBM Corporation and others.

All rights reserved. This program and the accompanying materials are made available under the terms of the Eclipse Public License v1.0 which accompanies this distribution, and is available at

<http://www.eclipse.org/legal/epl-v10.html>.

Other Reference Information

The following specifications, white papers, and design notes describe various aspects of the PDE.

- Map of PDE Plug-ins

Map of PDE Plug-ins

The PDE provides a comprehensive plug-in development environment.

PDE itself is divided up into a number of separate plug-ins. The following table shows which API packages are found in which plug-ins (the most commonly referenced API packages are highlighted). This table is useful for determining which plug-ins a given plug-in should include as prerequisites.

| API Package | Plug-in id |
|---------------------------------------|-----------------------------------|
| <code>org.eclipse.pde.core[.*]</code> | <code>org.eclipse.pde.core</code> |
| <code>org.eclipse.pde.ui[.*]</code> | <code>org.eclipse.pde.ui</code> |

PDE Dynamic Classpaths FAQ

Dynamic Classpaths is the way PDE computes the build path for plug-in projects in Eclipse 3.0.

Q: What is classpath stability?

A: Classpath stability is a measure of classpath change with respect to the self-hosting choice made by a developer. Ideally, classpaths should not change regardless of the complement of source projects in the workspace. Binary project self-hosting offers good classpath stability, where all classpaths contain only project references. External plug-in self-hosting provides less stable classpaths. They are still stable with respect to the local install location of the external libraries, but the list of plug-ins as source projects must remain constant for all the members of a team in order to share them via a repository.

Since 2.0, added plug-in version to the plug-in location on the file system further reduced the classpath stability when external plug-ins were used.

Q: If binary projects offer better classpath stability, why not use them all the time?

A: Self-hosting using imported binary projects is a good choice as long the number of imported plug-ins is reasonably small (a few dozen). For large products with hundreds of plug-ins, a wholesale import is not an option. Typically, their developers self-host with a few source projects, a few dozen directly related binary projects, and everything else as external plug-ins. From the purely theoretical point of view, it seems strange to waste time and resources to import dozens and dozens of external plug-ins to be able to compile a few source projects.

Q: I think that (binary projects/external plug-ins) self-hosting method is better. What is wrong with my team using it if we do it together?

A: Static classpaths (either using binary projects or external plug-ins) cast in stone your self-hosting method of choice and force everybody else to use it as well.

Q: What are dynamic classpaths?

A: **Dynamic classpaths** is a PDE feature whereby a portion of the plug-in project classpath that is related to plug-in dependencies is computed dynamically using JDT classpath container technology. The resolution of the dynamic classpaths is performed 'just in time' and is always up to date with the conditions in your workspace. Furthermore, the dynamic nature of the classpath resolution allows PDE to adapt to changes and always have the correct classpath regardless of your method of self-hosting.

Q: What is the classpath stability of dynamic classpaths?

A: Ultimate. Since all the entries for the required plug-ins are replaced with one classpath container entry, your classpath is always the same.

Q: How can dynamic classpaths help me?

A: With dynamic classpaths, there is no need to make upfront decisions with respect to the self-hosting style. If binary projects are present, dynamic classpaths will resolve to project references. If not, they will resolve to external plug-in JARs. As binary projects are added or removed, dynamic classpaths will track the changes and adapt. You will not need to update your classpath ever again. In addition, other teams that want to take one or more of your projects from CVS and get them to compile don't have to use your personal self-hosting style to do so.

Q: Since PDE Core is resolving the dynamic classpaths, does it mean that I will be dependent on PDE to do the right thing?

A: In a word, yes. Being dynamic, your classpath will always be computed on the fly, not hard-coded in the .classpath file (that was the whole idea, right?). But consider this: PDE has a sophisticated algorithm for computing the classpath that strives to get you as close as possible to the run-time conditions. What the JDT compiler 'sees' at development time should be as close as possible to what class loaders will see at run time. PDE Core is more capable of keeping your classpath up to date than yourself most of the time. If you need to manually tweak the classpath in order to compile, something is most likely wrong with your setup and there is a strong chance your plug-in will not run correctly (SWT team being an exception).

Q: My team uses binary projects for self-hosting exclusively. Will I lose anything by switching to dynamic classpaths?

