8 Introduction to Eclipse plug-ins

In this chapter...

- Understanding Eclipse's plug-in architecture
- Preparing your Workbench for plug-in development
- Using the Plug-in Development Environment (PDE)
- Creating simple plug-ins using the built-in wizards and templates

Up to now, you've been using Eclipse as it comes out of the box. As you'll discover in this chapter, however, the beauty of Eclipse lies in its extensible architecture. This architecture allows anyone to add features and capabilities the original designers never dreamed of.

Eclipse's loosely connected design is perfect for systems that aren't designed all at once, but instead are built up from components written for particular needs. From its earliest roots, Eclipse was designed as an "open extensible IDE for anything, and nothing in particular." The decentralization that plug-ins provide gives the Eclipse Platform the ability to morph into any application and support any language. Come with us now as we seek out that man behind the curtain and learn the secrets of Eclipse plug-ins.

8.1 Plug-ins and extension points

Imagine a giant jigsaw puzzle. A few pieces are already connected for you—these will form the core around which the rest of the puzzle is built. The boundaries between the pieces are uniquely cut to fit snugly together. If Eclipse is the puzzle, then the pieces are plug-ins. A *plug-in* is the smallest extensible unit in Eclipse. It can contain code, resources, or both.

The Eclipse Platform consists of nearly 100 plug-ins working together. The boundaries between these pieces that let plug-ins connect to one another are called extension points. An *extension point* is the mechanism by which one plug-in can add to the functionality of another.

Appendix C lists the extension points provided by the Platform. Each one can be used to add some new component such as a menu or view to the system, and is usually associated with a Java class that performs the logic for the component.

Unlike most jigsaw puzzles, though, Eclipse has no corners or straight edges. It can be extended forever, with each new plug-in defining its own extension points that other plug-ins can use. Large projects such as WebSphere Studio have hundreds of plug-ins. (Better bring a big table.)

8.1.1 Anatomy of a plug-in

Plug-ins are conceptually simple. If you look at the directory where you installed Eclipse in chapter 2, you will see a subdirectory called plugins. Inside this directory you'll find one directory for every plug-in. The name of each directory is the same as the name of the plug-in, followed by an underscore and a version number. For example:

```
C:\ECLIPSE
|
+---features
+---plugins
|   +---org.eclipse.ant.core_2.1.0
|   |   |    .options
|   |   |    about.html
|   |   |    antsupport.jar
|   |   |    plugin.properties
|   |   |    plugin.xml
|   |   |
|   |   +---lib
|   |
|   +---org.eclipse.compare_2.1.0
|   |
|   ...
+---workspace
```

The `org.eclipse.ant.core` plug-in provides the Eclipse Platform with its integration with the Ant builder (see chapter 5). In every plugins folder, including this one, you will find a *plug-in manifest* file (plugin.xml) together with some optional files. The manifest describes the plug-in—its name, its version number, and so forth. It also lists the required libraries and all the extension points used and defined by the plug-in.

The files and folders typically seen in a plugins folder are as follows:

- *plugin.xml*—Plug-in manifest
- *plugin.properties*—Contains translatable strings referenced by plugin.xml
- *about.html*—Standard location used for licensing information
- **.jar*—Any Java code needed for the plug-in
- *lib*—Directory for more JAR files
- *icons*—Directory for icons, usually in GIF format
- *(other files)*—As needed

8.1.2 The plug-in lifecycle

When you first start the Eclipse Platform, it scans the plugins directory to discover what plug-ins have been defined (this is a slight simplification, but close enough for this discussion). If it finds more than one version of the same plug-in, only one (typically the one with the highest version number) will be used. The list of plug-ins the Platform builds during this scan is called the *plug-in registry*. Although the Platform reads all the plug-in manifests, it doesn't actually load the

plug-ins (that is, run any plug-in code) at this point. Why? To make Eclipse start up faster.

