

Using the Plug-in Development Environment

IBM

Table of Contents

<u>PDE Guide</u>	1
<u>Introduction to PDE</u>	2
<u>Notices</u>	2
<u>About This Content</u>	2
<u>License</u>	2
<u>Contributions</u>	2
<u>Preparing the workbench</u>	3
<u>Error Log</u>	3
<u>Concepts</u>	4
<u>Host vs. runtime</u>	4
<u>Target Platform</u>	4
<u>External vs. workspace plug-ins</u>	5
<u>Creating a plug-in project</u>	7
<u>Plug-in manifest editor</u>	12
<u>Overview page</u>	12
<u>Dependencies page</u>	13
<u>Runtime page</u>	14
<u>Extensions page</u>	15
<u>Extension points page</u>	17
<u>Extension point schema editor</u>	18
<u>Example: Creating schema for the "Sample Parsers" extension point</u>	18
<u>Build configuration page</u>	24
<u>Source page</u>	25
<u>Sample Parsers</u>	28
<u>Source Locations</u>	29
<u>Extension point schema</u>	32
<u>The benefits of extension point schemas</u>	32
<u>Limitations of PDE XML Schema support</u>	32
<u>Running a plug-in</u>	34
<u>Example: Running the Sample</u>	35
<u>Choosing plug-ins to run</u>	35
<u>Running with tracing</u>	36
<u>Example: Adding tracing support to your plug-in</u>	37
<u>Generating Ant scripts</u>	39
<u>Deploying a plug-in</u>	40
<u>Fragments</u>	44
<u>Example: Writing a German fragment for XYZ Plug-in</u>	44
<u>Features</u>	47

Table of Contents

Fragments

<u>Setting up a feature project</u>	47
<u>Example: Setting up a feature for plug-ins and fragments</u>	48
<u>Feature manifest editor</u>	48
<u>Synchronizing versions</u>	51
<u>UI driven synchronization</u>	51
<u>Automatic synchronization at build time</u>	52
<u>Deploying a Feature</u>	52
<u>Build configuration</u>	54
<u>Generating Ant scripts from the command line</u>	56
<u>Directory file format</u>	57
<u>Using the targets</u>	58
<u>Working with update sites</u>	58
<u>Setting up an update site project</u>	58
<u>Example: Setting up an update site project</u>	58
<u>Building plug-ins, fragments and features using update site editor</u>	59
<u>Previewing an update site</u>	61
<u>Example: building and previewing the sample update site</u>	61
<u>Using extension point schema</u>	62
<u>Example: Using the "Sample Parsers" extension point</u>	62

<u>Converting existing projects into PDE projects</u>	64
--	-----------

<u>Reference</u>	65
-------------------------------	-----------

<u>PDE Extension Points</u>	66
--	-----------

<u>Extension Wizards</u>	67
---------------------------------------	-----------

<u>Plug-in Content Wizards</u>	71
---	-----------

<u>Extension Templates</u>	74
---	-----------

<u>Other Reference Information</u>	77
---	-----------

<u>Map of PDE Plug-ins</u>	78
---	-----------

<u>PDE Dynamic Classpaths FAQ</u>	78
---	----

<u>Tips and Tricks</u>	80
------------------------------	----

<u>What's New in 3.0</u>	82
--------------------------------	----

PDE Guide

- Getting Started
 - ◆ [Introduction to PDE](#)
 - ◆ [Preparing the workbench](#)
 - ◆ [Concepts](#)
 - ◆ Basic Plug-in Tutorial
 - ◇ [Creating a plug-in project](#)
 - ◇ [Plug-in manifest editor](#)
 - [Overview page](#)
 - [Dependencies page](#)
 - [Runtime page](#)
 - [Extensions page](#)
 - [Extension points page](#)
 - [Build configuration page](#)
 - [Source page](#)
 - ◇ [Running a plug-in](#)
 - [Choosing plug-ins to run](#)
 - [Running with tracing](#)
 - ◇ [Deploying a plug-in](#)
 - ◆ [Fragments](#)
 - ◇ [Fragment example](#)
 - ◆ [Features](#)
 - ◇ [Setting up a feature project](#)
 - ◇ [Feature manifest editor](#)
 - ◇ [Synchronizing versions](#)
 - ◇ [Deploying a feature](#)
 - ◇ Advanced topics in building features
 - [Build configuration](#)
 - [Generating Ant scripts from PDE](#)
 - [Generating Ant scripts from scripts](#)
 - ◆ [Update Sites](#)
 - ◇ [Setting up an update site project](#)
 - ◇ [Building plug-ins and features using update site editor](#)
 - ◇ [Previewing update sites](#)
- Tasks
 - ◆ [Creating an extension point schema](#)
 - ◇ [Extension point schema editor](#)
 - ◇ [Using extension point schema](#)
 - ◆ [Converting existing projects](#)
- [Reference](#)
 - ◆ [Dynamic classpaths FAQ](#)
- [Tips and tricks](#)
- [What's new](#)
- [Legal](#)

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.

© Copyright IBM Corporation and others 2000, 2004.

Notices

The material in this guide is Copyright (c) IBM Corporation and others 2000, 2004.

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

About This Content

20th June, 2002

License

Eclipse.org 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 Common Public License Version 1.0 ("CPL"). A copy of the CPL is available at <http://www.eclipse.org/legal/cpl-v10.html>. For purposes of the CPL, "Program" will mean the Content.

Contributions

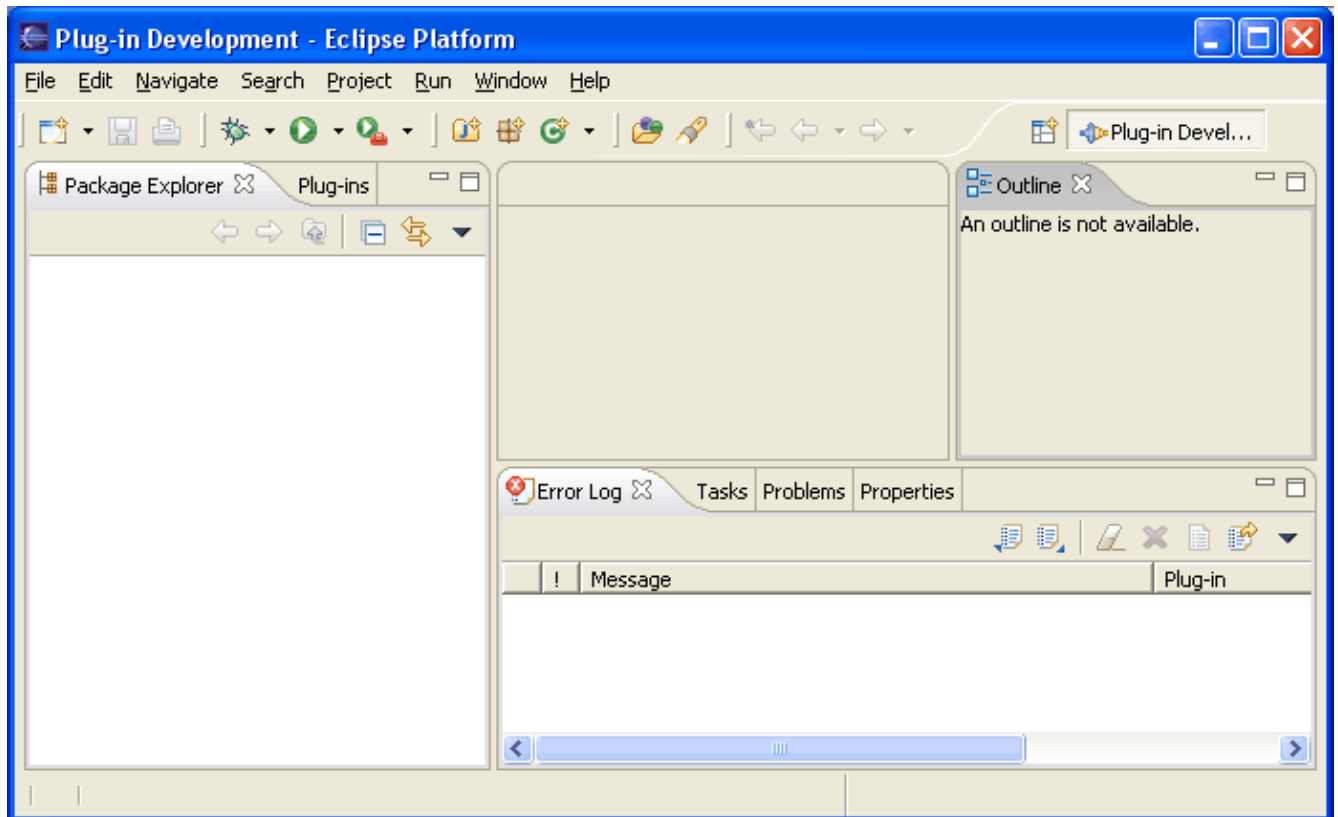
If this Content is licensed to you under the terms and conditions of the CPL, any Contributions, as defined in the CPL, uploaded, submitted, or otherwise made available to Eclipse.org, members of Eclipse.org and/or the host of Eclipse.org web site, by you that relate to such Content are provided under the terms and conditions of the CPL and can be made available to others under the terms of the CPL.

If this Content is licensed to you under license terms and conditions other than the CPL ("Other License"), any modifications, enhancements and/or other code and/or documentation ("Modifications") uploaded, submitted, or otherwise made available to Eclipse.org, members of Eclipse.org and/or the host of Eclipse.org, by you that relate to such Content are provided under terms and conditions of the Other License and can be made available to others under the terms of the Other License. In addition, with regard to Modifications for which you are the copyright holder, you are also providing the Modifications under the terms and conditions of the CPL and such Modifications can be made available to others under the terms of the CPL.

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.

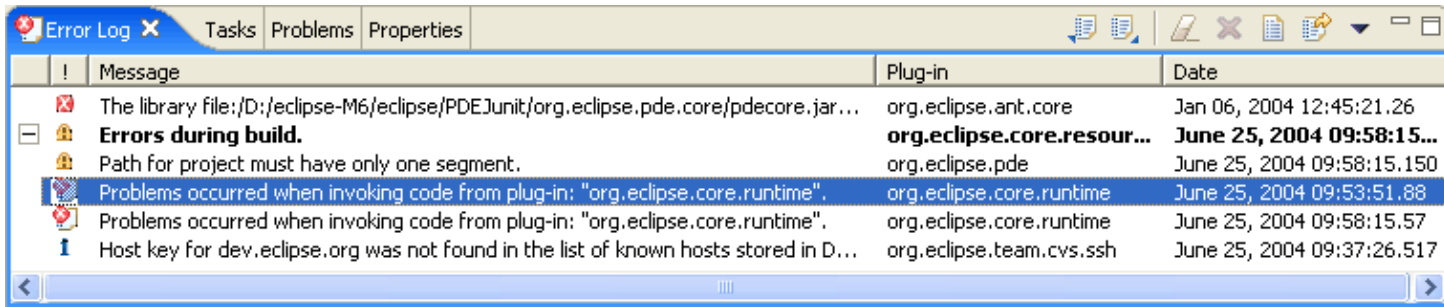
Error Log

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.

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.

Using the Plug-in Development Environment

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
The library file.:D:/eclipse-M6/eclipse/PDEJunit/org.eclipse.pde.core/pdecore.jar...	org.eclipse.ant.core	Jan 06, 2004 12:45:21.26
Errors during build.	org.eclipse.core.resour...	June 25, 2004 09:58:15...
Path for project must have only one segment.	org.eclipse.pde	June 25, 2004 09:58:15.150
Problems occurred when invoking code from plug-in: "org.eclipse.core.runtime".	org.eclipse.core.runtime	June 25, 2004 09:53:51.88
Problems occurred when invoking code from plug-in: "org.eclipse.core.runtime".	org.eclipse.core.runtime	June 25, 2004 09:58:15.57
Host key for dev.eclipse.org was not found in the list of known hosts stored in D...	org.eclipse.team.cvs.ssh	June 25, 2004 09:37:26.517

© Copyright IBM Corporation and others 2000, 2004.

Concepts

Host vs. runtime

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

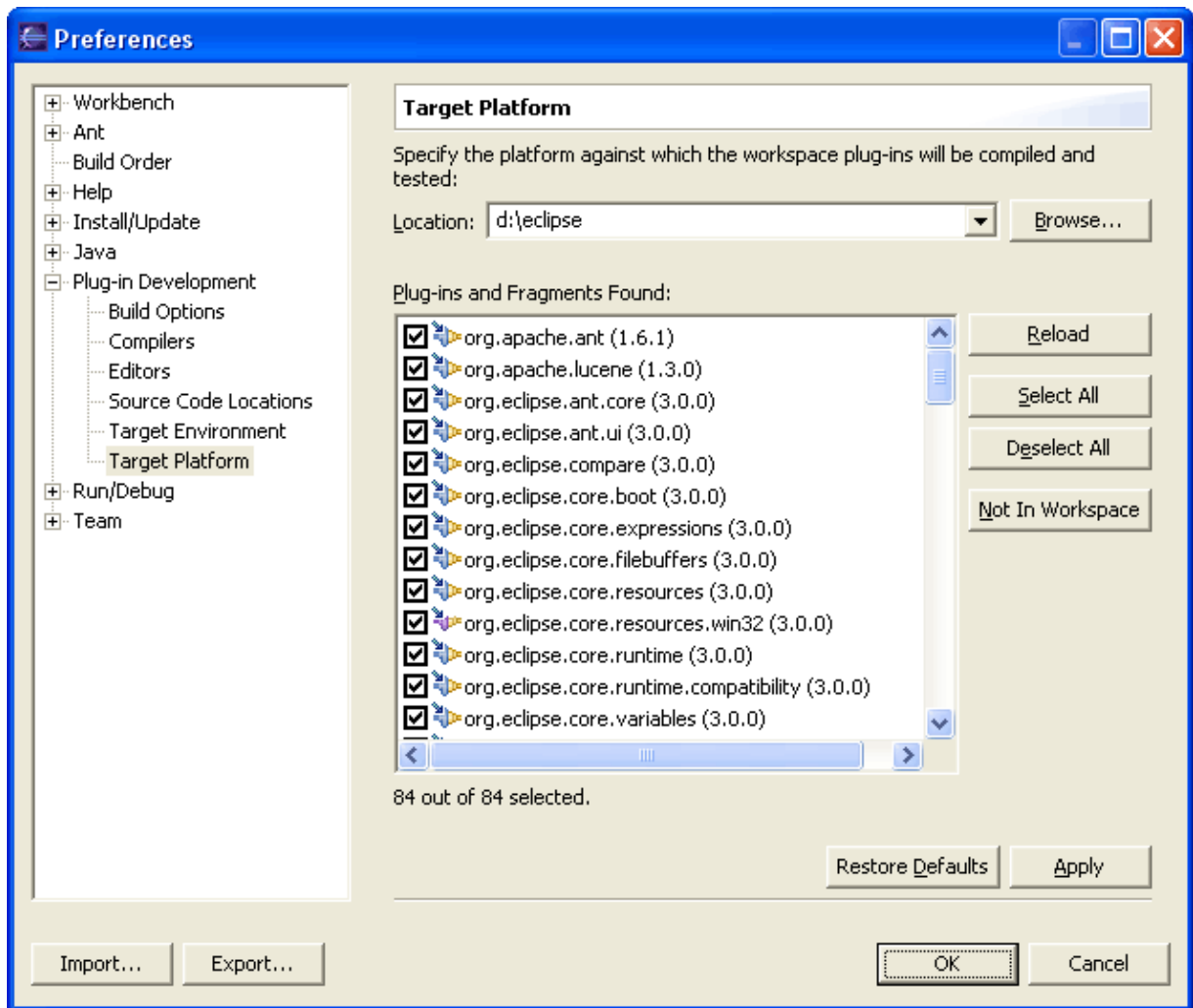
When you start up the workbench, you will use it to work on your projects that define the plug-ins you are building. The workbench instance that you are running as you develop your plug-in using the PDE and other tools is the **host** instance. The features available in this instance will come exclusively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

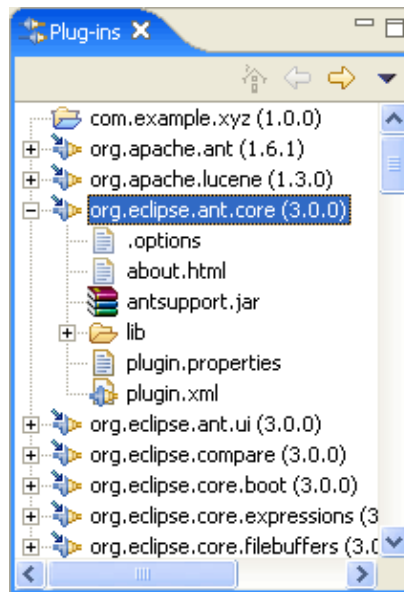
External vs. workspace plug-ins

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.

Using the Plug-in Development Environment

The Plug-ins view, which is part of the PDE perspective, will show the combined list of workspace and external plug-ins. In it, you will be able to browse the directory structure of external plug-ins, open files, etc.

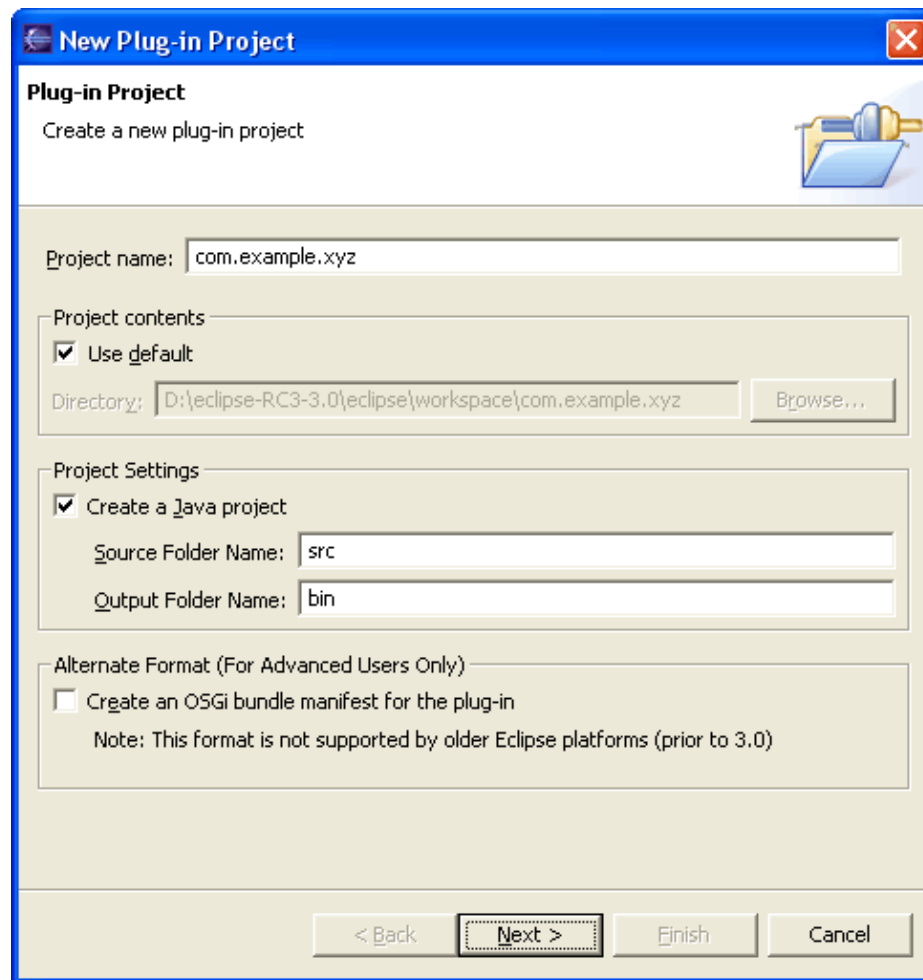


© Copyright IBM Corporation and others 2000, 2004.

Creating a plug-in project

In the workspace, a plug-in is represented by a single project that has a `plugin.xml` (manifest) file at its root and 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.

The 3.0 runtime supports a new plug-in format, where the plug-in content is split between the traditional `plugin.xml` file and an OSGi bundle manifest file. This new format is NOT required for plug-ins created in 3.0. It is optional and only recommended for advanced users.