A: No. Dynamic classpaths do not dictate your personal choice of self-hosting arrangements. It simply resolves your plug-in dependencies in the given context. If you continue to import external plug-ins as binary projects, dynamic classpaths will resolve to project references as before.

Q: What do I need to activate dynamic classpaths?

A: Update the classpaths of all your 2.1 plug-ins just once. You will notice that classpaths are now shorter and all the dependent plug-in references are now replaced with one container entry. You can continue to work. Make sure to check the source projects into the repository, including the changed .classpath files.

Q: I have extra classpath entries so that I can compile my Ant tasks/servlets/JSPs.

A: As part of the classpath computation, PDE takes 'jars.extra.classpath' property from build.properties file into account. If you are set up correctly for building, PDE will generate the correct classpath.

Q: How do I manipulate the dynamically computed classpath entries?

A: In the unlikely event you need to manipulate your dynamic classpath entries, you can do so from Properties>Java Build Path>Libraries tab. Expand the 'Plug-in Dependencies' node and manipulate the entries there.

Q: Some of the computed entries for libraries don't have source attachments. Can I add them manually?

A: PDE computes source attachments for most of the libraries. There are some odd cases where automatic source attachments may fail due to source zips not following naming conventions. You can attach sources manually for these entries in the build path properties dialog.

Q: Will my manual source attachments be wiped out the next time PDE computes the classpath dynamically?

A: No. PDE keeps track of these manual cases and reapplies them after dynamic computation as long as the library paths didn't change.

Q: I am an SWT developer. Can I use dynamic classpaths?

A: Sadly, no. SWT team has a unique self-hosting setup whereby classpaths for various environments are

saved in the repository and renamed into .classpath in the project depending on the platform they are working on. They will have to continue to use their self-hosting methods.

Tips and Tricks

| | |
|-----------------------------------|--|
| Feature-based self-hosting | <p>The current method of self-hosting in Eclipse is plug-in-based. PDE launches a second run-time workbench instance by passing an array of plug-ins that it should load. A regular Eclipse product is feature-based: during startup, it checks all the features that should be active, computes plug-ins that belong to those features, and passes the result for loading.</p> <p>This difference in behavior makes it complicated to self-host in scenarios where a full startup that involves features is required. PDE now supports this scenario if care is taken with the setup:</p> <ol style="list-style-type: none"> 1. The workspace needs to be <work-area>/plugins. 2. Features must be imported into the workspace using the new 'Feature Import' wizard (they will be created in <work-area>/features). 3. All plug-ins must be in the workspace (either in source or imported as binary projects WITHOUT linking). 4. When launching, Run-time Workbench launcher must be configured to use features (in Plug-ins and Fragments tab). <p>If all these conditions are met, the runtime Eclipse instance will be launched in a way that is the closest possible approximation of a normal Eclipse startup. This facilitates testing About dialogs and other aspects that may depend on the set of installed features.</p> |
| To clean or not to clean | <p>When you create a new runtime workbench launch configuration, PDE presets the Program Arguments on the launch configuration to include a -clean argument.</p> <p>This -clean argument clears all runtime-cached data in your runtime workbench from one invocation to the next to ensure that all the changes made in your host workbench, e.g. new Java packages were added to a plug-in project, etc., are picked up when you launch a runtime workbench.</p> <p>This clearing of the cache may hinder performance if your target platform contains a large number of plug-ins.</p> <p>Therefore, if you're in a situation where your target platform has a large number of plug-ins and you're at a stage where you are not actively adding/removing packages from your plug-in projects, you could remove the -clean argument from the launch configuration to improve startup time.</p> |