Plug-ins are loaded only when they are first used. For example, if you write a plug-in that defines a menu item, Eclipse can tell by looking at the manifest where the menu should go and what the text of the menu is. Because of the information in the manifest, Eclipse can delay loading your plug-in until it is really needed.

If you select the menu, the plug-in is loaded at that point. This behavior is especially important in large Eclipse-based products with hundreds of plug-ins. Most of the plug-ins will not be needed, because they are in specialized parts of the product that may never be run. So, any time spent loading and initializing those plug-ins would be wasted. This is sometimes referred to as *lazy loading*.

When are plug-ins unloaded? The short answer is, never. However, one of the goals of the Equinox project (http://www.eclipse.org/equinox) is to allow plug-ins to be loaded and unloaded on demand, so this situation may change in the future.

8.1.3 Creating a simple plug-in by hand

Eclipse plug-ins can be created without any special tools. To demonstrate, create a subdirectory in the plugins directory called org.eclipseguide.simpleplugin_1.0.0. Inside this directory, use a text editor like Notepad or vi to create a plugin.xml file containing the following lines:

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
   id="org.eclipseguide.simpleplugin"
   name="Simple Plug-in"
   version="1.0.0"
   provider-name="Eclipse in Action">
</plugin>
```

Now save the file and restart Eclipse. You won't notice anything different, because this plug-in doesn't do anything. However, you can tell it was registered by selecting Help→About Eclipse Platform and then clicking Plug-in Details. Scroll down to the bottom of this window, and you'll see the plug-in listed as shown in figure 8.1. The More Info button is grayed out because you didn't create an about.html file.

Table 8.1 describes the purpose of each line in plugin.xml.

Congratulations—you have just created your first plug-in! Next we'll look at the tools Eclipse provides to make this process manageable for more complex projects.

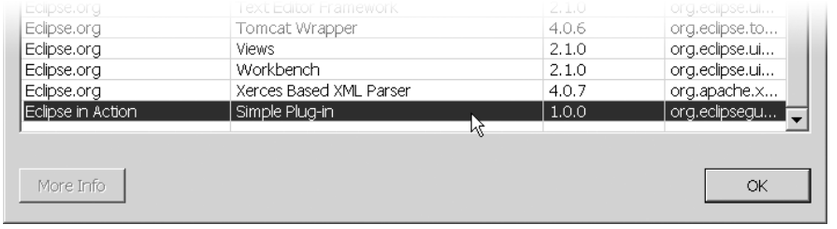

Figure 8.1 In the About page, you can click the Plug-in Details button to see the list of installed plug-ins. The Simple Plug-in shown here was discovered by the Eclipse Platform during startup.

Table 8.1 The plug-in manifest file (plugin.xml) for each plug-in is read when Eclipse starts, in order to build up its plug-in registry. Here is the simplest manifest possible and the meaning of each line.

Line	Purpose
`<?xml version="1.0" encoding="UTF-8"?>`	Required XML prolog; never changes
`<plugin`	Starts defining a new plug-in
`id="org.eclipseguide.simpleplugin"`	Provides the fully qualified id for the plug-in