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:

Runtime Library:

Plug-in Class

Generate the Java class that controls the plug-in's life cycle (recommended)

Class Name:

This plug-in will make contributions to the UI

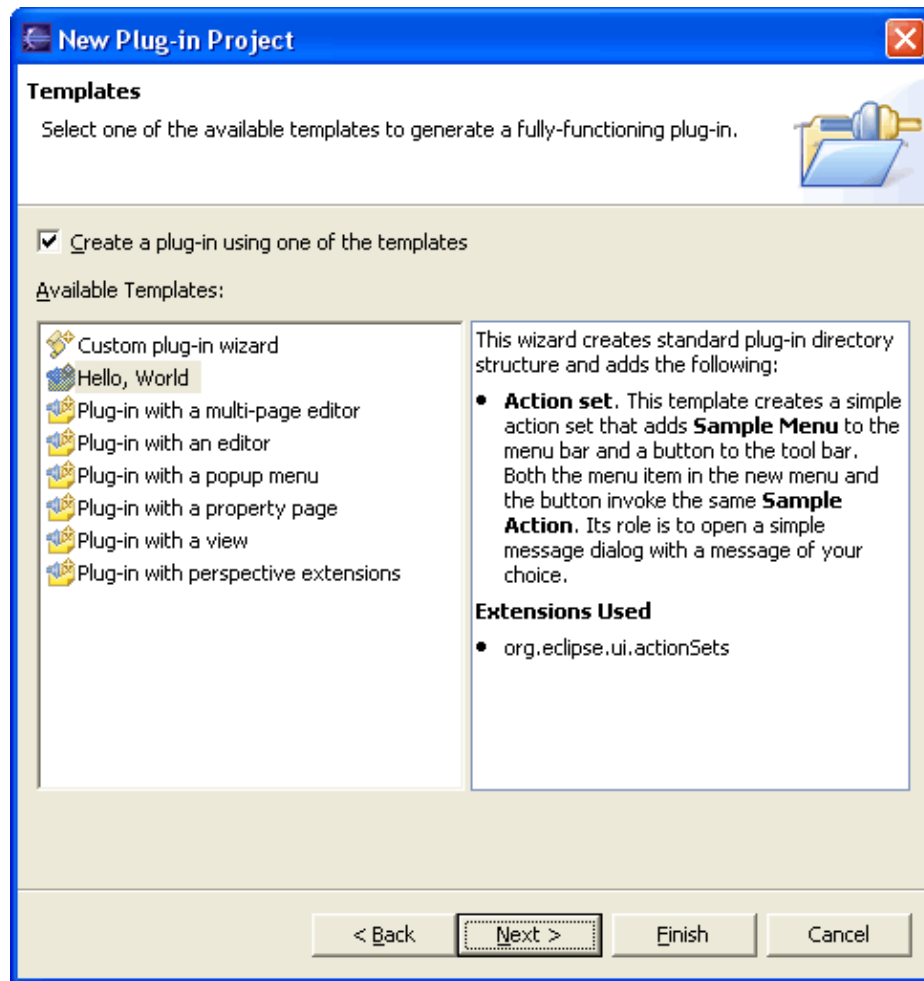
Intended for use with older Eclipse platforms (prior to 3.0)

< 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 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. It can also be used as a central place to create and access the resource bundle that can hold your plug-in's locale-specific objects, such as translatable strings.

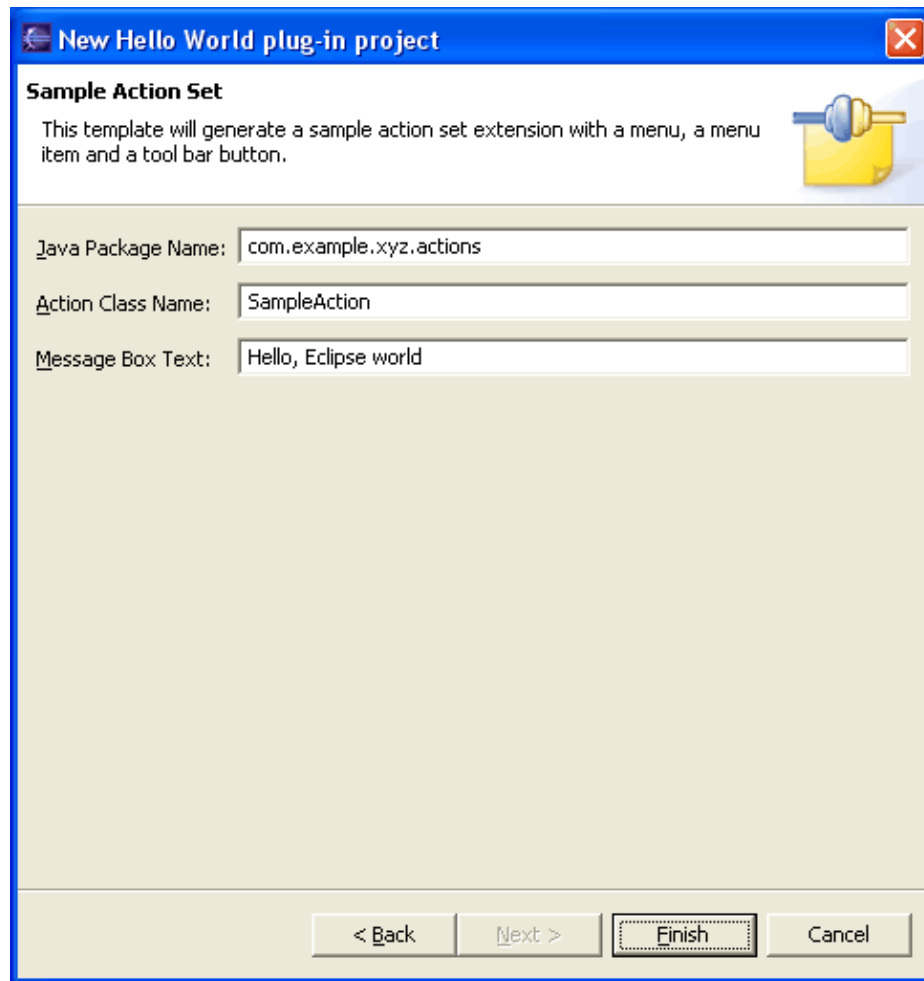
Using the Plug-in Development Environment



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

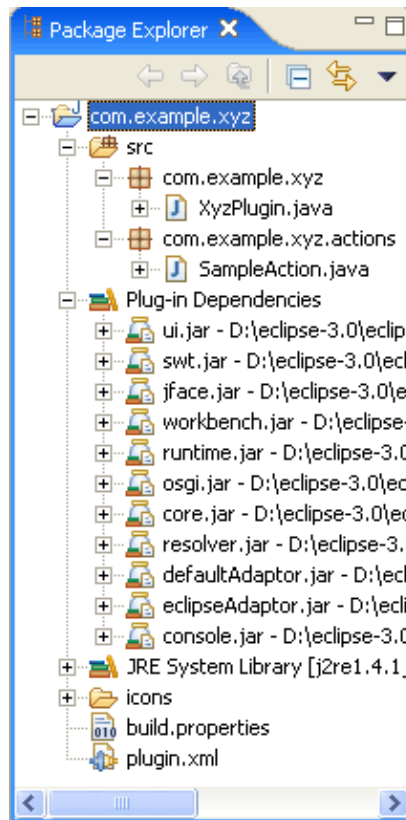


The next page will let you customize the sample extension that we are creating.

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 so that you can define additional attributes of your plug-in.

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

Using the Plug-in Development Environment



© Copyright IBM Corporation and others 2000, 2004.

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 (plugin.xml, build.properties and manifest.mf).

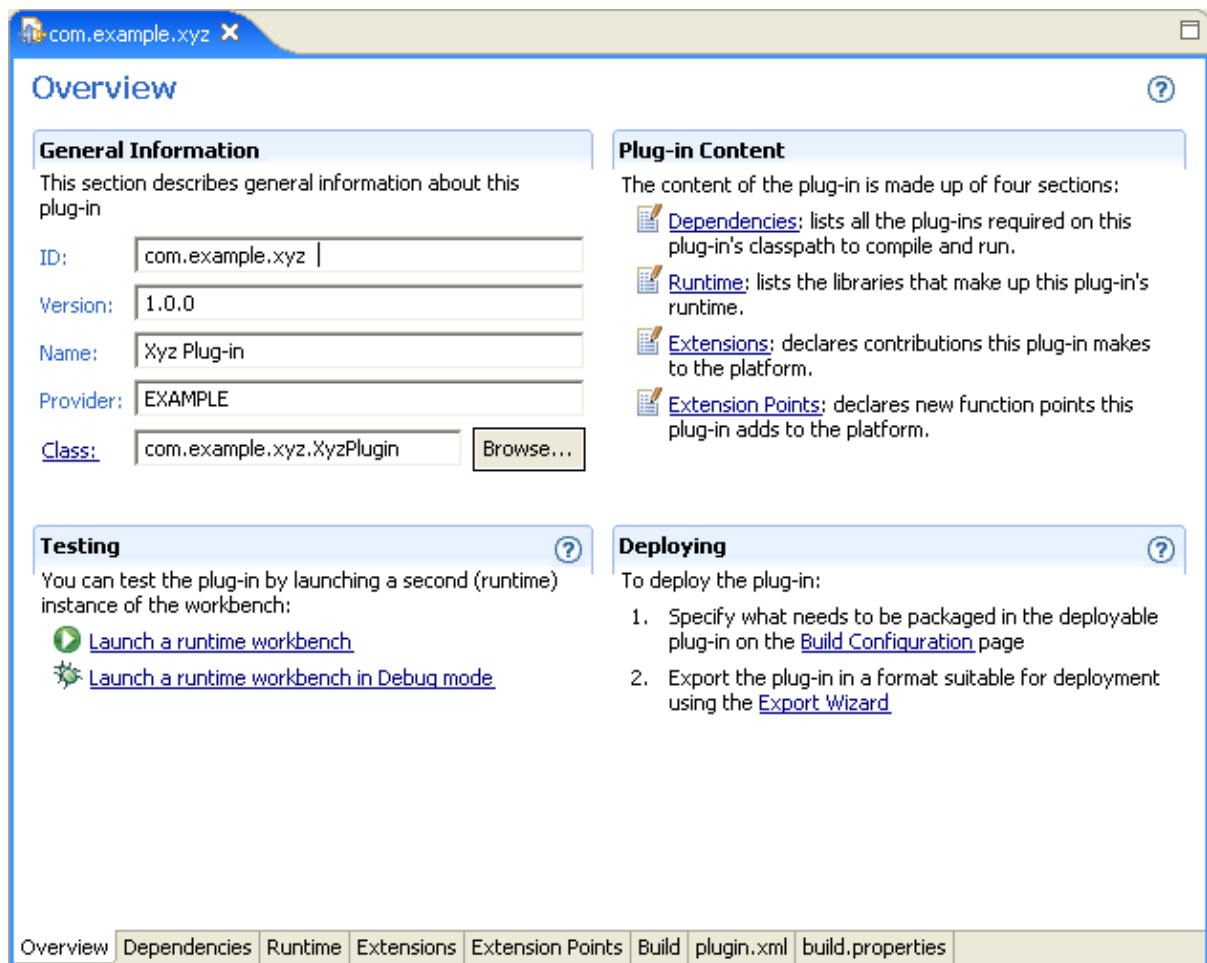
When you use the editor's forms, PDE transparently handles the task of writing the changes to the right files.

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

© Copyright IBM Corporation and others 2000, 2004.

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

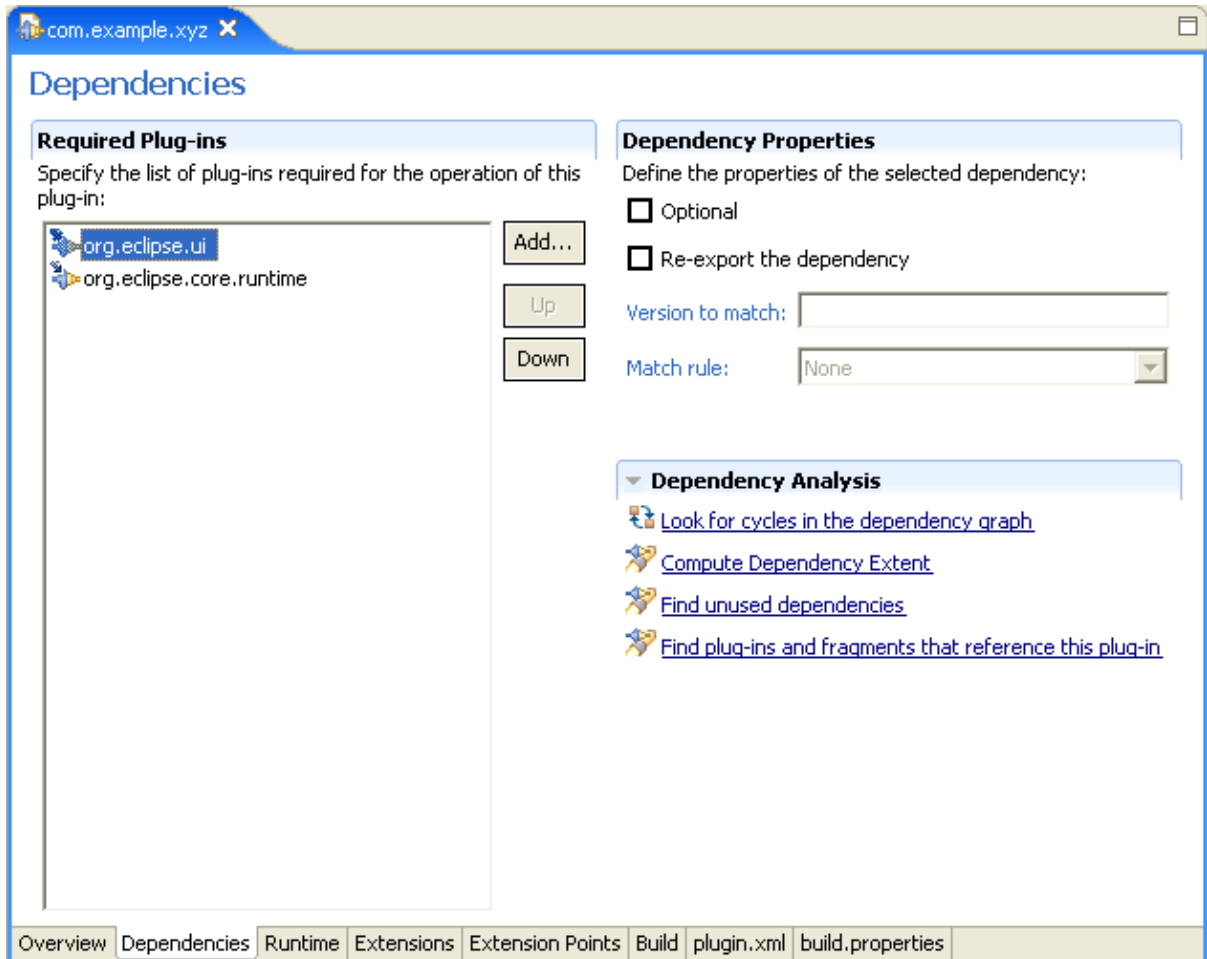
The Testing section provides shortcuts to quickly launch a runtime workbench to test and debug the plug-in.

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

© Copyright IBM Corporation and others 2000, 2004.

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. You must also list all plug-ins that contribute extension points that your plug-in uses. 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.

When you select an entry in the list of required plug-ins, you can mark the dependency as re-exported in the **Dependency Properties** section. 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.

Using the Plug-in Development Environment

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 **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.

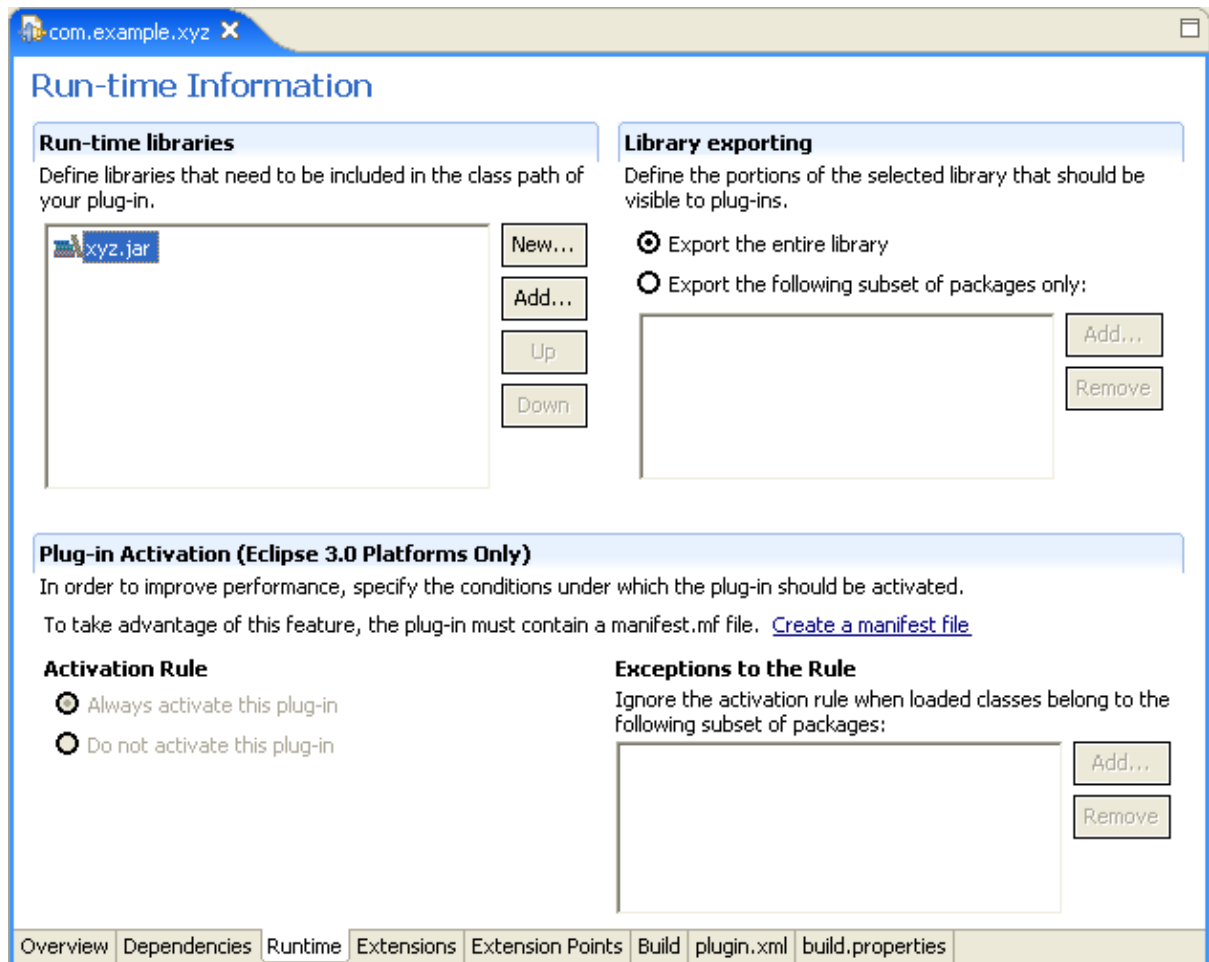
For a selected plug-in in the list, **Compute Dependency Extent** will give you a list of all the Java types and all the extension points that your plug-in needs from that dependency. So, in essence, it tells you why you need that plug-in.

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.

© Copyright IBM Corporation and others 2000, 2004.

Runtime page

The Runtime page shows information about run-time libraries. When packaged, platform plug-ins deliver all of their Java classes in JAR libraries.



You can also determine exporting rules for your libraries. By default, no classes in your libraries can be seen by others. This is appropriate if your plug-in is not meant to be extended. If your classes should be visible to other plug-ins, the libraries must be exported. You can export the entire library or just a subset of packages.