Using the Plug-in Development Environment

| | |
|------------------------------------|--|
| Importing with linking | <p>Importing external plug-ins and fragments can be time consuming and may result in large workspaces, depending on the content of the plug-ins being imported. Therefore, the 'Import External Plug-ins and Fragments' wizard gives you the option to import with linking. This means that the import operation will not copy the resources being imported into your workspace. It will simply create links to the files being imported. You will be able to browse these linked resources, as if they had been copied into your workspace. However, they are physically not there on your file system, so you will not be able to modify them. Beware of operations that depend on files being physically in your workspace, as they will not work on linked resources.</p> |
| Templates | <p>For a quick start, PDE provides several template plug-ins that will generate a plug-in with one or more fully-working extensions. In addition, if at any point, you would like to add a new extension from the template list (without having to generate a plug-in), you could access these extension templates directly from the manifest editor. From the 'Extensions' page of the editor, click 'Add...'. In the wizard that comes up, select Extension Templates in the left pane and choose the template of choice in the right pane.</p> |
| Plug-in dependency extent | <p>If you have ever looked at the list of plug-ins that your plug-in depends on and wondered why your plug-in needs a particular plug-in X, now you can easily find out why.</p> <p>The Compute Dependency Extent operation found on the context menu in several contexts (including manifest file Dependencies page and Dependencies view) performs a combined Java and plug-in search to find all Java types and extension points provided by plug-in X which are referenced by your plug-in. The results will be displayed in the Search view. When a type is selected in the Search results view, the References in MyPlugIn action in the context menu searches for the places in the plug-in where the selected type is referenced.</p> <p>If the search returns 0 results, you should definitely remove plug-in X from your list of dependencies, as it is not being used at all, and it would just slow class loading.</p> <p>The Compute Dependency Extent is also useful to check if you are using internal (non-API) classes from plug-in X, which might not be desirable.</p> |
| Finding unused dependencies | <p>Minimizing a plug-in's number of dependencies is certain to improve performance. As your plug-in evolves, its list of dependencies might become stale, as it might be still containing references to plug-ins that it no longer needs. A quick way to check that all dependencies listed by your plug-in are actually used by the plug-in is to run the 'Find Unused Dependencies' utility, which is available through the context menu of the 'Dependencies' page of PDE's manifest editor.</p> |

| | |
|--|---|
| Extending the Java search scope | Java Search is limited to projects in your workspace and external jars that these projects reference. If you would like to add more libraries from external plug-ins into the search: open the Plug-ins View, select a plug-in and choose Add to Java Search from the context menu. This is handy for remaining aware of other plug-ins that depend on ones you're working on. |
|--|---|

What's New in 3.1

This document contains descriptions of some of the more interesting or significant changes made to PDE for the 3.1 release of Eclipse since 3.0.

PDE

Using the Plug-in Development Environment

Bundle manifests for plug-ins

In Eclipse 3.1, it is strongly recommended that plug-ins contain an OSGi bundle manifest.mf. In addition to faster startup and classloading, this format will allow you to take advantage of many of the new runtime capabilities such as fine control over what packages you want to expose to clients.

The option to create a manifest.mf in the New Plug-in Project creation wizard is now on by default.

Plug-in Project

Create a new plug-in project

Project name:

Project contents

☒ Use default

Directory:

Project Settings

☒ Create a Java project

Source Folder Name:

Output Folder Name:

Plug-in Format

What version of Eclipse is this plug-in targeted for?

☒ Create an OSGi bundle manifest

You can create a bundle manifest.mf for an existing plug-in on the Overview page of the plug-in manifest editor.

Plug-in Content

The content of the plug-in is made up of four 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:** declares contributions this plug-in makes to the platform.
- Extension Points:** declares new function points this plug-in adds to the platform.

For this plug-in to take advantage of additional runtime capabilities, you need to [create an OSGi bundle manifest](#).

Using the Plug-in Development Environment

PDE enforces code accessibility

The plug-in's manifest.mf file allows you to control on a per-package basis the visibility of your plug-in's code to downstream plug-ins.

PDE manages each plug-in's Java classpath and checks these visibility rules at compile time. This means no one will never be caught by surprise by classloading errors at runtime, and will always be aware when they are referencing internal (discouraged) types.

```
in != null) {  
File file = new File(destination, zipEntry.getName());  
AbstractFrameworkAdaptor.readFile(in, file);  
Discouraged access: The type AbstractFrameworkAdaptor is not accessible due to restriction on  
required library D:\Eclipse\ eclipse\plugins\org.eclipse.osgi_3.1.0.jar  
(IOException e) {
```

For full details, refer to the [Access Restrictions](#) document.

Creating a rich client application

The New Plug-in Project wizard gives you the option to create a rich client application.

Plug-in Content

Enter the data required to generate the plug-in.