`name="Simple Plug-in"`	Gives the plug-in a human-readable name
`version="1.0.0"`	Specifies a version number
`provider-name="Eclipse in Action">`	Provides information about the author
`</plugin>`	Finishes defining the plug-in

8.2 The Plug-in Development Environment (PDE)

Creating a plug-in by hand is an interesting exercise, but it would quickly become tedious in practice. Plug-in manifests can grow to be hundreds of lines long, and they need to be coordinated with names and data in various source and property files. Also, plug-ins need a fair amount of boilerplate code in order to run. That's why Eclipse provides a complete Plug-in Development Environment (PDE). The PDE adds a new perspective and several views and wizards to the Eclipse Platform to support creating, maintaining, and publishing plug-ins:

- *Plug-in Project*—A normal plug-in; the most common type
- *Fragment Project*—An addition to a plug-in (for languages, targets, and so on)
- *Feature Project*—An installation unit for one or more plug-ins
- *Update Site Project*—A web site for automatic installs of features

8.2.1 Preparing your Workbench

Before you start using the PDE, you should turn on a few preferences. They are off by default, because Eclipse users who are not building plug-ins don't need them. Bring up the Preferences window (Window→Preferences) and do the following:

1. Select Workbench→Label Decorations and turn on the Binary Plug-in Projects decoration. This is optional, but if you use binary plug-ins (see section 8.2.2) it will help them stand out from the rest of your projects.
2. Select Plug-In Development→Compilers and set all the messages to Warning. Doing so will provide an early indication of any problems in your plug-in manifests.
3. Select Plug-In Development→Java Build Path Control and turn on the Use Classpath Containers for Dependent Plug-ins option. This confusingly named option causes all plug-in JARs that your plug-in uses to appear in a folder of your project called Required Plug-in Entries. The nice thing about this special folder is that Eclipse dynamically manages it based on your plug-in's dependencies.
4. Click OK. A dialog will appear, stating that the compiler options have changed and asking whether you would like to recompile all the projects. Click Yes.

8.2.2 Importing the SDK plug-ins

As mentioned earlier, the Eclipse Platform is made up of dozens of plug-ins. Wouldn't it be nice if you could see the source code for all those plug-ins, and do searches to see how certain classes and interfaces are used internally? The API documentation is not perfect, so this is an important tool for plug-in developers. Of course, you could connect to the Eclipse CVS Repository (host **dev.eclipse.org**, path **/home/eclipse**, user **anonymous**) and download what you need, but there is a better way. It turns out that if you downloaded the Eclipse Platform SDK then all the source code is already installed, just waiting to be used.