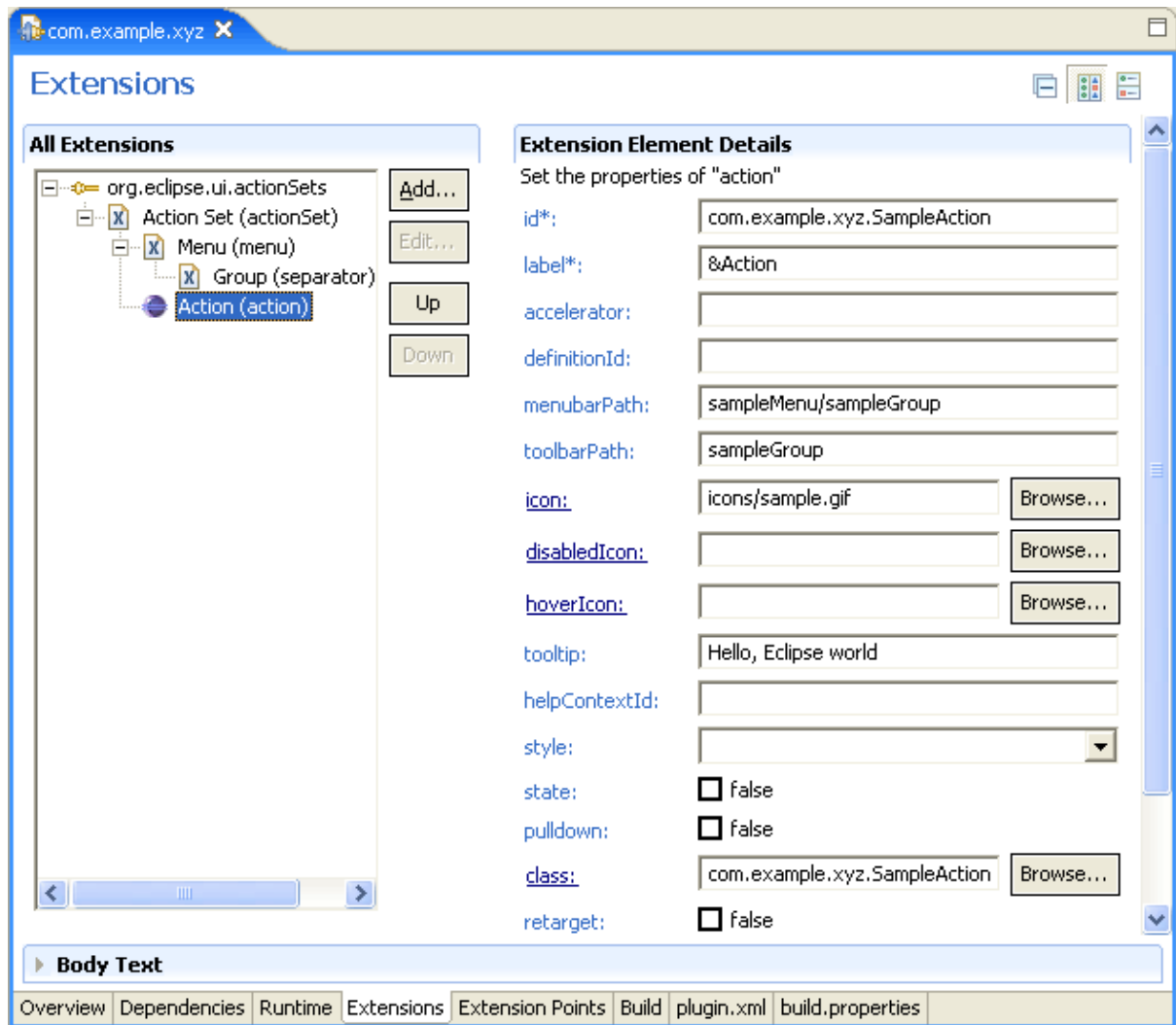
The first time a class is accessed from one of your plug-in's libraries at runtime, the plug-in gets activated. In 3.0, this activation is selective. You can specify the conditions when your plug-in should be activated at runtime, thus potentially reducing overhead and memory footprint. This functionality is only available for plug-ins that have an OSGi bundle manifest.mf file. Therefore, if you want to take advantage of this functionality, PDE provides a link in the **Plug-in Activation** section that will create a manifest.mf file for your plug-in with all the correct contents on the fly.

© Copyright IBM Corporation and others 2000, 2004.

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.

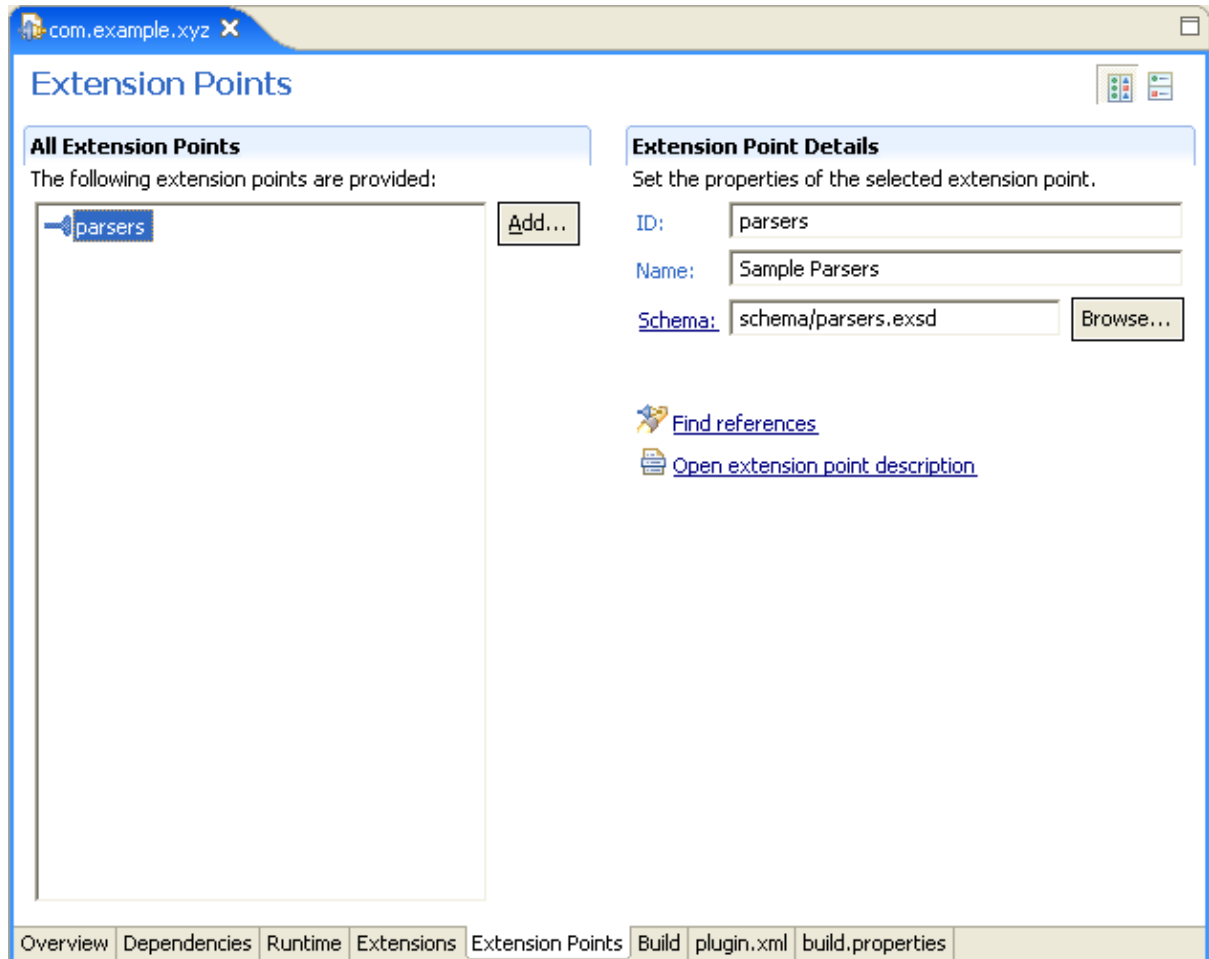
When you hover your mouse over an attribute name, a tooltip box appears describing the purpose of that attribute.

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.

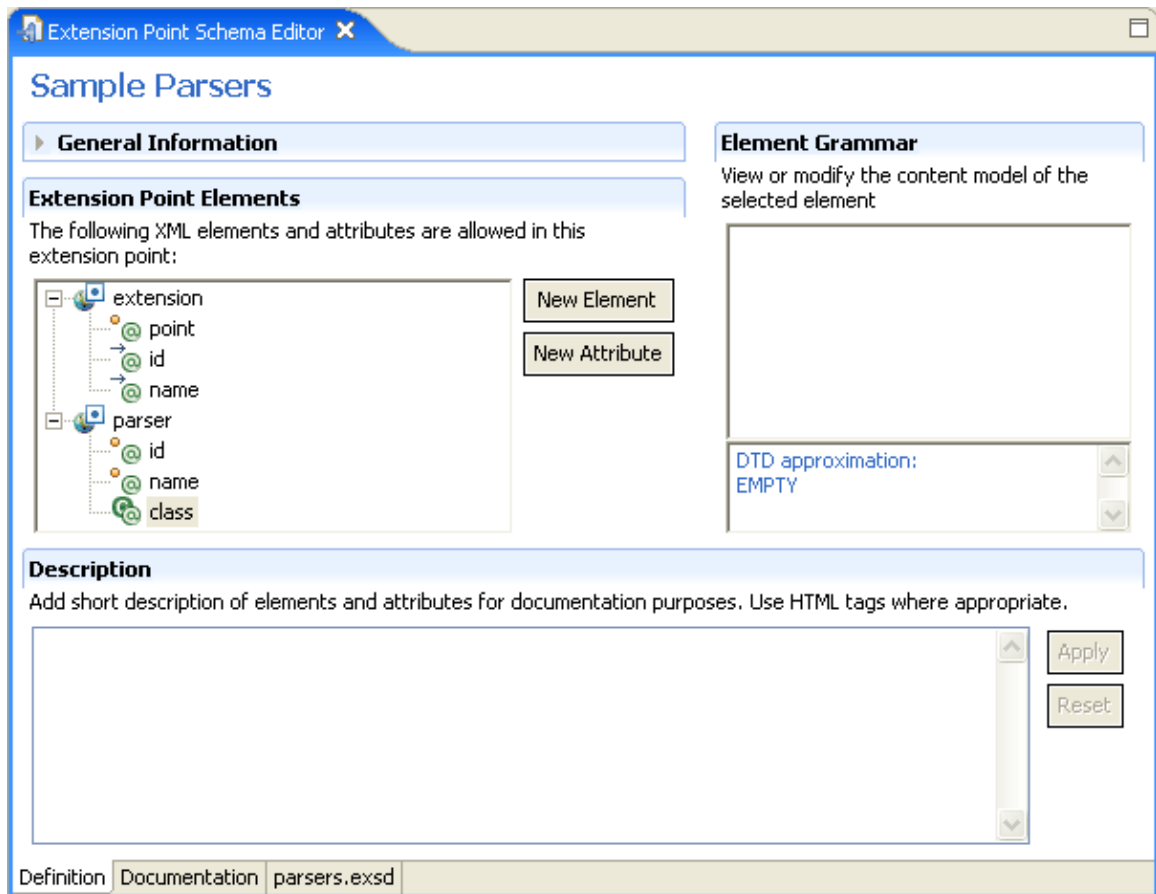
We want to do the following:

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:

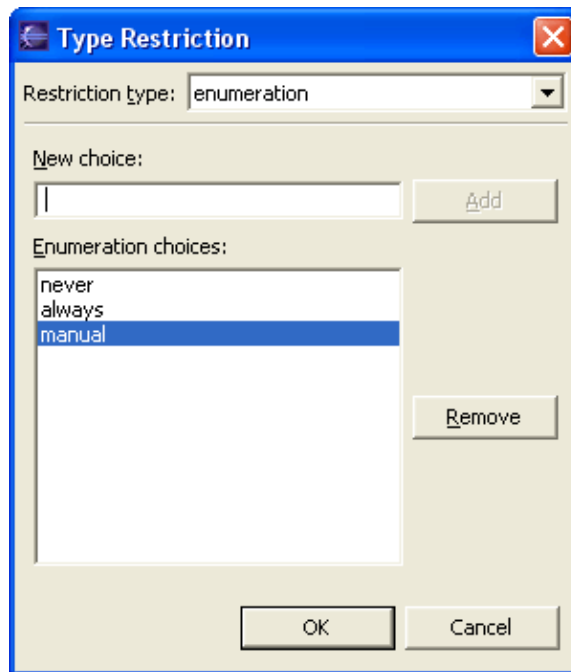


We will now add an additional attribute that takes values from a discrete list of choices. This means we need to create an enumeration restriction of the base **string** type. In addition, we will set a default value for the attribute.

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



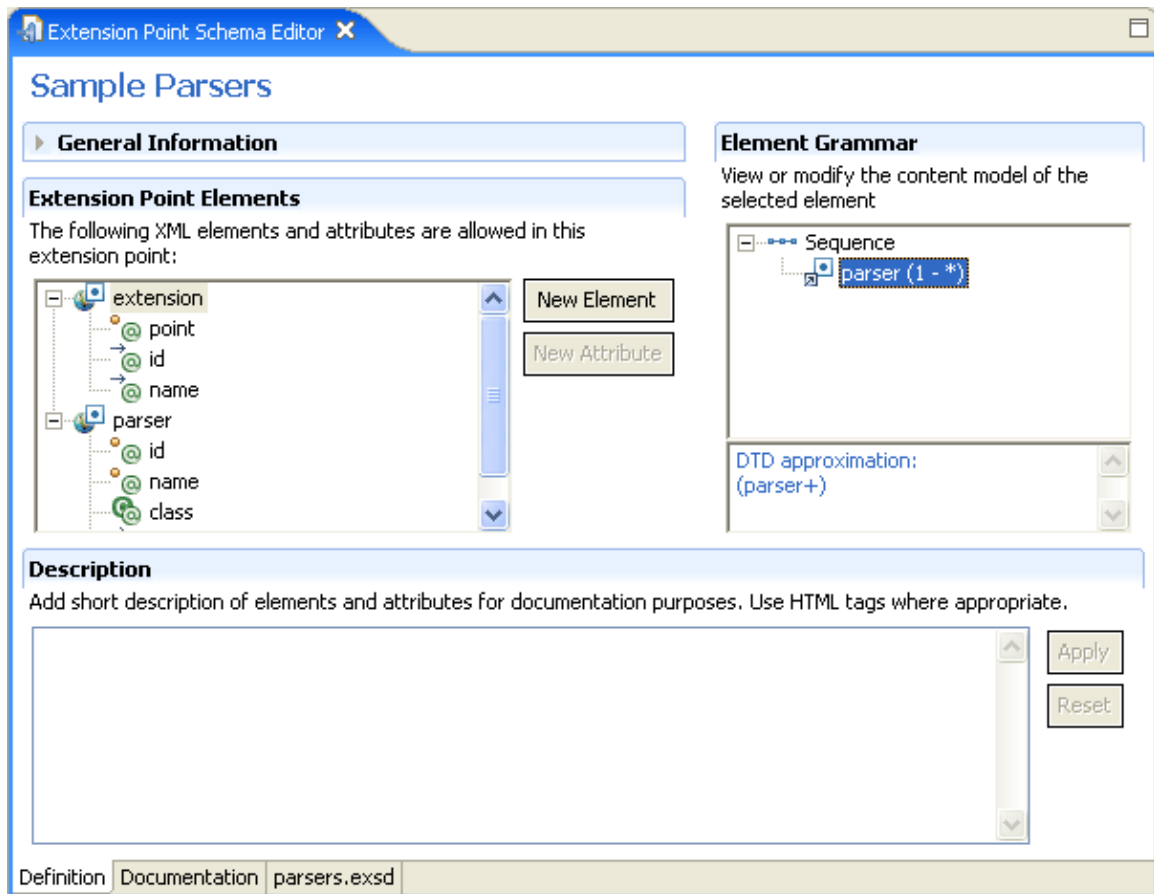
When the dialog is closed, change the "use" attribute to "default" and "value" attribute to "always." This establishes the default value. Note that the status line shows an error message as you are typing the value, since it restricts valid values to the three enumeration choices. Once you finish typing,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

Using the Plug-in Development Environment



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:

Using the Plug-in Development Environment

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

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 `com.example.xyz.IParser` interface.

Press **Apply**.

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

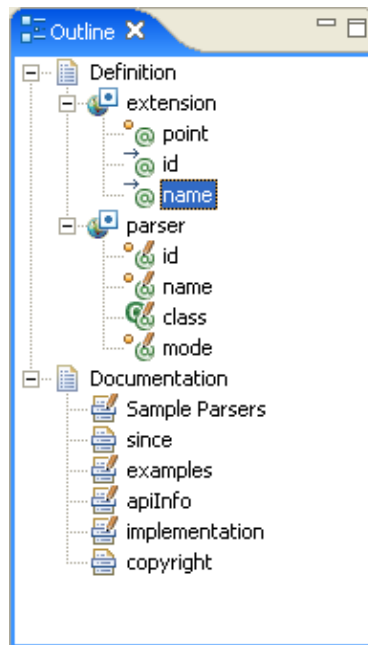
XYZ Plugin provides default implementation of the parser.

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](#).

© Copyright IBM Corporation and others 2000, 2004.