Plug-in Properties

Plug-in ID:

Plug-in Version:

Plug-in Name:

Plug-in Provider:

Classpath:

Plug-in Class

☒ Generate the Java class that controls the plug-in's life cycle

Class Name:

☒ This plug-in will make contributions to the UI

Rich Client Application

Would you like to create a rich client application? ☒ Yes ☐ No

Using the Plug-in Development Environment

RCP templates The New Plug-in Project wizard provides ready-to-run RCP templates. The templates range from a minimal Hello RCP template to a rich, fully-branded RCP mail template.





Templates

Select one of the available templates to generate a fully-functioning plug-in.



☒ Create a plug-in using one of the templates

Available Templates:

-  Hello RCP
-  RCP application with an intro
-  RCP application with a view
-  RCP Mail Template

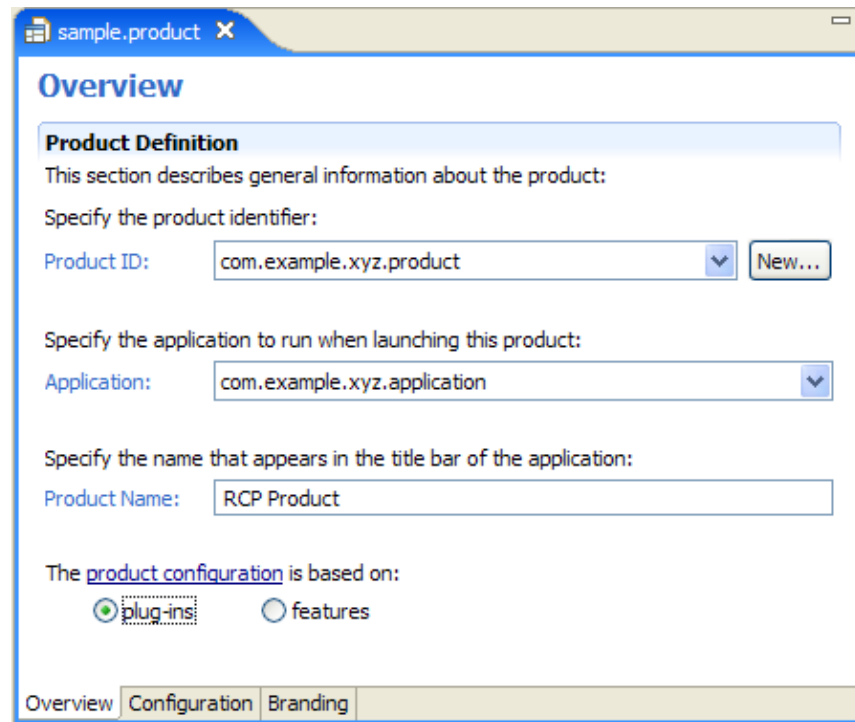
This wizard creates a standalone RCP application that shows how to:

- add top-level menu and toolbar with actions
- add keybindings to actions
- create views that can't be closed and multiple instances of the same view
- create perspectives with placeholders for new views

Build an Eclipse product with a single click

You can now create and manage an Eclipse product in a *.product file, which can be created via **File > New > Other... > Product Configuration**.

The product configuration editor manages all aspects of a product from basic definition to branding. You can create plug-in-based and feature-based products. The overview page provides hot links to test and export the product.