The easiest method is to hold down the Ctrl key and hover your mouse over a class or object name in the Java editor, and then click on the name. If the source is available, Eclipse will open it. Or, using the Package Explorer, you can expand just about any JAR file and double-click on one of its class files to open it in the editor.

Sometimes, though, it's more convenient to bring these plug-ins into your workspace just like your regular projects. For example, you can search your

entire workspace for references to a type, but if the type is not currently in the workspace, then it won't be found. The Required Plug-in Entries folder is searched, but it contains only the JAR files from plug-ins you are currently dependent on.

To bring installed plug-ins into your workspace, select File→Import→External Plug-ins and Fragments and then click Next (see figure 8.2). Turn off the option to Copy Plug-in Contents into the Workspace Location and click Next. Then, select the plug-ins you want to import and click Finish. This is called a *binary import*, and projects created this way are *binary plug-ins* because you didn't build them from source.

Later, if you decide you don't want them in your workspace, just delete them—doing so will not affect the Eclipse installation. You can also temporarily

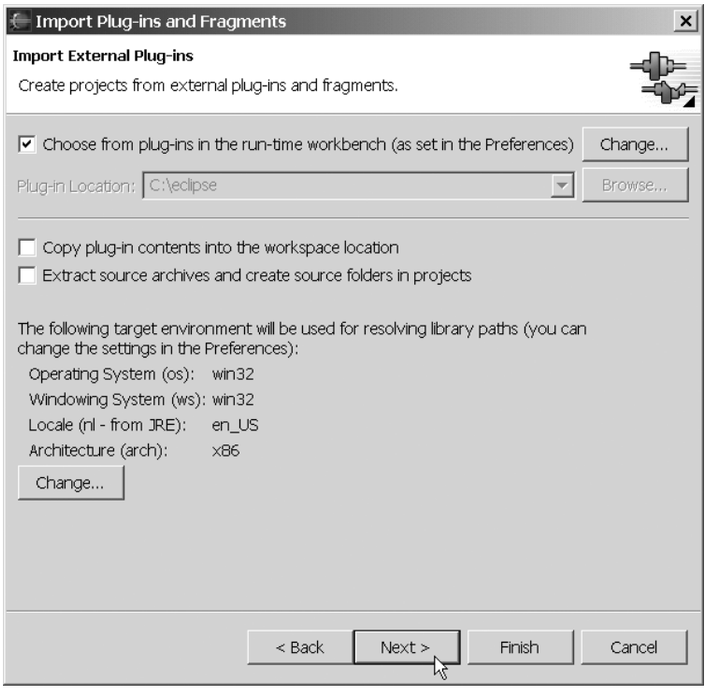

Figure 8.2 You can bring any installed plug-ins into your workspace by importing them. Doing so creates a binary plug-in project for each one and makes them available for searching and browsing. This is a great way to discover how the Eclipse Platform uses the Eclipse SDK classes and interfaces.

hide them from the Package Explorer menu: Select Filters, turn on the option to Exclude Binary Plug-in and Feature Projects, and click OK.

8.2.3 Using the Plug-in Project Wizard

The PDE makes it easy to create a new plug-in by providing wizards that ask a few questions and generate much of the code for you. Let's walk through a simple example:

1. Select File→New→Project to bring up the familiar New Project Wizard shown in figure 8.3.
2. Select Plug-in Development on the left-hand side to bring up the list of plug-in wizards on the right. You can use the Plug-in Project Wizard to create new plug-ins; select it and then click Next to open the first page of the wizard.
3. Enter a name for the plug-in, such as **org.eclipseguide.helloplugin** (see figure 8.4). We recommend using a fully qualified name like this so it can match the plug-in name and not collide with anyone else's name. By default, the PDE creates the plug-in in your normal workspace directory (either the workspace directory where you installed Eclipse or the directory you specified with Eclipse's `-data` option). Click Next to get to the next page.