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 (manual|always|never) >
```

- **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=  
"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.

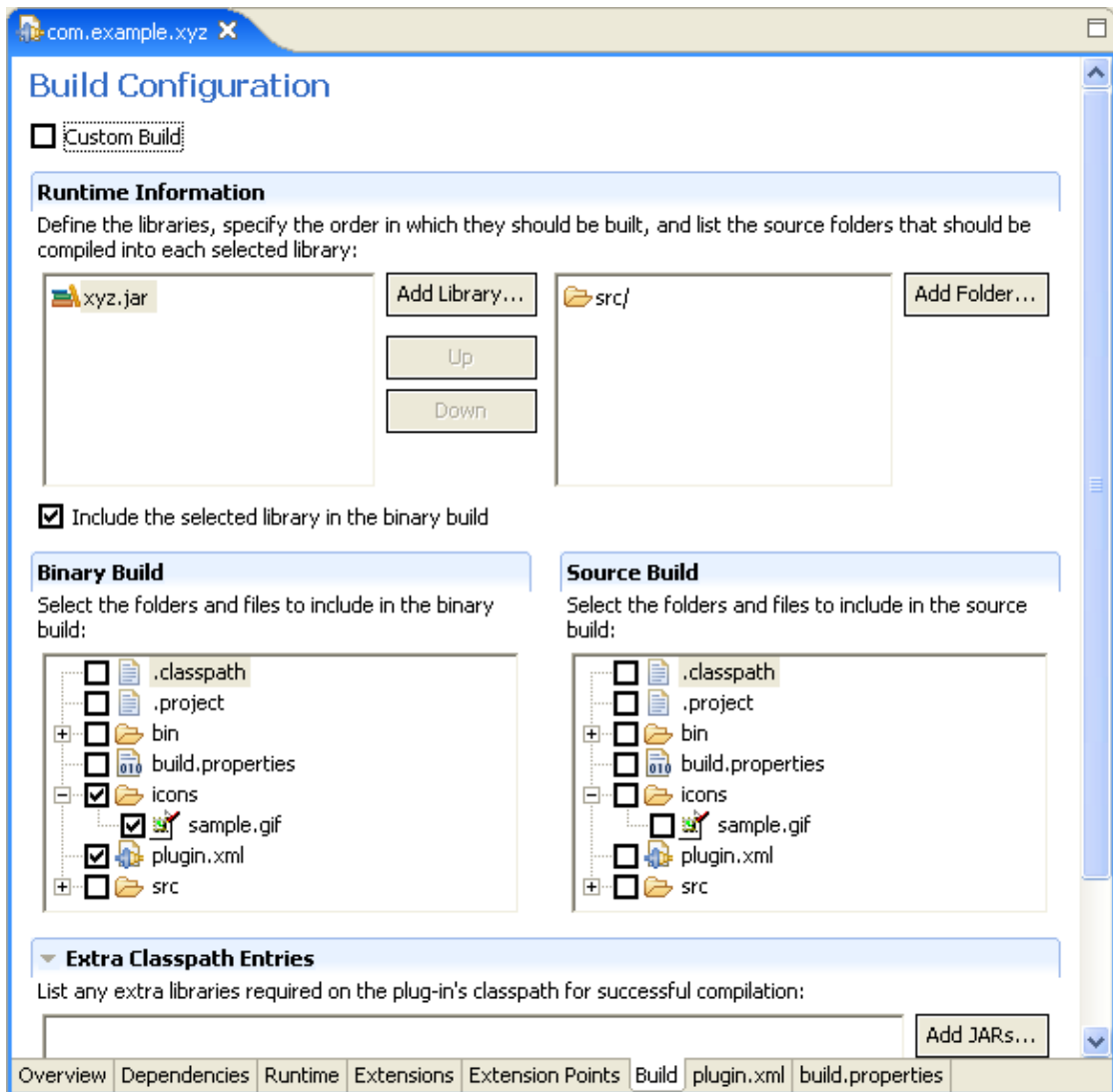
Supplied Implementation:

XYZ Plugin provides default implementation of the parser.

Copyright XYZ 2003. All rights reserved.

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.

Using the Plug-in Development Environment

© Copyright IBM Corporation and others 2000, 2004.

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. In addition, it should contain any file or directory specified in the build.properties using `source.include` 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.

Using the Plug-in Development Environment

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.

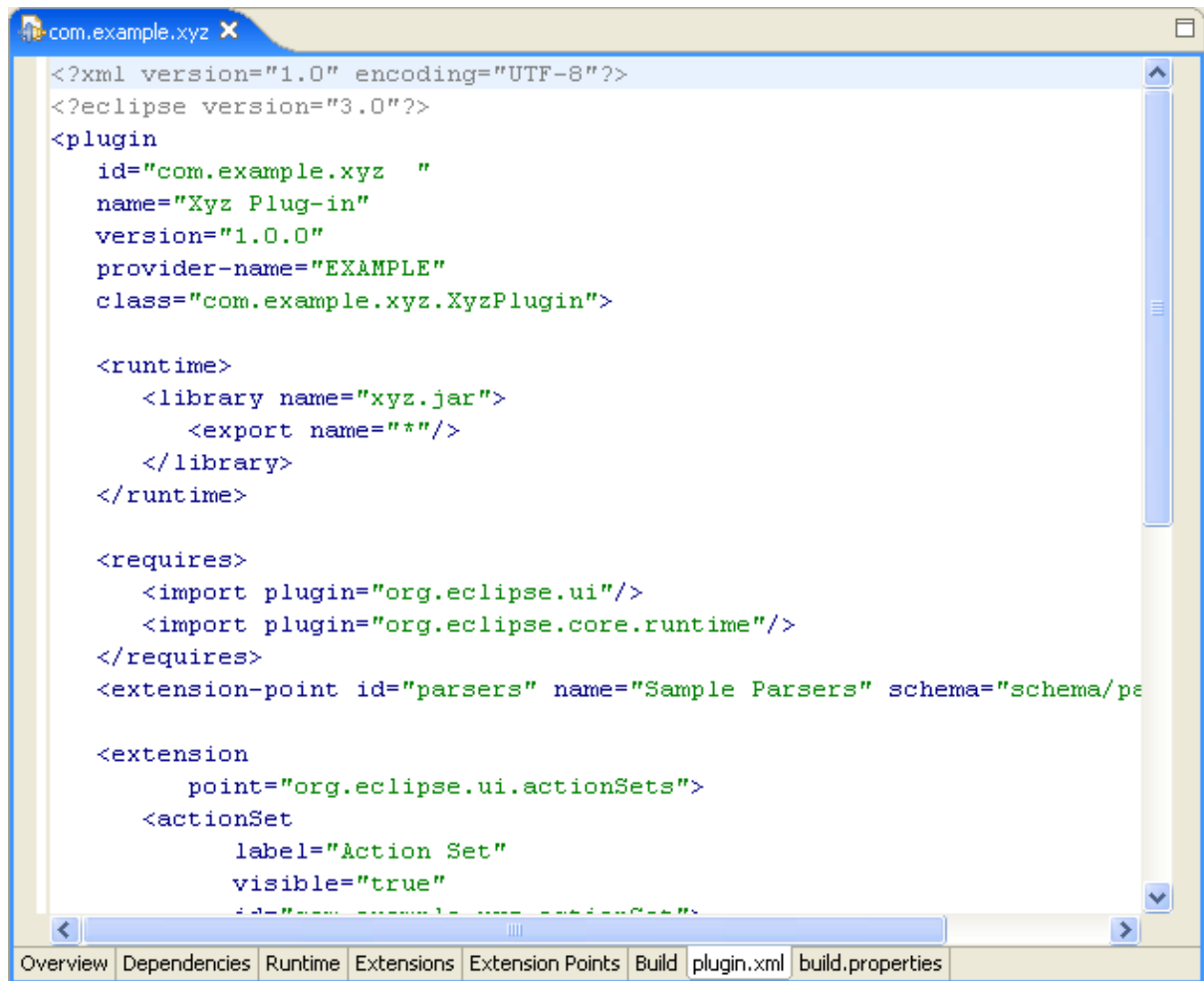
All rights reserved. This program and the accompanying materials are made available under the terms of the Common Public License v1.0 which accompanies this distribution, and is available at <http://www.eclipse.org/legal/cpl-v10.html>.

Source page

In addition to the GUI pages, PDE provides a source page, where users who prefer to edit the `plugin.xml` file by hand, can work.

Since manual editing of the file can open the door to a wide variety of errors, both syntactic and semantic, PDE validates the file upon saving or switching tabs to flag such errors.

Using the Plug-in Development Environment



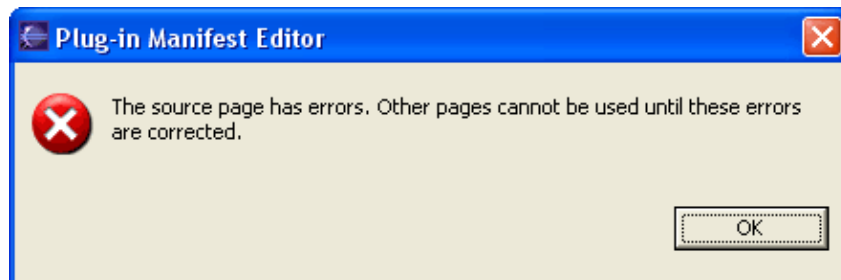
```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin
  id="com.example.xyz  "
  name="Xyz Plug-in"
  version="1.0.0"
  provider-name="EXAMPLE"
  class="com.example.xyz.XyzPlugin">

  <runtime>
    <library name="xyz.jar">
      <export name="*" />
    </library>
  </runtime>

  <requires>
    <import plugin="org.eclipse.ui"/>
    <import plugin="org.eclipse.core.runtime"/>
  </requires>
  <extension-point id="parsers" name="Sample Parsers" schema="schema/pa

  <extension
    point="org.eclipse.ui.actionSets">
    <actionSet
      label="Action Set"
      visible="true"
      id="new.example.org.actionSet">
```

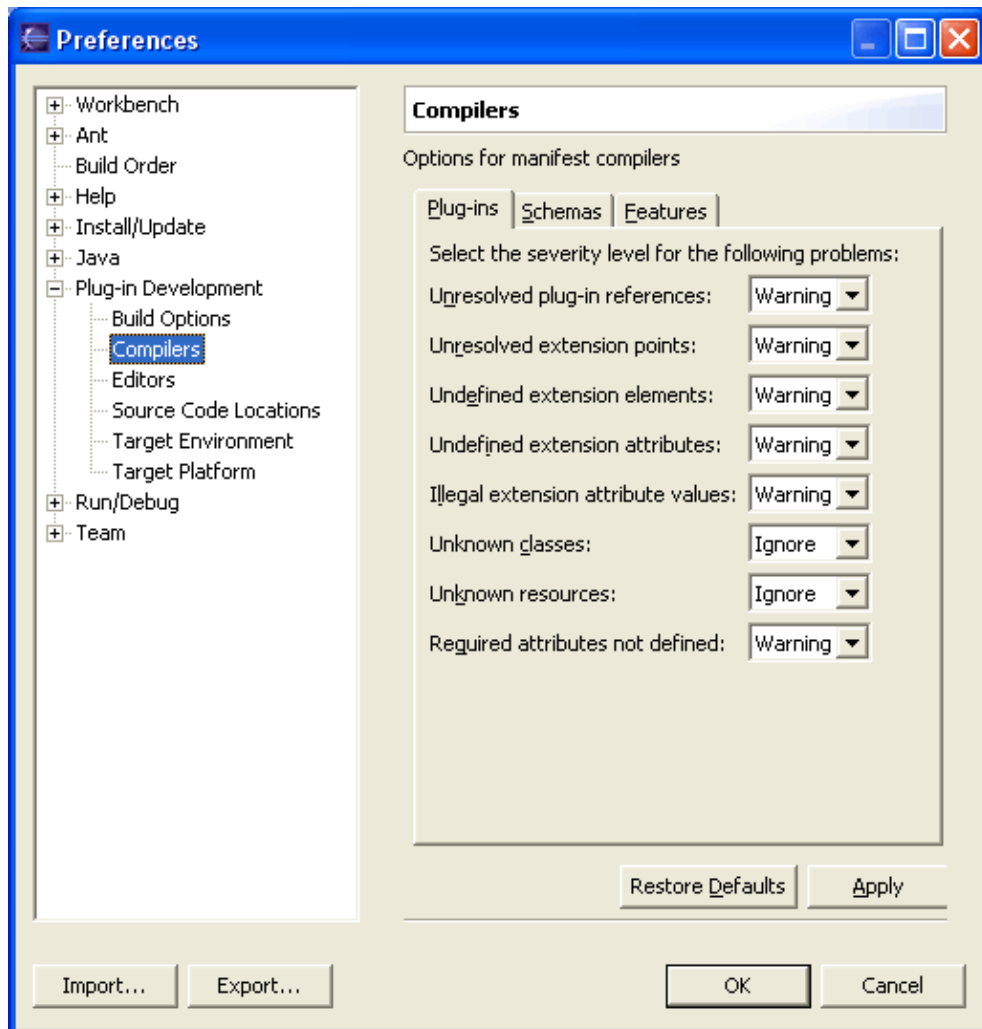
If a syntax error is introduced, PDE will lock out the form pages until the error is fixed. If you try to switch to any of the non-source pages, you will see the following dialog:



If you close the editor with syntactic errors in it and open it again, the editor will open into the Source page and other pages will not be accessible until the source is fixed.

Using the Plug-in Development Environment

The semantic errors, which PDE is able to detect, are listed on the **Plug-ins** tab of the **Plug-in Development > Compilers** preference page:



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.

© Copyright IBM Corporation and others 2000, 2004.

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->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.

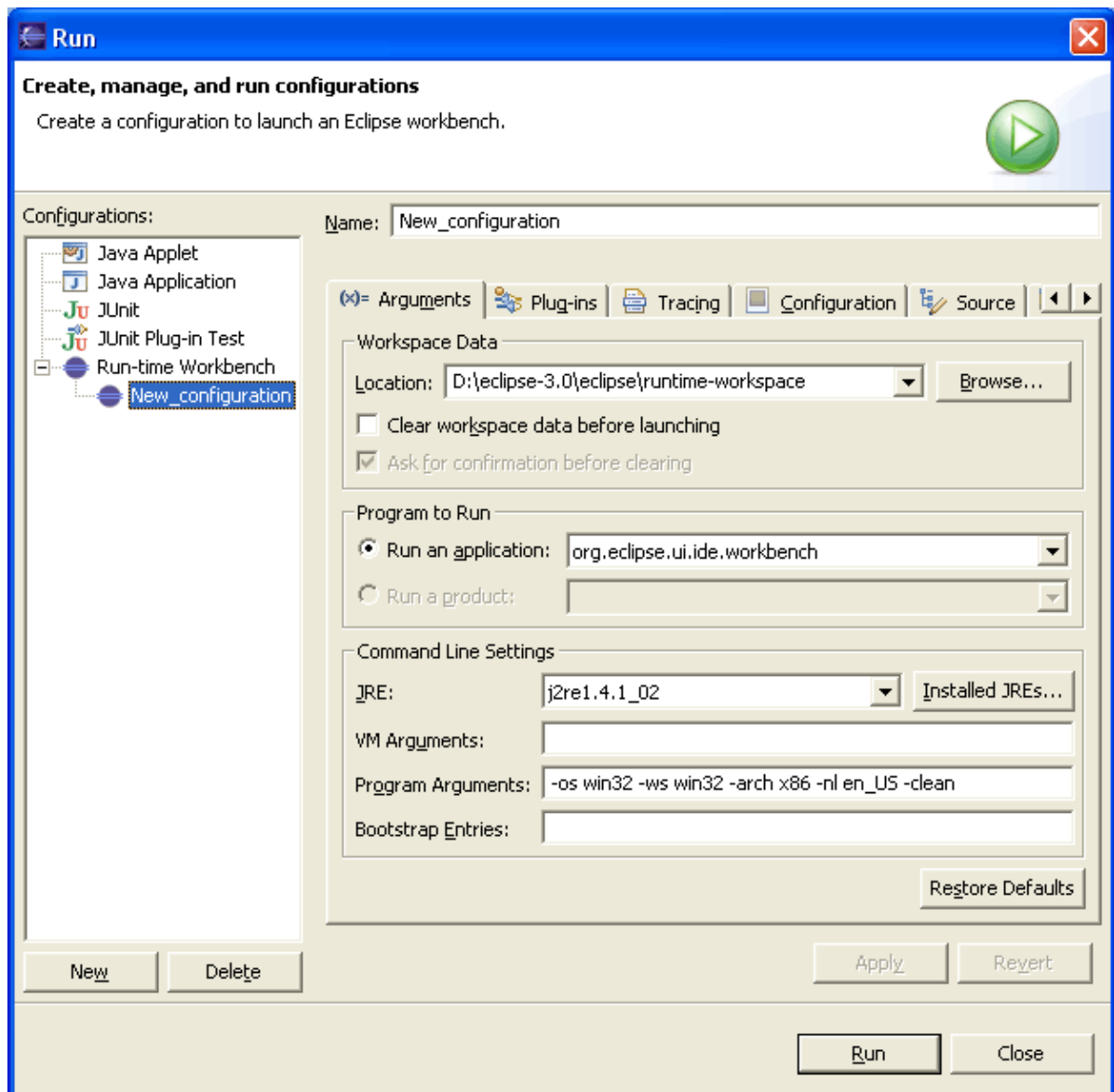
© Copyright IBM Corporation and others 2000, 2004.

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 runtime workbench instance to test your new plug-in.

The easiest way to launch a runtime workbench is via the link in the Testing section of the plug-in manifest editor's Overview page. **Run**→**Run As**→**Run-time Workbench** from the main menu bar. This will immediately create a second (run-time) workbench 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.



You can create several configurations and give them unique names, each one having different parameters. This allows you to test your plug-in under different conditions by simply switching between different configurations.

Using the Plug-in Development Environment

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

The default Eclipse application is *org.eclipse.ui.ide.workbench*. 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.

PDE uses the values specified on the **Plug-in Development > Target Environment** preference page to set the default program arguments for the launch configuration. If you manually change these values in the configuration, changes will only affect that particular configuration. Changes made on the preference page will affect all configurations created thereafter.

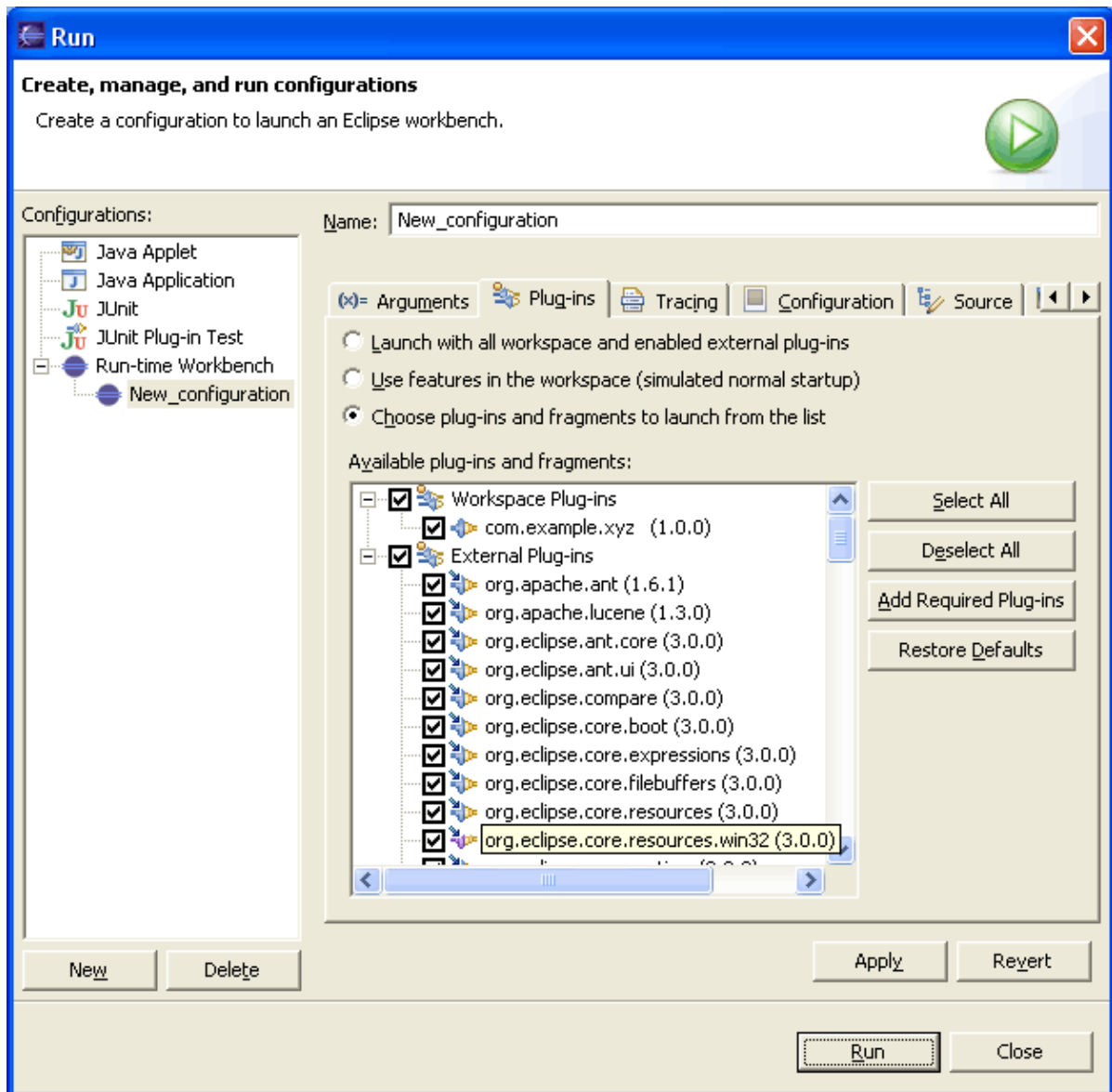
Example: Running the Sample

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".

© Copyright IBM Corporation and others 2000, 2004.

Choosing plug-ins to run

It is possible to alter the list of plug-ins and fragments that PDE will pass to the run-time workbench when running or debugging.



By default, the **Plug-ins** tab is configured to select all the workspace plug-ins and external plug-ins that are enabled in the Preferences.

You also have the option to launch with any subset of those plug-ins. This gives you great flexibility, but with flexibility comes the danger of ending up with an invalid configuration. Use this feature carefully and press **Add Required Plug-ins** to ensure the subset is complete.

© Copyright IBM Corporation and others 2000, 2004.

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>
```

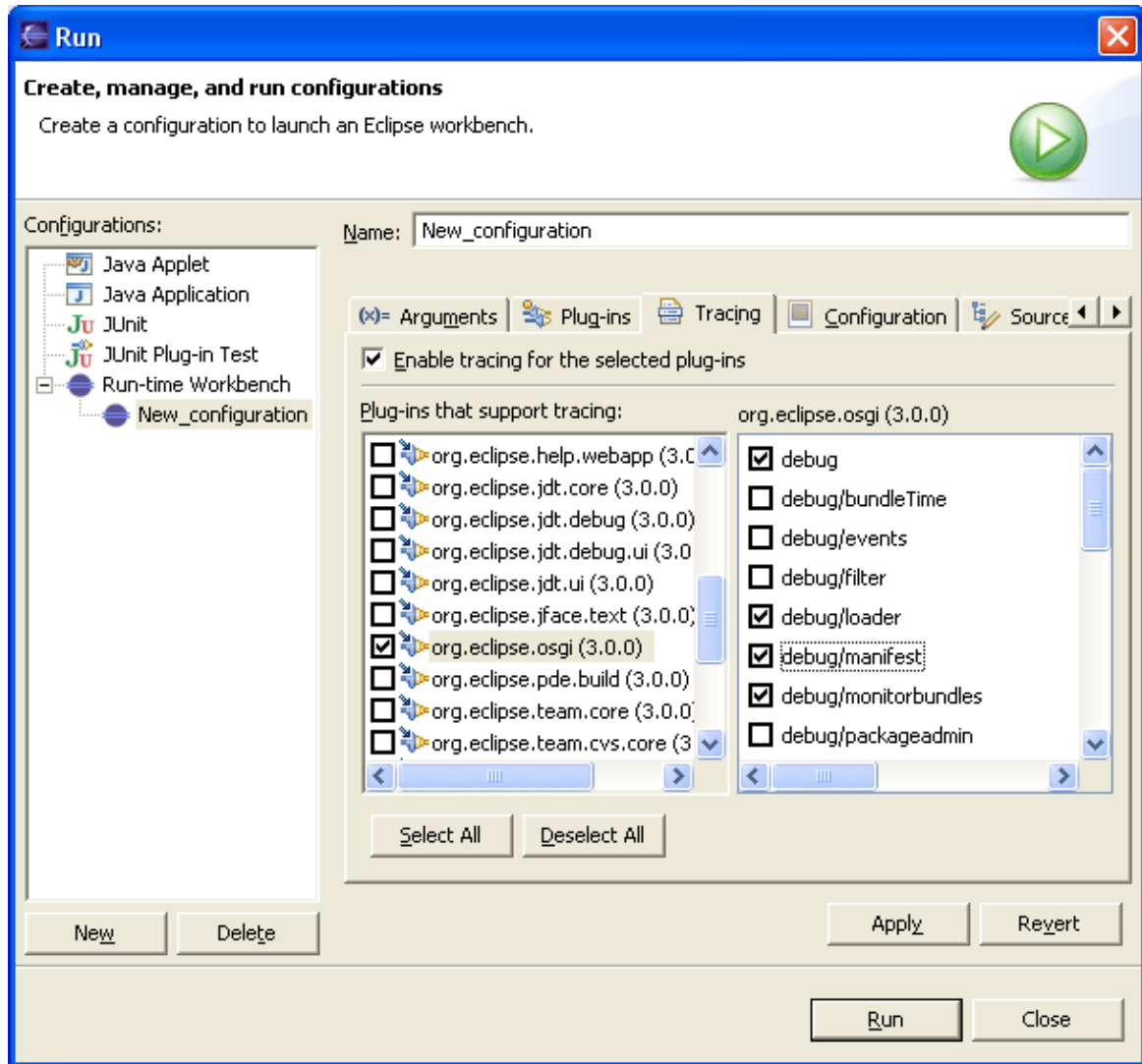
Using the Plug-in Development Environment