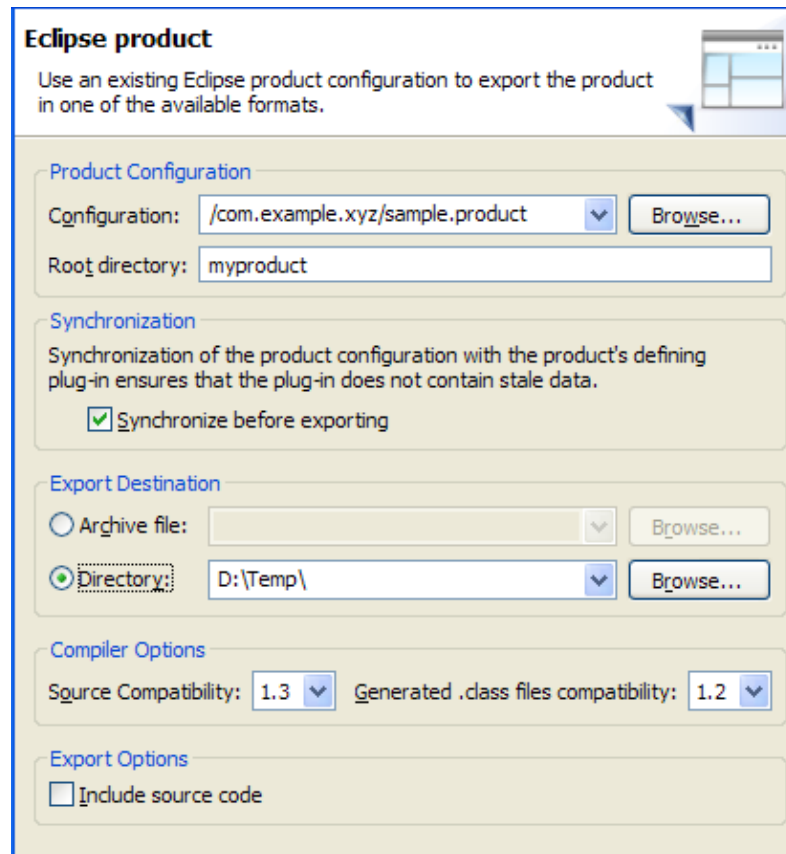


Eclipse product export wizard

You can export an Eclipse product as an archive or a directory structure in the Eclipse Product Export wizard.

The wizard is invoked via **File > Export > Eclipse Product** or from the **Overview** page of the Product Configuration editor.

Using the Plug-in Development Environment



Eclipse product

Use an existing Eclipse product configuration to export the product in one of the available formats.

Product Configuration

Configuration:

Root directory:

Synchronization

Synchronization of the product configuration with the product's defining plug-in ensures that the plug-in does not contain stale data.

☒ Synchronize before exporting

Export Destination

☐ Archive file:

☒ Directory:

Compiler Options

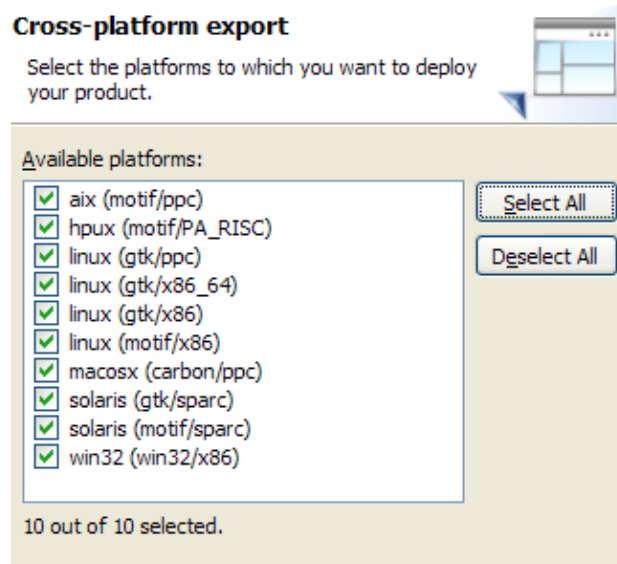
Source Compatibility: Generated .class files compatibility:

Export Options

☐ Include source code

Cross-platform product export

If you have the RCP delta pack installed, you can now build and export your product for multiple platforms at the same time via the Eclipse Product export wizard (**File > Export > Eclipse Product**).



Cross-platform export

Select the platforms to which you want to deploy your product.

Available platforms:

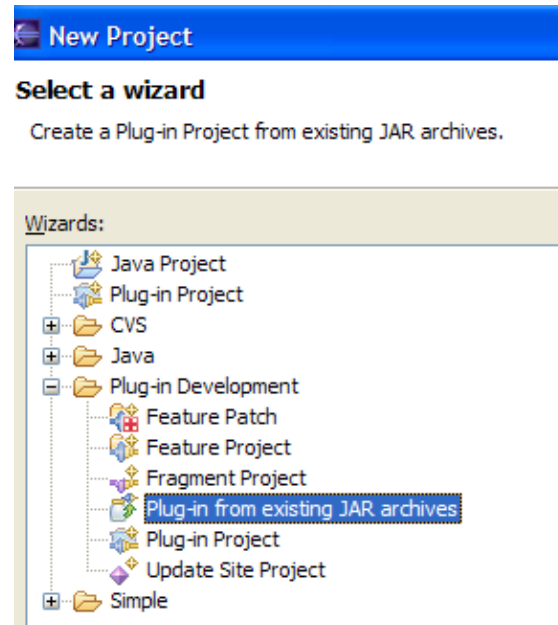
| | |
|-------------------------------------|-----------------------|
| <input checked="" type="checkbox"/> | aix (motif/ppc) |
| <input checked="" type="checkbox"/> | hpux (motif/PA_RISC) |
| <input checked="" type="checkbox"/> | linux (gtk/ppc) |
| <input checked="" type="checkbox"/> | linux (gtk/x86_64) |
| <input checked="" type="checkbox"/> | linux (gtk/x86) |
| <input checked="" type="checkbox"/> | linux (motif/x86) |
| <input checked="" type="checkbox"/> | macosx (carbon/ppc) |
| <input checked="" type="checkbox"/> | solaris (gtk/sparc) |
| <input checked="" type="checkbox"/> | solaris (motif/sparc) |
| <input checked="" type="checkbox"/> | win32 (win32/x86) |

10 out of 10 selected.

Create a plug-in from existing JAR

PDE now provides a wizard that creates a plug-in from existing JAR archives. This wizard is ideal if you would like to package third-party non-Eclipse JARs as an Eclipse plug-in.

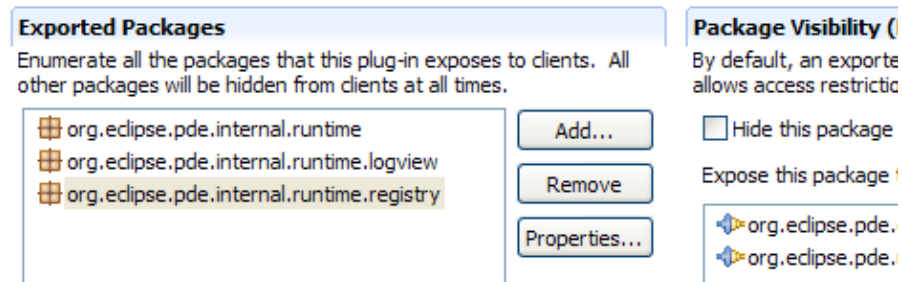
The wizard can be invoked via **File > New > Project > Plug-in from existing JAR archives**.



Manifest editor supports more OSGi bundle manifest headers

The PDE plug-in editor now exposes many interesting features of the runtime that are available only if your plug-in has a manifest.mf file. The Runtime page of the editor, for example, is the place where you can control access to your plug-in's code on a fine-grained level.

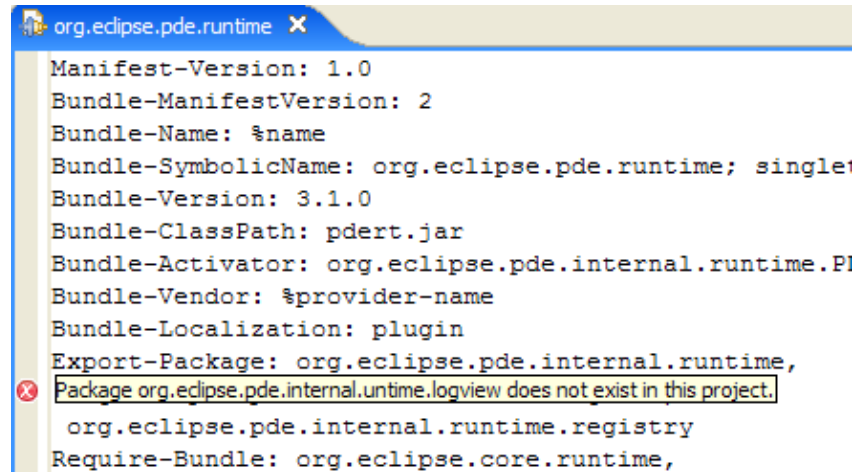
Runtime



Using the Plug-in Development Environment

Manifest.mf validation

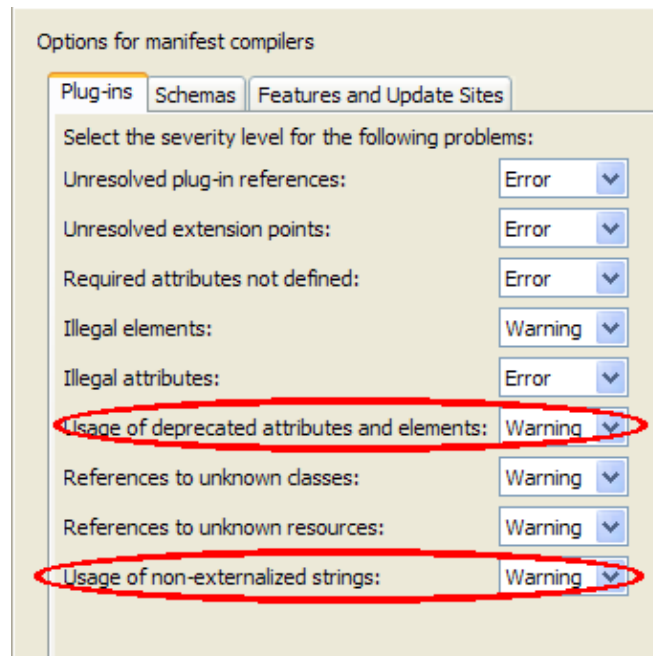
PDE now flags syntactic and semantic errors in the plug-in's manifest.mf file.



Improved plugin.xml validation

Attributes and elements defined in extension point schemas can now be marked as translatable. Also, obsolete attributes and elements can be marked as deprecated, in the same spirit as `@deprecated` tag in obsolete Java APIs.

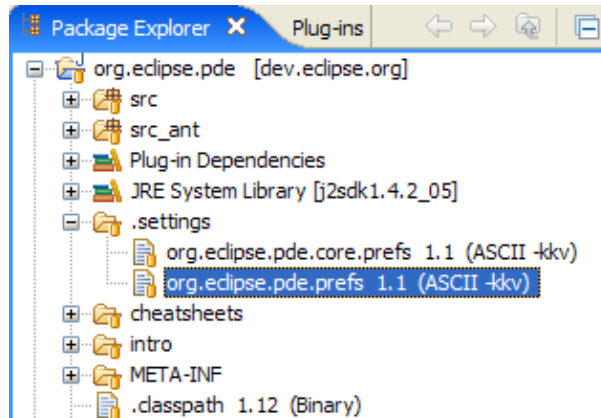
PDE uses this metadata to flag the usage of deprecated and non-externalized attributes and elements in the plug-in's manifest files.



Using the Plug-in Development Environment

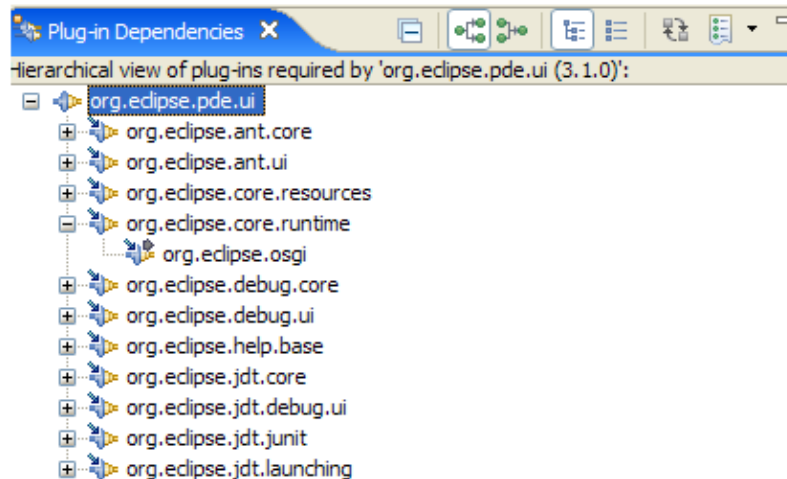
Sharing plug-in manifest compiler settings

You can now set the PDE plug-in manifest compiler settings on a per-project basis and share these settings among team members.