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

```
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.

I



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

Example: Adding tracing support to your plug-in

Using the Plug-in Development Environment

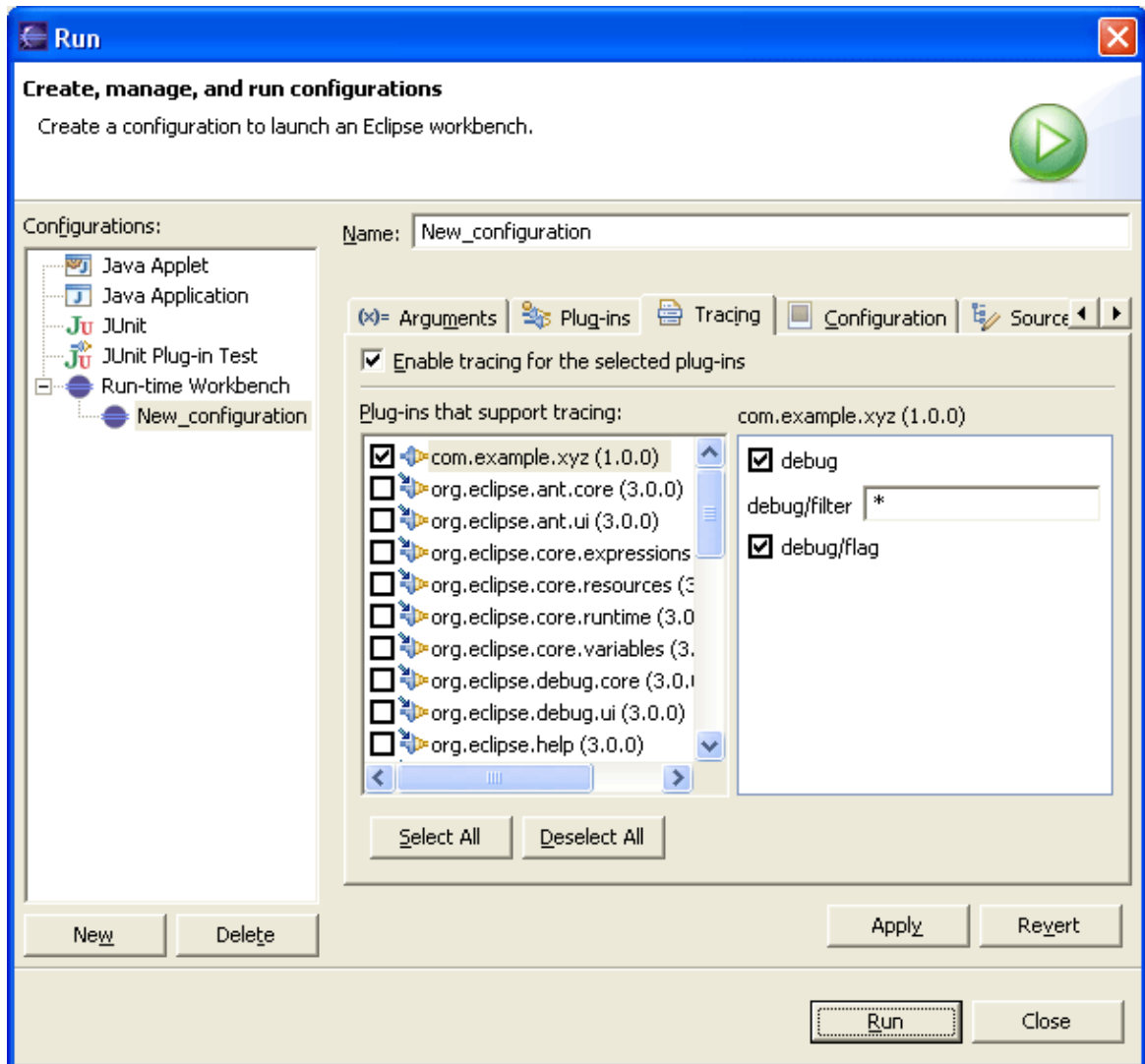
as well as their default values.

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

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()**.

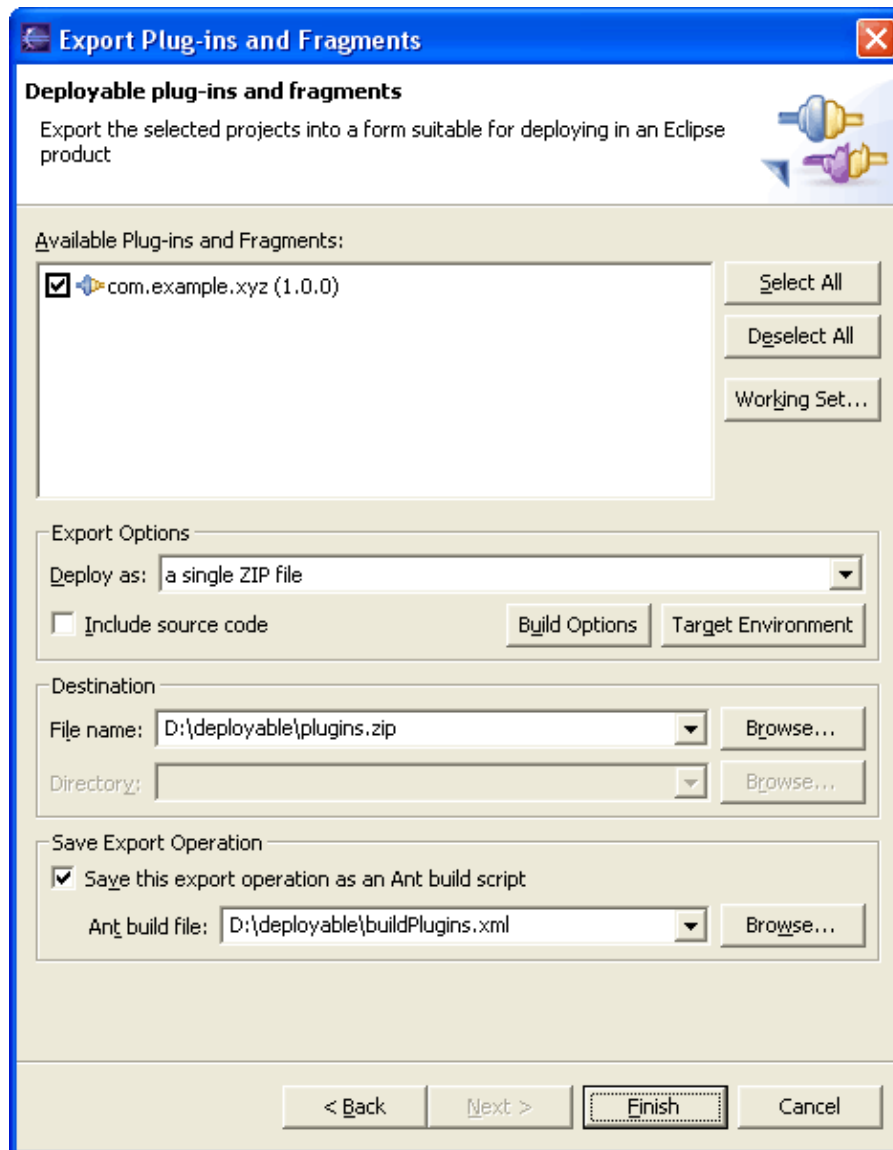
© Copyright IBM Corporation and others 2000, 2004.

Deploying 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 fit for delivery on the platform.

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:

1. Select **File > Export... > Deployable plug-ins and fragments**.
2. Select the plug-ins and fragments you want to export.
3. You can deploy the plug-ins in one of three formats:
 - ◆ a single ZIP file whose content can be unzipped into any Eclipse-based product.
 - ◆ a directory structure, so the destination of the export operation can be the root of an Eclipse installation.
 - ◆ individual JAR archives for an Update site. This option will result in the creation of one JAR per plug-in, and the JARs will be placed in a plugins/ subdirectory of the chosen destination.



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.

© Copyright IBM Corporation and others 2000, 2004.

Generating Ant scripts

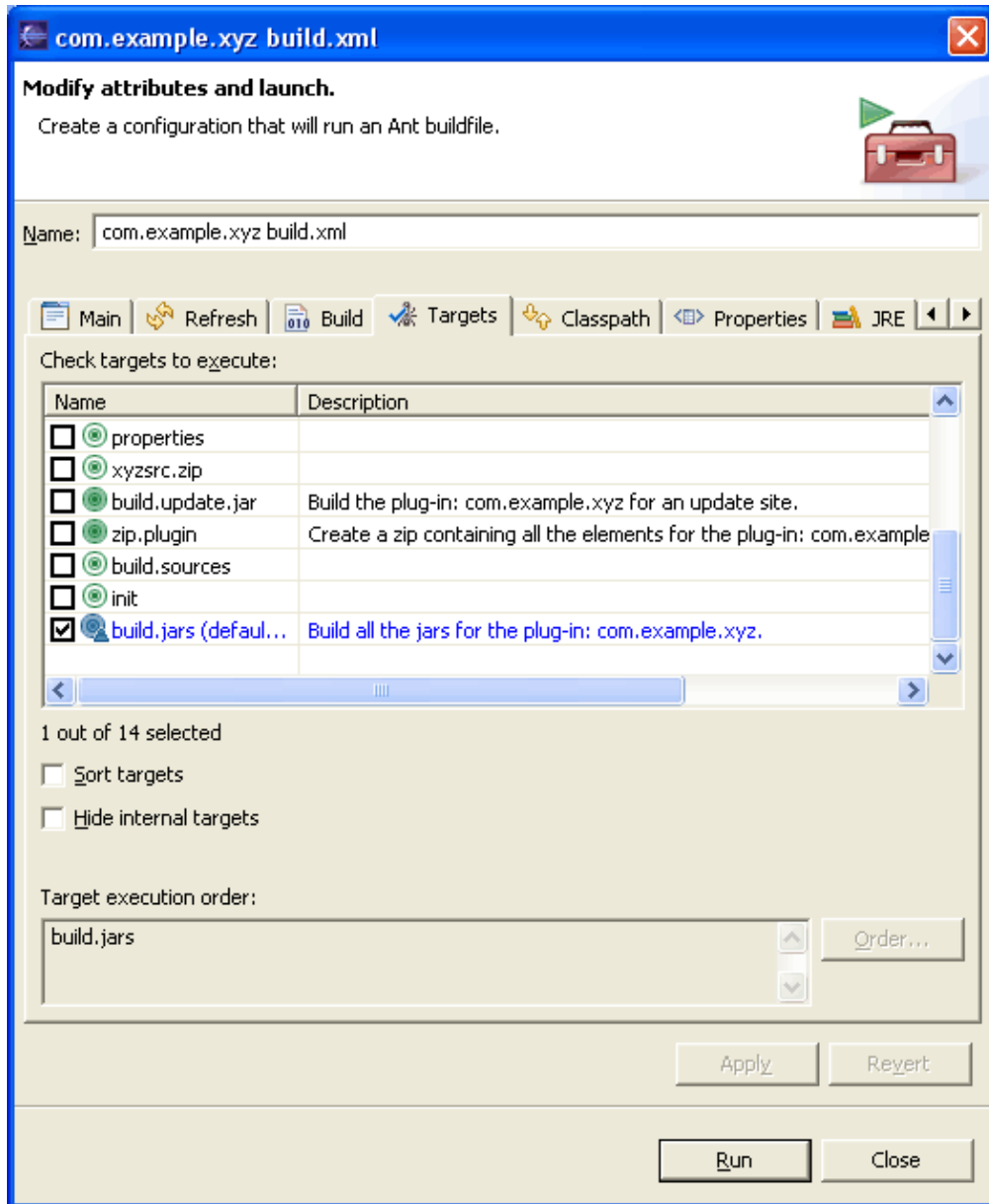
Ant is a simple open-source scripting engine that is capable of running scripts written in XML format. Ant is ideal for executing tasks usually found in automated builds.

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.

Using the Plug-in Development Environment

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 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:

Using the Plug-in Development Environment

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.

© Copyright IBM Corporation and others 2000, 2004.

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.

© Copyright IBM Corporation and others 2000, 2004.

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: parent 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:

Runtime Library:

Parent Plug-in

Plug-in ID:

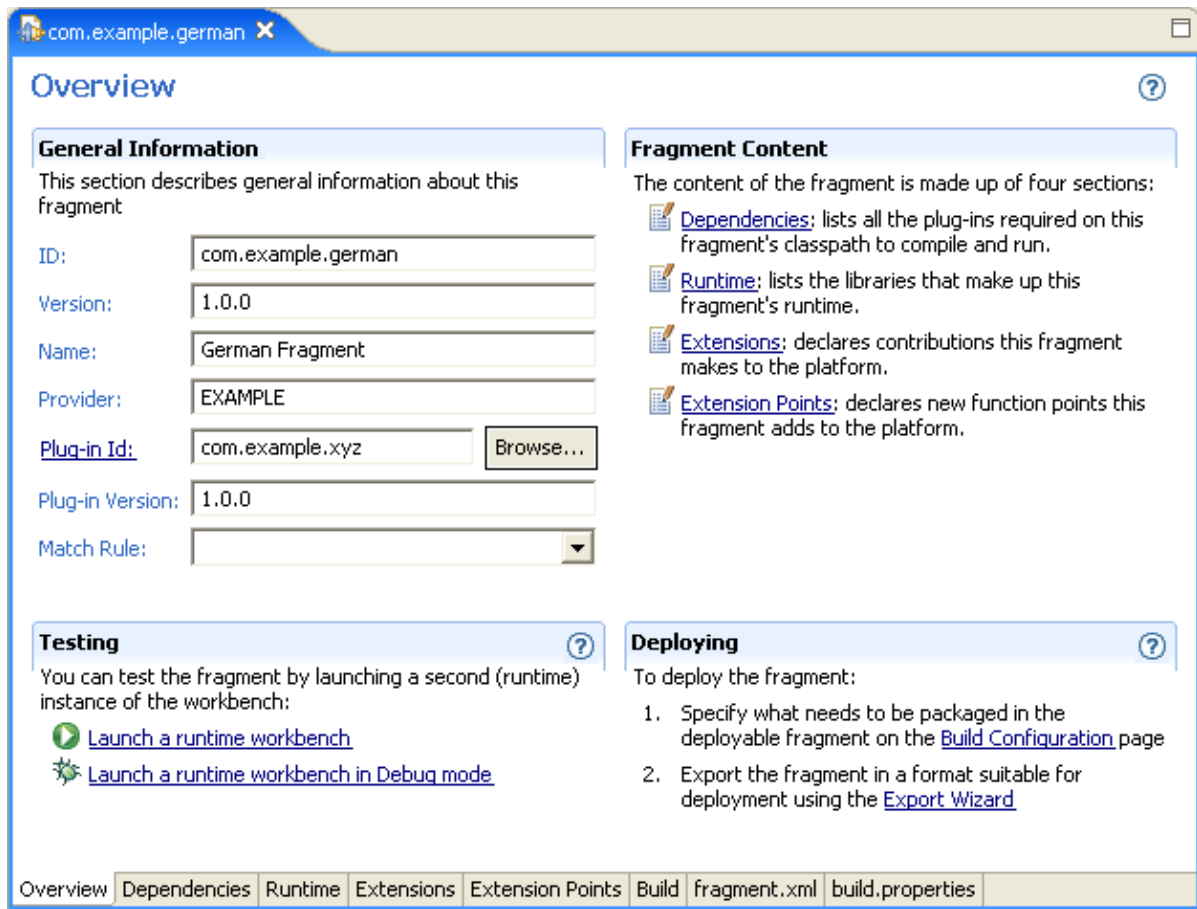
Plug-in Version:

Match Rule:

Intended for use with older Eclipse platforms (prior to 3.0)

< Back Next > Finish Cancel

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



It is almost identical to plug-in manifest editor with the following exceptions:

1. In the Overview page, the "class" attribute is gone. Fragments do not have a plug-in class since they follow the life cycle of their parent plug-in. Instead, the parent plug-in id and version fields are shown.
2. A drop-down combo box labelled "Match Rule" allows for precise definition of the plug-in reference (perfect, compatible, equivalent etc.).

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**.

Using the Plug-in Development Environment

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.

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.

© Copyright IBM Corporation and others 2000, 2004.

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.

© Copyright IBM Corporation and others 2000, 2004.

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.

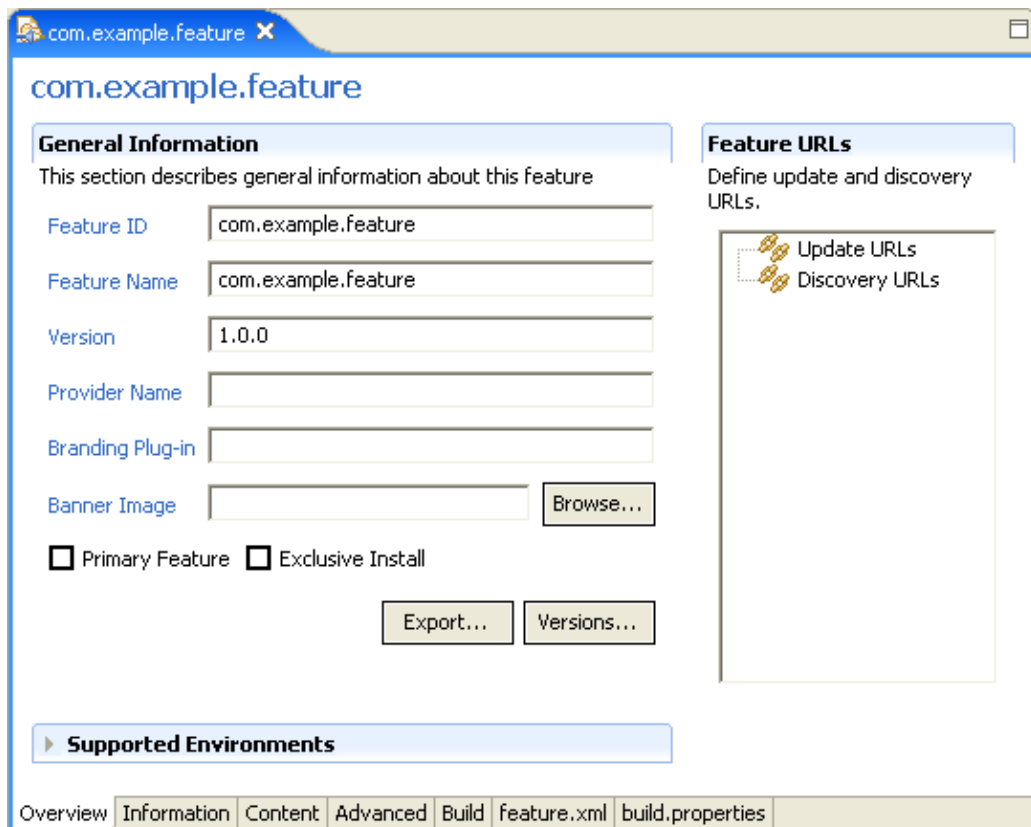
1. Bring up the feature wizard (**New**→**Project**→**Plug-in Development**→**Feature Project**)
2. Set the name of the project to "com.example.feature" and press **Next**.
3. Set the feature name to "Sample Feature" and the feature version to "1.2.2". Set the provider to "Example".
4. In the following 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.

© Copyright IBM Corporation and others 2000, 2004.

Feature manifest editor

The feature manifest editor uses the same concepts seen in the other PDE editors. It has four form pages (Overview, Information, Content and Advanced) and a source page that shows you raw XML code of the manifest file. If build.properties file is present in the project, additional page (Build) shows up allowing you to configure how the feature will be built. Source page for the build.properties file will also appear in this case.



Using the Plug-in Development Environment

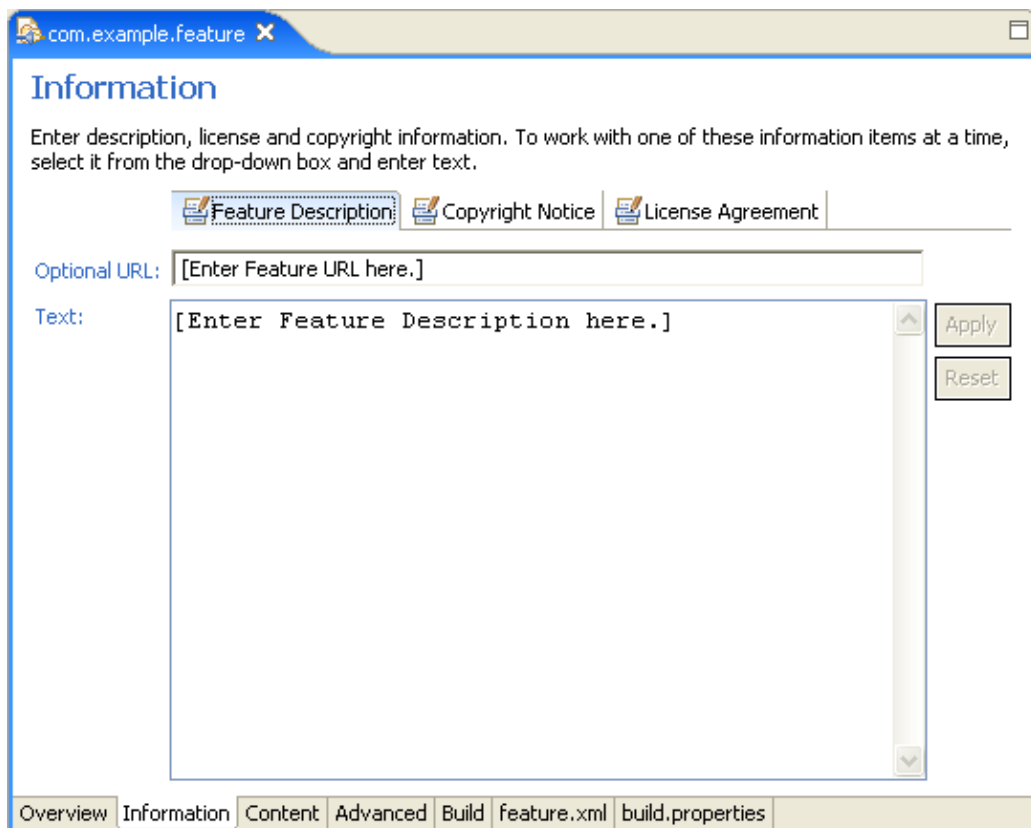
Information that is entered during the feature project setup can be changed on the Overview page. In addition, you can provide URLs for **update site** and **discovery sites** in the property sheet. The update site URL is used by the Update Manager when searching for new updates. The discovery site URLs are used to point users to other interesting features and/or sites.

You can provide your own banner image that will be used in the Update Manager when users browse your feature. If you mark the feature as 'primary', it will be treated as a 'product' feature and will have additional responsibilities (providing 'About' information, splash screen etc.). Finally, you can require an exclusive install for your feature if it should be take part in batch installation (several features at once).

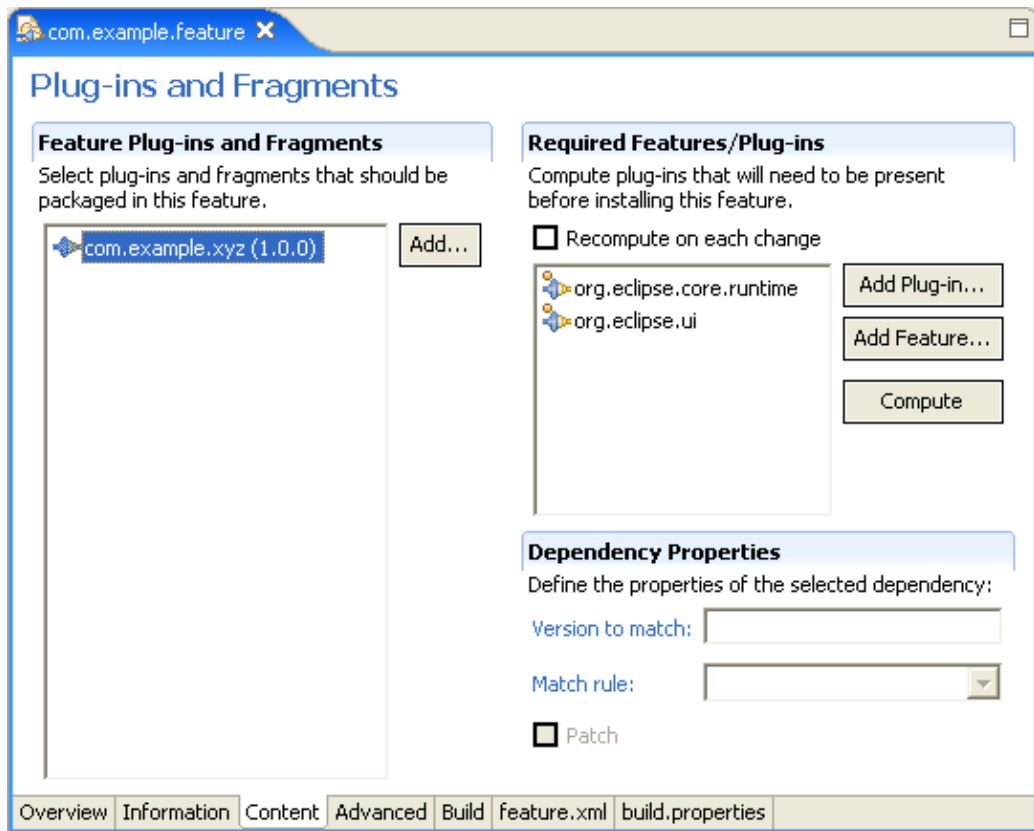
Branding information for primary features is stored in a branding plug-in. If not explicitly set, Eclipse will assume that the branding plug-in has the same identifier as the feature.

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.

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.



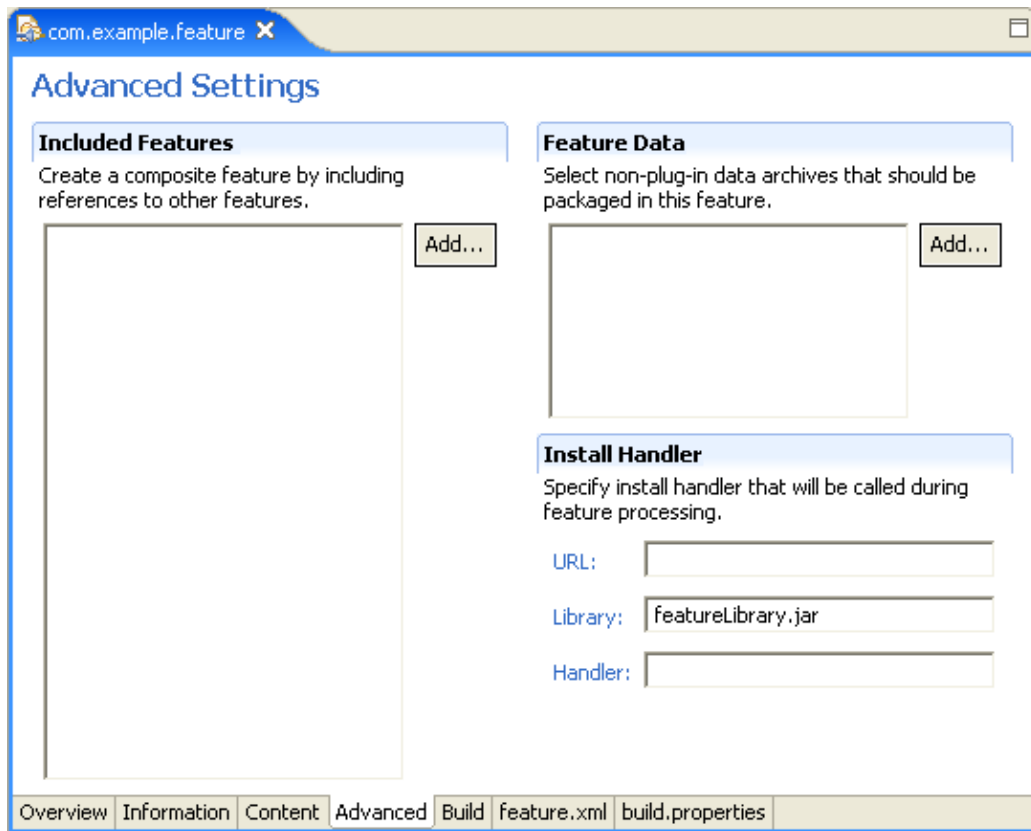
Plug-ins and fragments that are part of your feature are listed in the Content page. Pressing the **Add...** button brings up a list of checkboxes representing all of the valid plug-ins and fragments in the workspace. If you double-click on a plug-in or fragment entry, the plug-in or fragment manifest editor will open for the selected item.



Required plug-ins are plug-ins that are not part of the feature but must be present in the target platform as a prerequisite for installing the feature. If any of these plug-ins are missing, the feature will not be installed. Required plug-ins can be computed based on collective dependency information in plug-ins and fragments, or added manually using the **Add Plug-in...** button. The requirement can be based solely on plug-in IDs, or further constrained using expected versions and match rules.

Similar to plug-ins, features can be used as prerequisites. Similar rules apply. The distinct difference is that feature dependency is required when **patch** checkbox needs to be selected. See feature manifest reference for more details.

In addition to these standard pages, feature manifest can be used to set more advanced feature manifest data. Larger features may be built by including other features, thus creating a feature hierarchy. 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.



© Copyright IBM Corporation and others 2000, 2004.

Synchronizing versions

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 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.

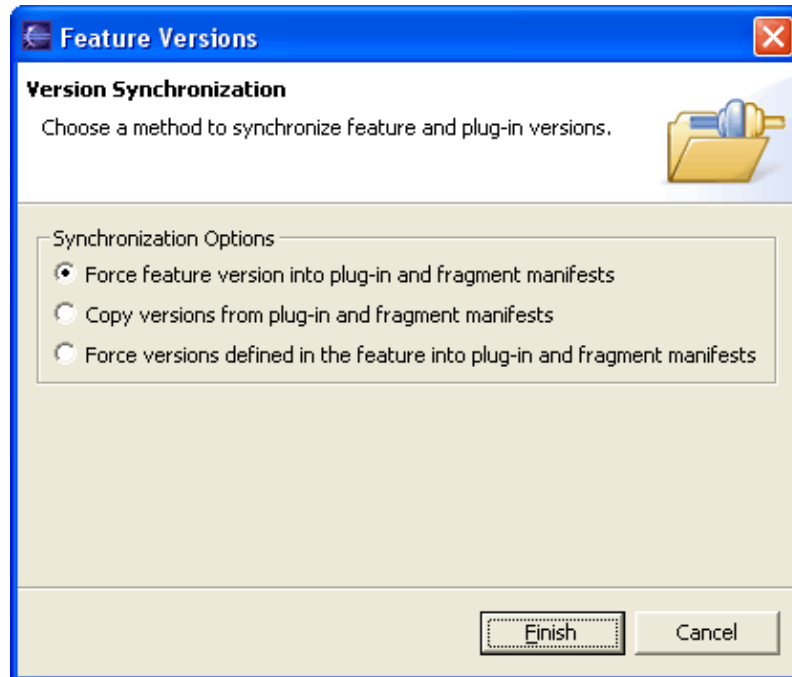
Example: Synchronizing versions in the feature editor

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.

Using the Plug-in Development Environment

5. Double-click on "XYZ Plug-in" and "German Fragment" objects and verify versions in their corresponding manifest editors.



Automatic synchronization at build time

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.

© Copyright IBM Corporation and others 2000, 2004.

Deploying a Feature

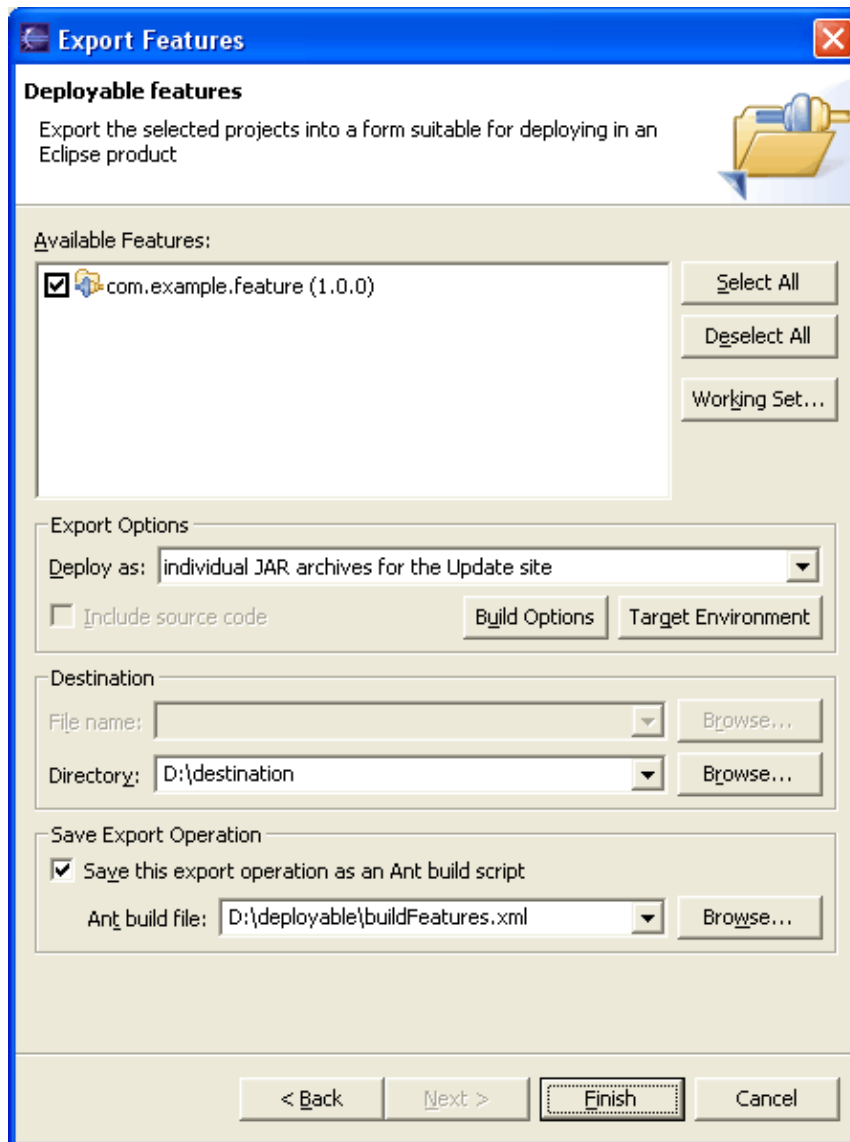
Once versions are synchronized, we can package our feature in the format fit for publishing.

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.

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.

1. Select **File>Export...>Deployable Features**.
2. Select the feature(s) you want to export.

Using the Plug-in Development Environment



The wizard will build all the features selected and their included plug-ins and fragments.

If the 'single deployable zip file' option is selected, everything will go into one zip file, which can then be unzipped into any Eclipse product.

If the 'update JAR archives' option is selected, the wizard will create a JAR for every feature selected. The JAR name will be of the format: **<feature_id>_<feature_version>.jar**, and will be placed in a **features/** sub-directory of the directory you specified in the wizard. The wizard will also create a JAR for every plug-in or fragment included in the feature. The JAR name will be of the format **<id>_<version>.jar**, and will be placed in a **plug-ins/** sub-directory of the directory you specified.

The alternative would be to build features manually:

Using the Plug-in Development Environment

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.

© Copyright IBM Corporation and others 2000, 2004.

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.

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 to include in the binary build;
- **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";

- **custom=true** – indicates that the build script is hand-crafted as opposed to automatically generated. Therefore no other value is consulted.

Plugin specific properties

- **source.<library>** – lists source folders to include for the library (e.g. **source.xyz.jar=src/, src-ant/**); If the library is specified in your plugin.xml, the value should match it;
- **output.<library>** – lists the output folder receiving the result of the compilation;
- **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**). If you use those build files in external build processes, refrains from using platform urls, or complement them with relative paths;

- **src.includes** – lists files to include in the source build;
- **src.excludes** – lists files to exclude from the source build;

Using the Plug-in Development Environment

- **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;
- **root.permissions.<config>.<permissionValue>** – list the files and folders to `chmod` to the given value for a specific configuration;
- **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 plugin **pluginId** will be the source plugin 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=cpl-v10.html,eclipse_update_120.jpg,feature.xml,feature.properties,license.html
root=rootfiles,file:../../plugins/org.eclipse.platform/startup.jar,configuration.files
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*

© Copyright IBM Corporation and others 2000, 2004.

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.

- **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 : a comma separated list of entries that will be fetched. Entries are expected to be 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 indicates the targeted configuration. The default is set to be platform independent;

install : deprecated, use buildDirectory instead;

scriptName : deprecated, the name is now always generated to be fetch_{elementId}.xml.

- **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;

Using the Plug-in Development Environment

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

buildingOSGi : optional, indicates if the pluginsflag indicating if the target is 3.0 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;

install : deprecated, use buildDirectory instead.

Examples

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

```
<eclipse.buildScript elements="plugin@org.eclipse.core.boot "  
    buildDirectory="c:\toBuild"/>
```

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;

- *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:/home/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 -buildfile build.xml
```

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

© Copyright IBM Corporation and others 2000, 2004.

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.

© Copyright IBM Corporation and others 2000, 2004.

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.

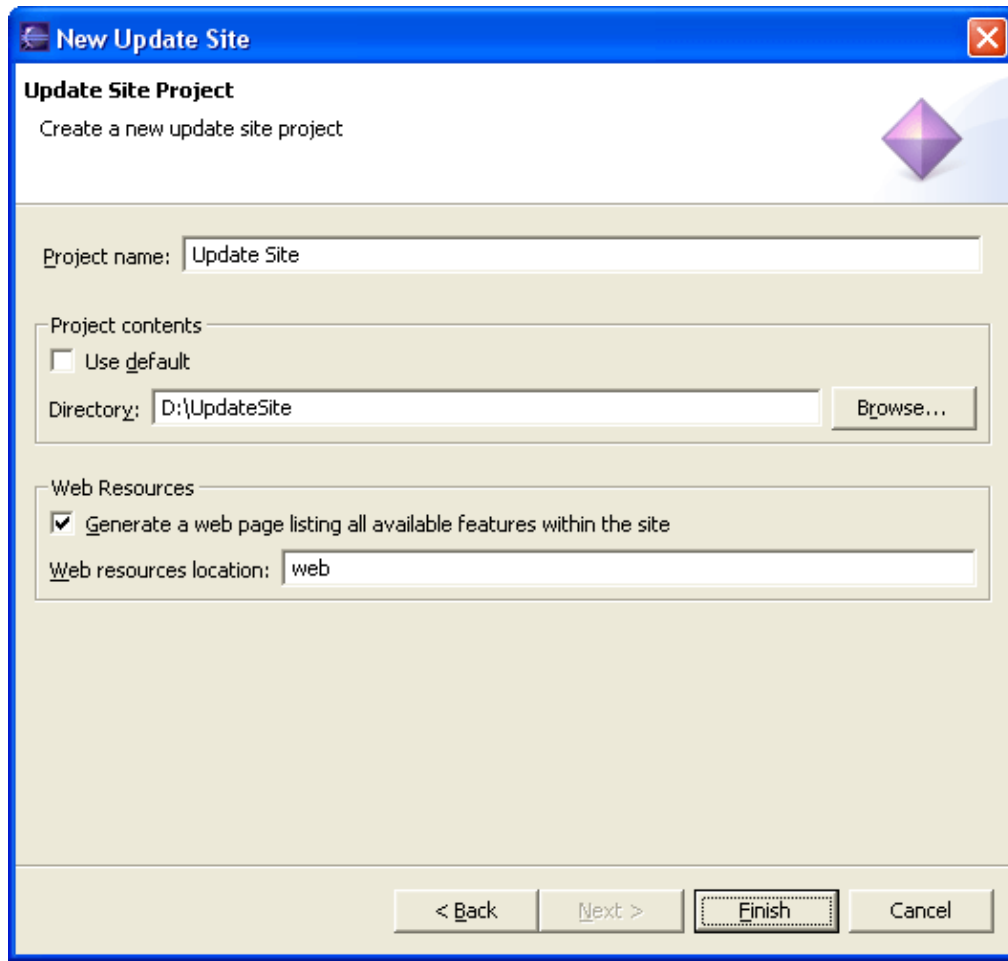
By default, an update site project is created local to your workspace. Since you may want to host multiple versions of your plug-ins and features in this site, you may want to keep the update site outside the workspace location. In that case, change the location of the update site project during creation. 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.