Improved plug-in dependencies view

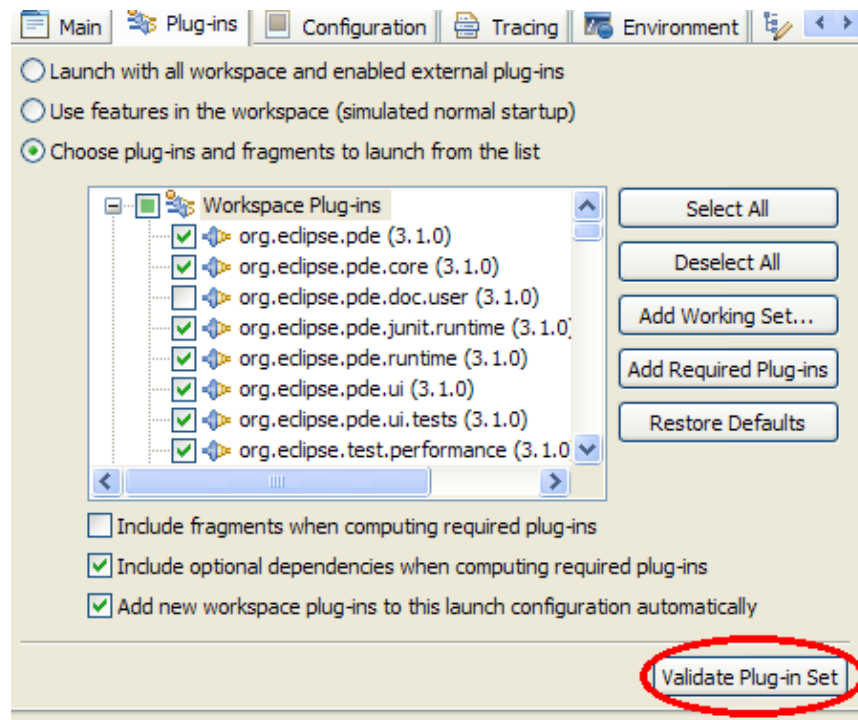
The PDE Plug-in Dependencies view now shows both tabular and tree visualizations of dependencies, as well as caller/callee relationships and cyclic dependencies. This view can be opened from the context menu of plug-in project via **PDE Tools > Open Dependencies**.



Validate plug-in set before launching

Prior to launching your Eclipse application, you can now validate the selected subset of plug-ins to find lurking launch startup problems such as unsatisfied plug-in dependencies, missing applications, etc.

Using the Plug-in Development Environment



| | |
|------------------------------|--|
| No need to
-clean | When self-hosting with PDE, you no longer need to launch a runtime Eclipse application with the <code>-clean</code> program argument. Leaving this argument off significantly improves startup time. |
|------------------------------|--|

| | |
|---|---|
| Improved
feature and
update site
support | The PDE feature and update site manifest editors have been redesigned to provide a simpler and better workflow. Improvements include the ability to build and package features without having to import them into your workspace. |
|---|---|

| | |
|---|---|
| JNLP
manifests and
JAR signing | The feature export wizard now provides you with the option to create JNLP manifests and digitally sign your plug-in and feature archives for Java Web Start deployment. |
|---|---|

Advanced Options

Sign the JAR archives and generate JNLP manifests



Signing JAR Archives
☒ Sign the JAR archives using a keystore (a password-protected database)
Keystore location:
Alias:
Password:

Java Network Launching Protocol (JNLP)
☒ Create JNLP manifests for the JAR archives
Site URL:
JRE version:

Notices

The material in this guide is Copyright (c) IBM Corporation and others 2000, 2005.

[Terms and conditions regarding the use of this guide.](#)

About This Content

February 24, 2005

License

The Eclipse Foundation makes available all content in this plug-in ("Content"). Unless otherwise indicated below, the Content is provided to you under the terms and conditions of the Eclipse Public License Version 1.0 ("EPL"). A copy of the EPL is available at <http://www.eclipse.org/legal/epl-v10.html>. For purposes of the EPL, "Program" will mean the Content.

If you did not receive this Content directly from the Eclipse Foundation, the Content is being redistributed by another party ("Redistributor") and different terms and conditions may apply to your use of any object code in the Content. Check the Redistributor's license that was provided with the Content. If no such license exists, contact the Redistributor. Unless otherwise indicated below, the terms and conditions of the EPL still apply to any source code in the Content.