Using the Plug-in Development Environment

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**.

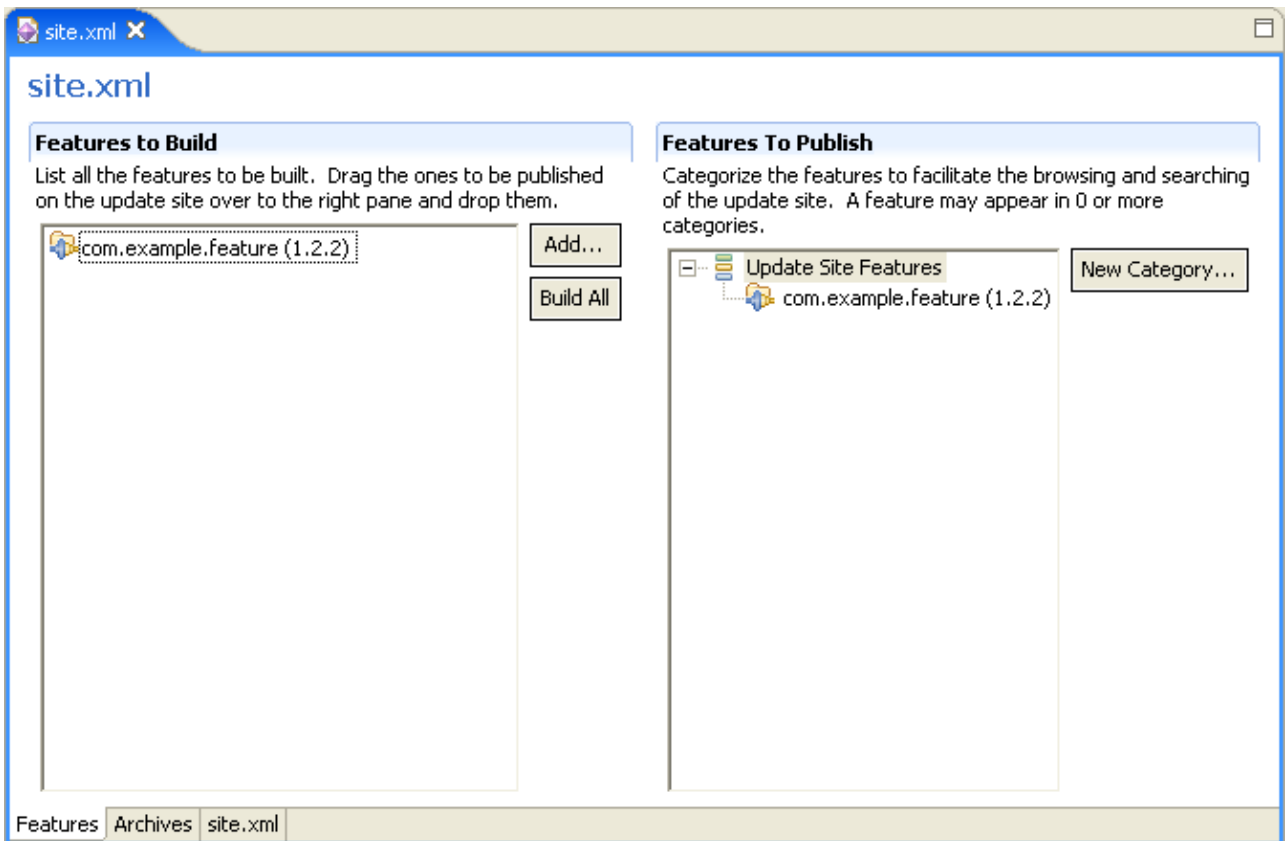


© Copyright IBM Corporation and others 2000, 2004.

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.

Using the Plug-in Development Environment



Features added to the **Features to Build** 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 a subset of those features, there is a **Build** action available through the context menu of that section.

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.

Not all features on the update site may be meant for general consumption. Therefore, the features you intend to publish need to be explicitly dragged and dropped in the **Features to Publish** section.

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 website would look like, save the site.xml file and open the index.html file at the root of your site project in a browser.

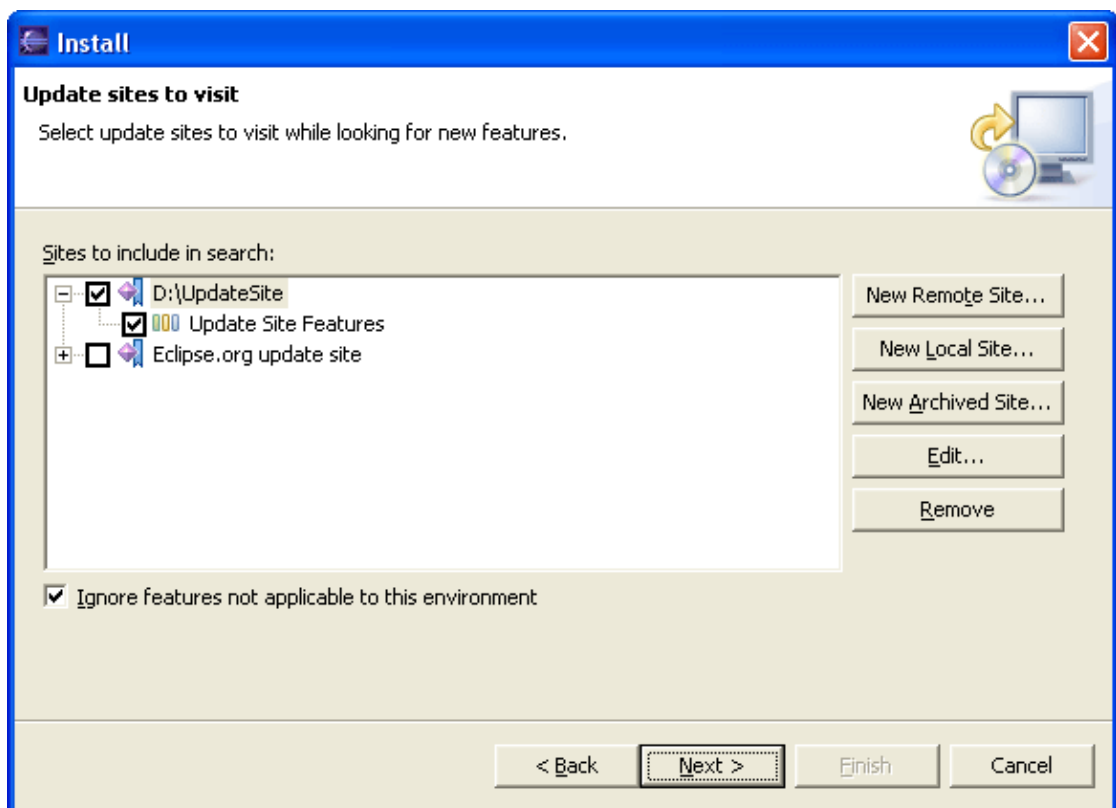
© Copyright IBM Corporation and others 2000, 2004.

Previewing an update site

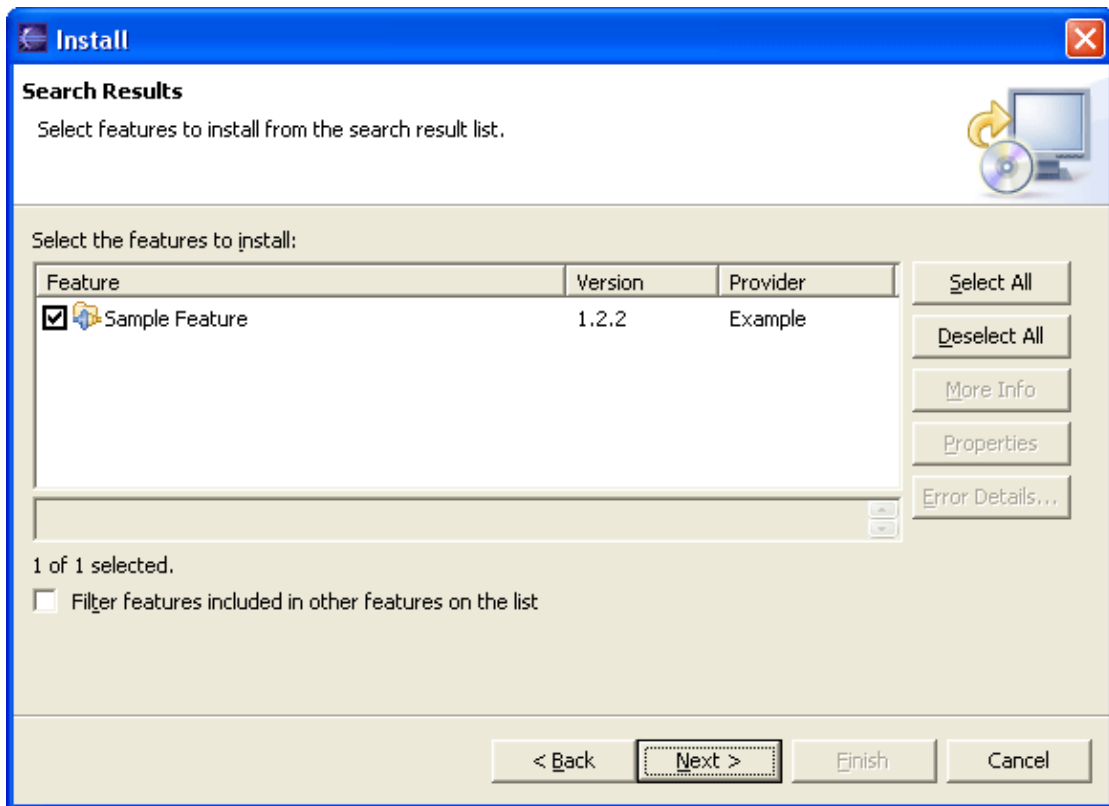
After you have built your feature using the update site editor and all the archives are in the right places, you can preview it in-place using the Update Manager.

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. Expand it to see and select the category you created. Click Next.



4. Verify that the feature you created has been located by the Update Manager. Proceeding beyond that point in the Install Wizard will go through the steps of the actual installation, which is not the focus here.




© Copyright IBM Corporation and others 2000, 2004.

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 {
    /**
```

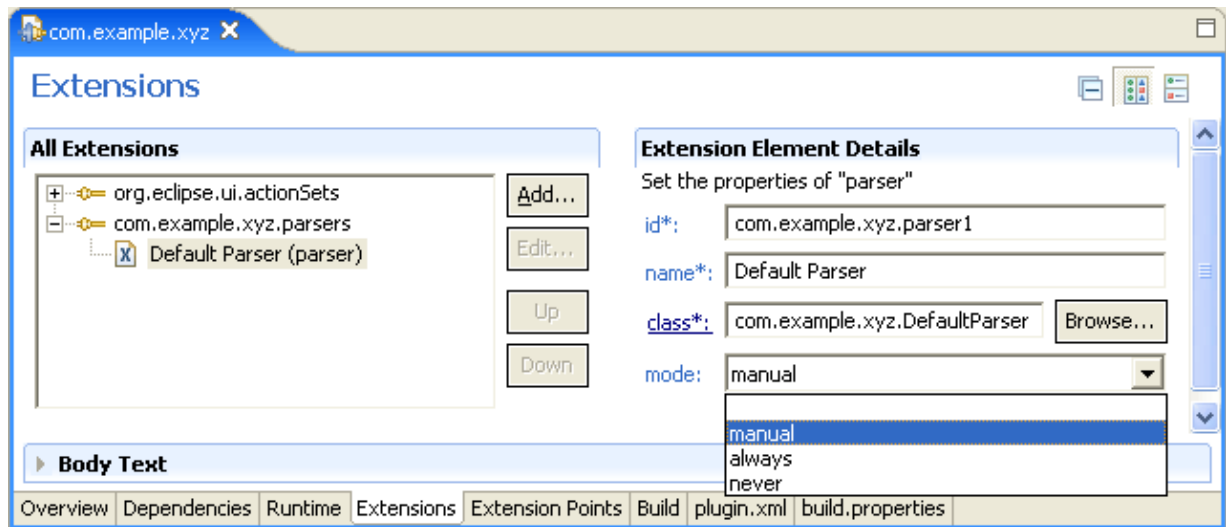
Using the Plug-in Development Environment

```
* 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 **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.



© Copyright IBM Corporation and others 2000, 2004.

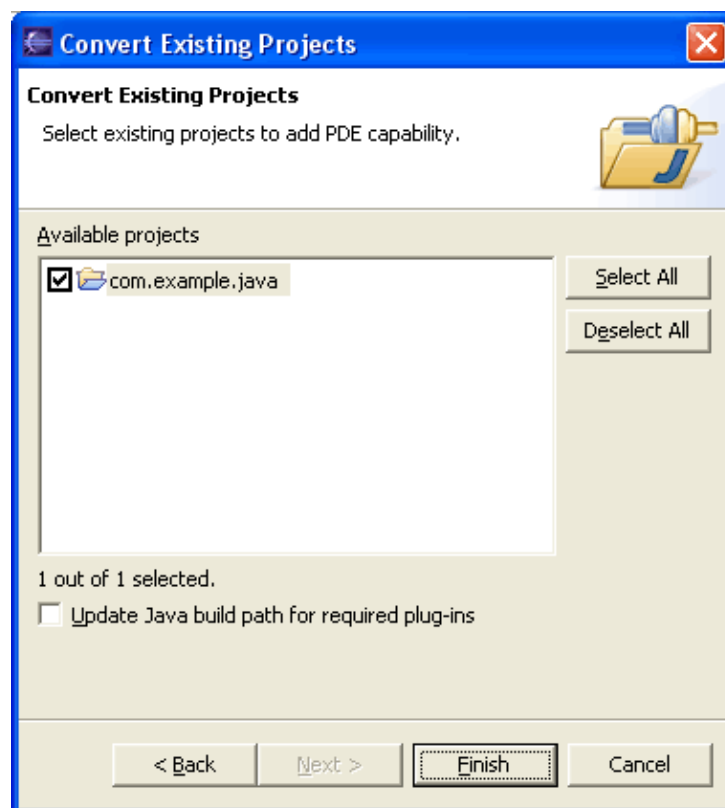
Converting existing projects into PDE projects

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 **File > New > Other...** and select **Convert Projects to Plug-in Projects** wizard from the **Plug-in Development** category. Click **Next**.



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. As an option, you can re-compute the build path in the process. If your project has a classpath you want preserved, deselect this checkbox.

© Copyright IBM Corporation and others 2000, 2004.

Reference

- [API Reference](#)
- [Extension Points Reference](#)
- [Other Reference Information](#)

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](#)

© Copyright IBM Corporation and others 2000, 2004.

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

Extension Wizards

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 (#CDATA)>

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"
```

Extension Wizards

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 Common Public License v1.0 which accompanies this distribution, and is available at <http://www.eclipse.org/legal/cpl-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">
```

- **id** – a unique name that will be used to identify this wizard.

Using the Plug-in Development Environment

- **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).

<!ELEMENT description (#CDATA)>

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=
"com.example.xyz.ExampleContentGenerator"
>
```

Using the Plug-in Development Environment

<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 Common Public License v1.0 which accompanies this distribution, and is available at <http://www.eclipse.org/legal/cpl-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.

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 Common Public License v1.0 which accompanies this distribution, and is available at <http://www.eclipse.org/legal/cpl-v10.html>.

Other Reference Information

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

- [Map of PDE Plug-ins](#)

© Copyright IBM Corporation and others 2000, 2004.

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
org.eclipse.pde.core[.*]	org.eclipse.pde.core
org.eclipse.pde.ui[.*]	org.eclipse.pde.ui

© Copyright IBM Corporation and others 2000, 2004.

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.

© Copyright IBM Corporation and others 2000, 2004.

Tips and Tricks

<p>Feature-based self-hosting</p>	<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>
<p>To clean or not to clean</p>	<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>

Using the Plug-in Development Environment

	<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 <code>-clean</code> argument from the launch configuration to improve startup time.</p>
<p>Importing with linking</p>	<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>
<p>Templates</p>	<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>
<p>Plug-in dependency extent</p>	<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>

Using the Plug-in Development Environment

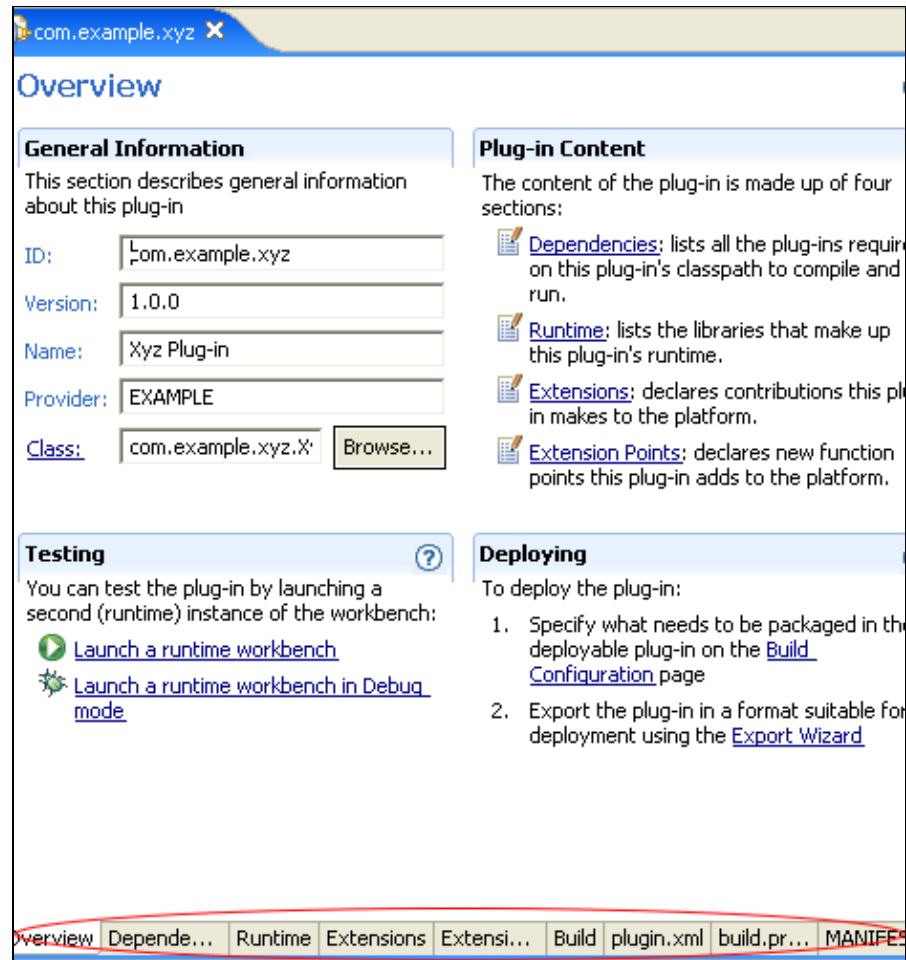
Finding unused dependencies	Minimizing a plug-in's number of dependencies is certain to improve performance. As your plug-in evolves, its list of dependencies might become stale, as it might be still containing references to plug-ins that it no longer needs. A quick way to check that all dependencies 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.
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.

© Copyright IBM Corporation and others 2000, 2004.

What's New in 3.0

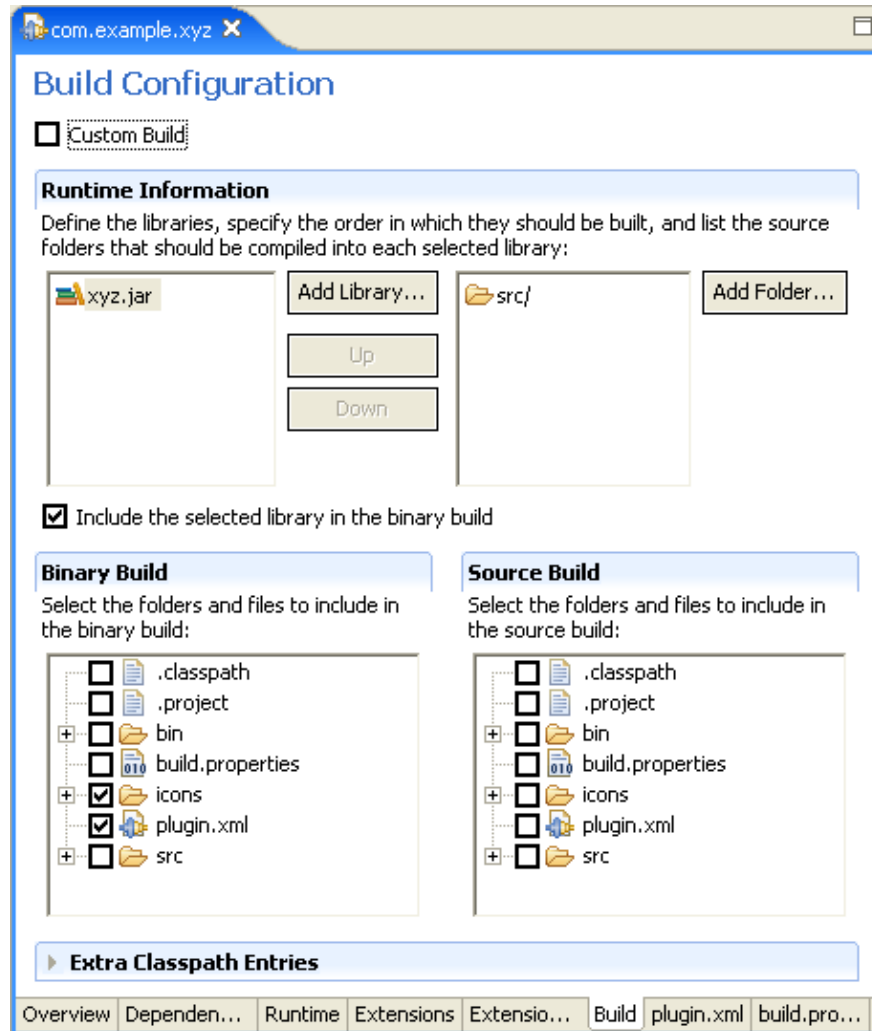
Here are some of the more interesting or significant changes made to PDE for the 3.0 release of Eclipse since 2.1:

One Plug-in, One Editor	The new plug-in manifest editor is a single multi-page editor that can be used to manage your plug-in and edit all its files (plugin.xml, build.properties, and manifest.mf). PDE transparently handles the task of writing the changes to the right files.
--------------------------------	---



New PDE build configuration editor

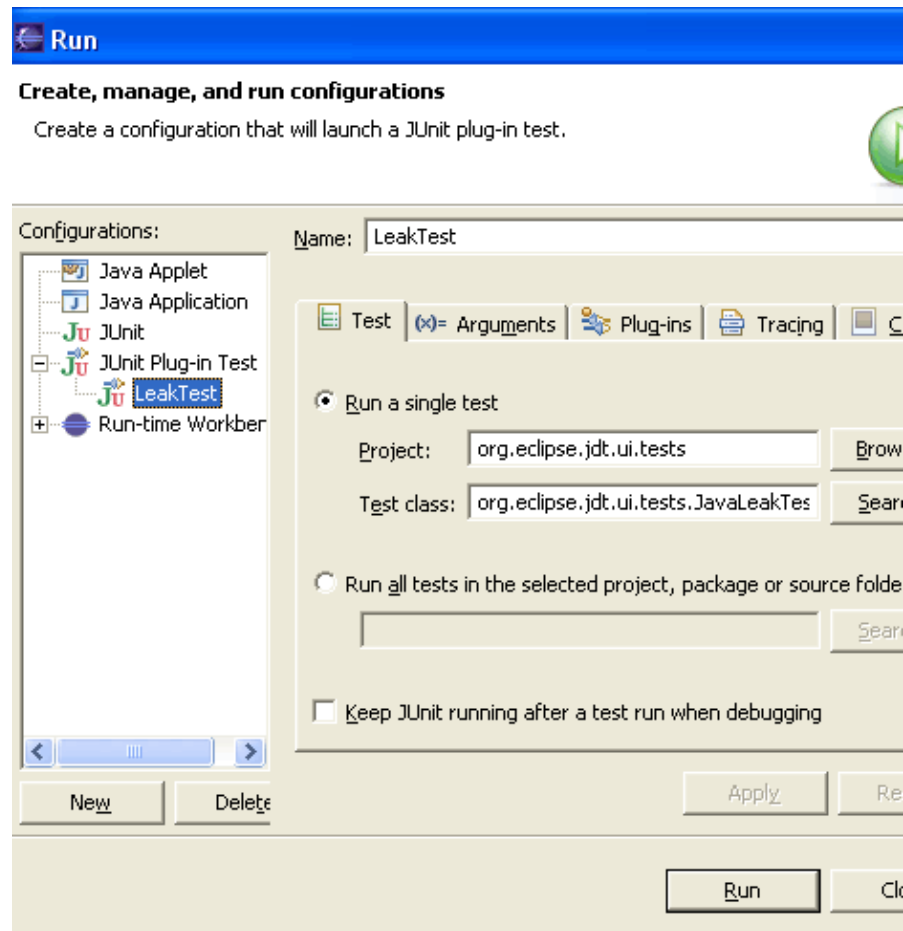
There's no longer any reason to dread editing your plug-in's cryptic build.properties file. PDE now provides a specialized build configuration editor that makes this task easy and intuitive.



Unit testing for plug-ins

PDE provides a new launcher for JUnit-based unit test suites for plug-ins. The launcher gives you fine-grained control over the set of plug-ins to run in a test, lets you debug with tracing, and can handle GUI as well as non-GUI plug-ins.

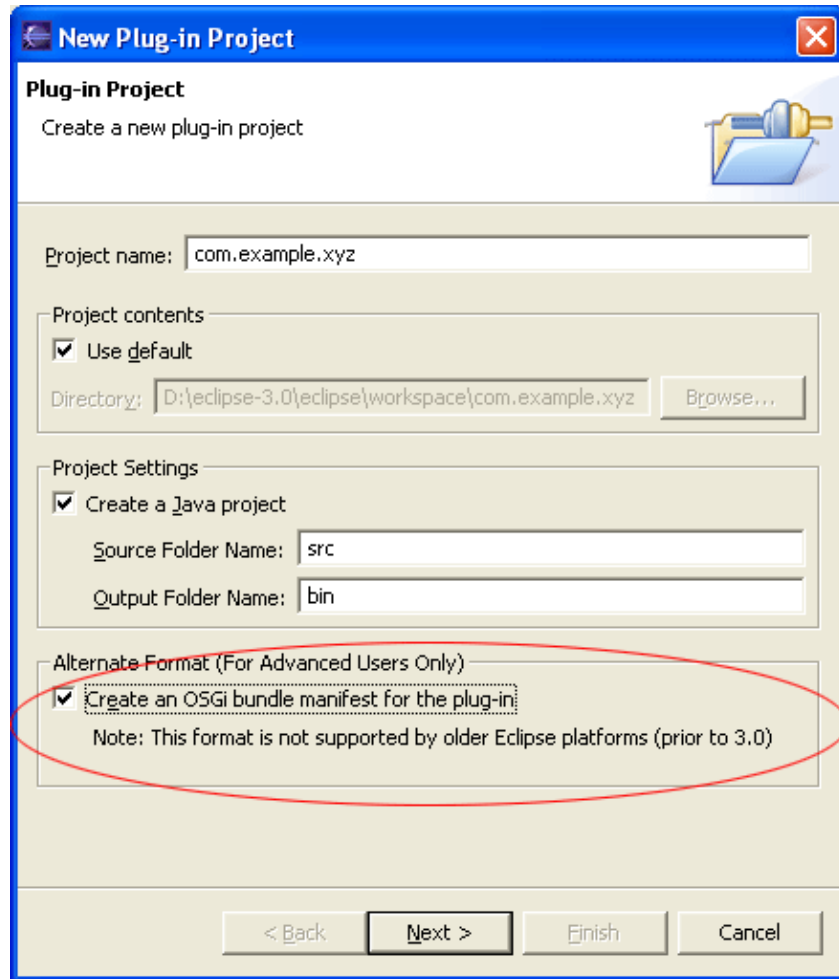
Using the Plug-in Development Environment



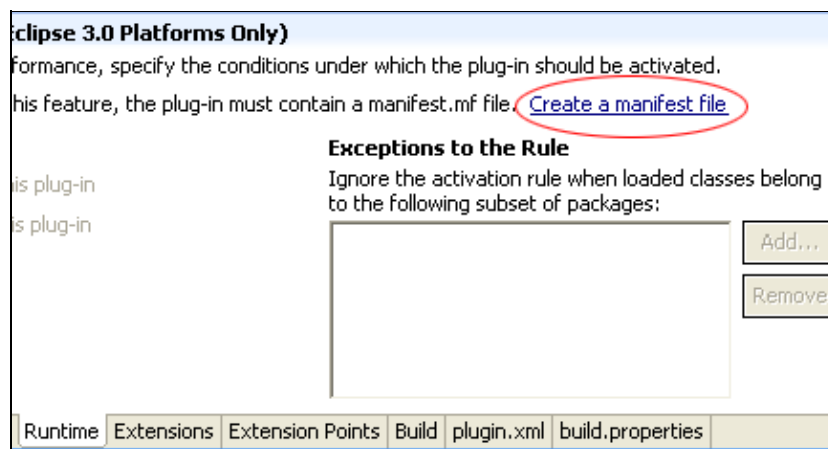
PDE support for advanced runtime options

Plug-in developers wishing to exploit the capabilities of the new OSGi-based platform runtime can now self-host with plug-ins that use explicit OSGi bundle manifests (manifest.mf file). They will be able to import, develop, and test their plug-ins using PDE.

To take advantage of the new OSGi-based platform runtime on plug-in creation, PDE's plug-in project creation wizard now has an option for creating plug-ins with explicit OSGi bundle manifests.



The Runtime page of the PDE manifest editor exposes one of several OSGi-based runtime features (control of plug-in activation) and will even create a manifest.mf file for your plug-in on demand.



PDE computes plug-in build

You never need to update the Java build path of your plug-in ever again. PDE uses the JDT classpath container mechanism to dynamically compute the Java build path of a plug-in project. Because classpath containers are

Using the Plug-in Development Environment

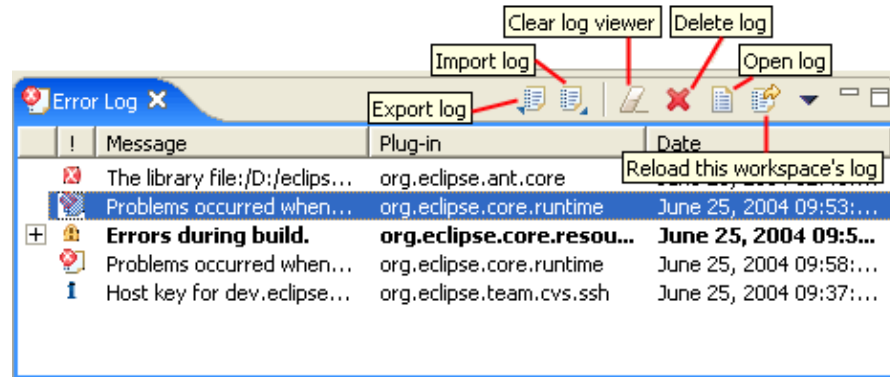
class paths dynamically

resolved on the fly, they are always accurate regardless of which plug-ins you are building against and whether they are loaded in workspace.

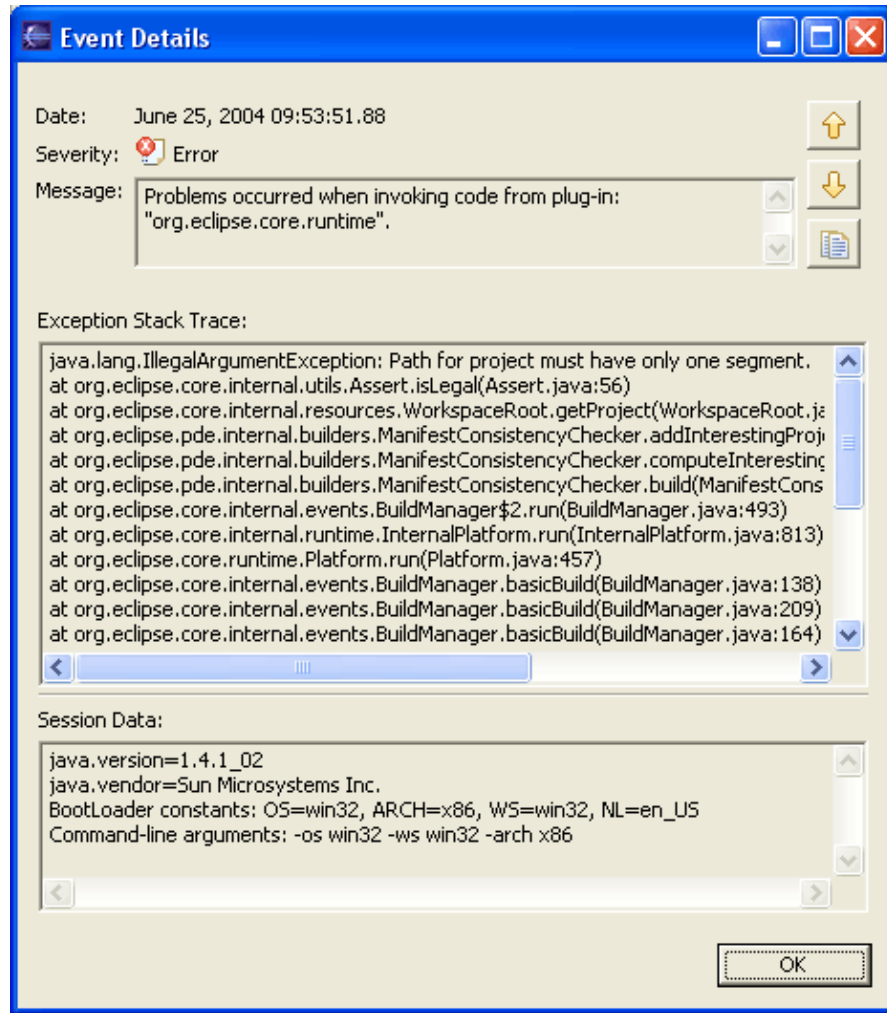
Refer to the [Dynamic Classpaths FAQ](#) for more details.

Improved error log view

Additional features in the tool bar of the Error Log view make it effortless to export, open, or delete the current log. You can also import external logs and reload your workspace log into the view. Events can be organized via filtering and sorting by message, plug-in name, or date.



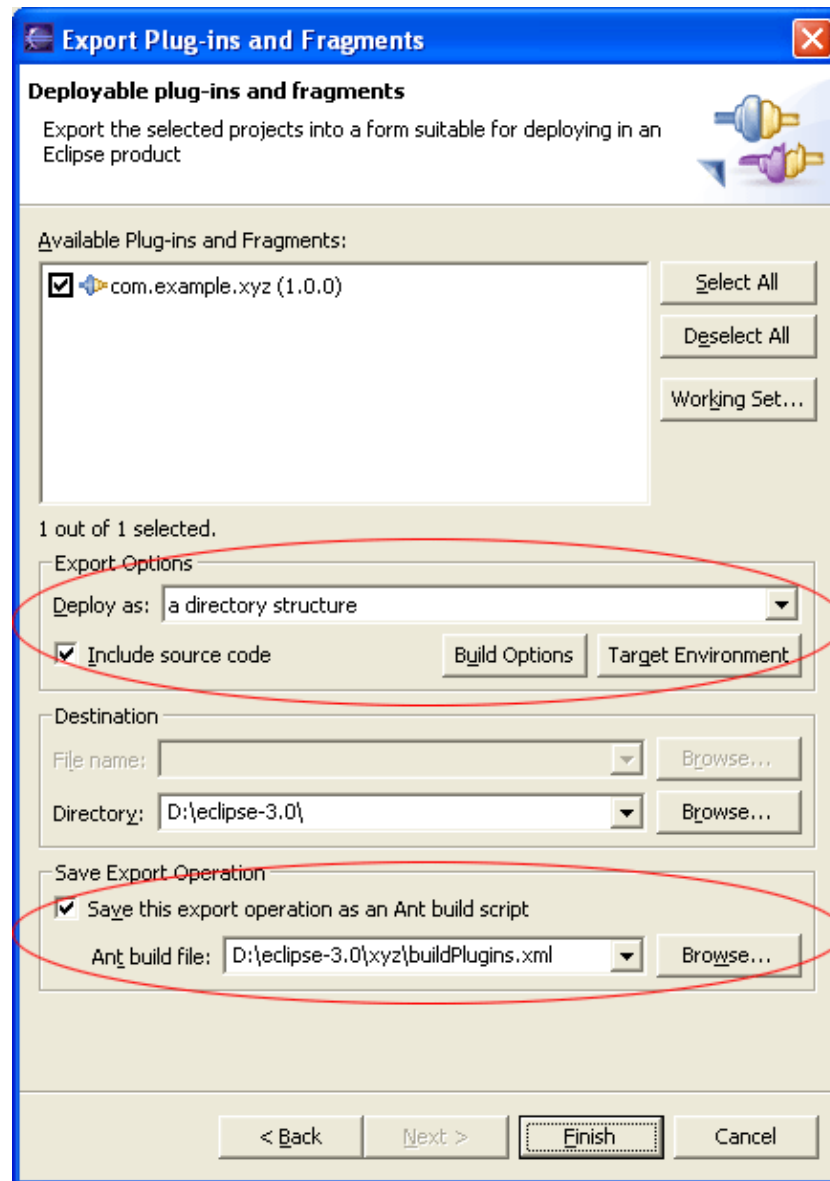
The error log view now lets you inspect the full details of an event, and easily navigate from one event to the next.



Additional exporting functionality

The PDE Export Deployable Plug-ins wizard now allows the plug-ins to be exported as a directory structure, and provides direct access to the preference page with the compiler settings that will be used.

Also, the plug-in export operations can now be saved as Ant build scripts so that the same operation can be run later via the Ant runner without having to go through the export wizard.



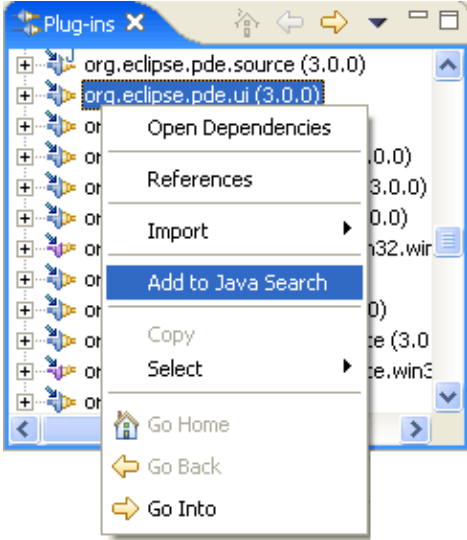
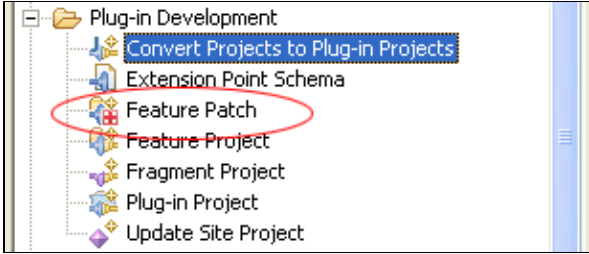
Expanding the Java search scope and source lookup

The "Add to Java Search" functionality was introduced in 2.1 to expand the scope of the Java search beyond workspace projects to include JARs from external plug-ins that constitute your target platform.

Select plug-ins and invoke **Add to Java Search** via the context menu of the Plug-ins view.

PDE now manages and updates references to JARs automatically as you upgrade from build to build, and makes these JARs visible to the debugger. Taking advantage of this functionality will ensure that the debugger will automatically locate the relevant source code (if available).

Using the Plug-in Development Environment

	
<p>New feature patch wizard</p>	<p>Available under New > Project...> Plug-in Development > Feature Patch, there is now a wizard to help you create a patch for a feature. You can then publish the patch on an Update site so that customers of your feature can easily download and install it via the Update Manager.</p> 
<p>New PDE extension point</p>	<p>The new <i>org.eclipse.pde.ui.newExtension</i> extension point allows a tool to register custom extension editing wizards. These wizards allows developers contributing to extension points to work at a higher level; the wizards handle the conversion into XML elements.</p>

© Copyright IBM Corporation and others 2000, 2004.