4. Enter the fully qualified ID of the plug-in (see figure 8.5); by default, the ID is the same as the project name, which is what you want. Select the Create a Java Project option. Some plug-ins consist of only resources,

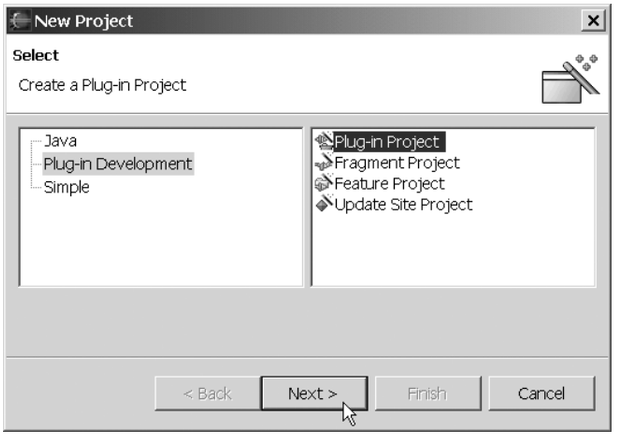

Figure 8.3
The PDE provides several wizards for creating new projects. See section 8.2 for a description of each type of project supported.

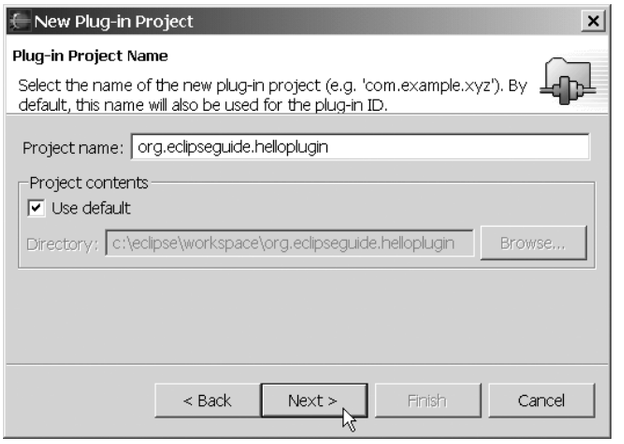

Figure 8.4
Specify the name and location of the project in the first page of the Plug-in Project Wizard.

such as a plug-in that only contains help files, but most of the time plug-ins have some Java code associated with them. The Java Builder Output option controls where the Eclipse compiler places generated .class files. The Plug-in Runtime Library is the JAR file that holds all your Java code, and Source Folder allows you to change the subdirectory that contains your .java files. The defaults for these settings are fine, so click Next to bring up the code generation page.

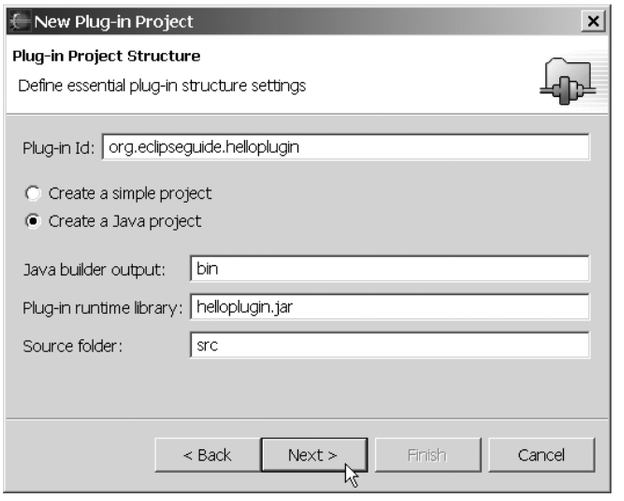

Figure 8.5
Specify the fully qualified ID of the plug-in in the second page of the New Plug-in Project Wizard. You can also control whether this plug-in will contain Java code.

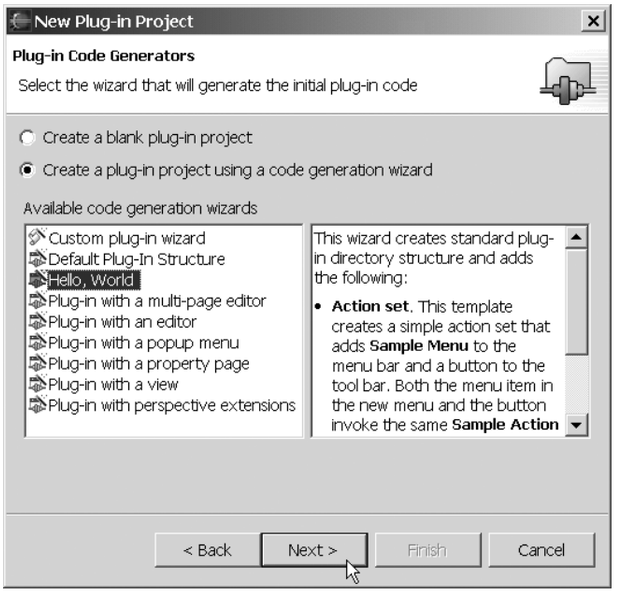

Figure 8.6
Select a code-generation wizard to quickly create a new plug-in from a template. Using the templates lessens the learning curve for Eclipse extensions and is less error-prone than creating the code from scratch. There also is an experimental feature in Eclipse 2.1 for adding your own wizards to this dialog.

5 The PDE provides several standard templates to help you quickly create some common kinds of plug-ins. A few are listed on this page (figure 8.6); you can see the rest by selecting the Custom Plug-in Wizard. The option to Create a Blank Plug-in Project makes a minimal directory with a plugin.xml file but not much else; no code is generated. The Default Plug-in Structure Wizard creates a top-level Java class for you but does not use any extension points. It is possible to add new templates, if necessary.

You're almost done. Now you just have to decide which template to use.

8.3 The "Hello, World" plug-in example

It's time for another "Hello, World" example—this one for plug-ins. In the New Plug-in Project page, select the Hello, World Wizard. It creates the default plug-in structure and also uses two extension points (`org.eclipse.ui.actionSets` and `org.eclipse.ui.perspectiveExtensions`) to add an item to the menu bar and the tool bar. (These extension points and the other wizards will be discussed later). Now, follow these steps:

1 In the code-generation dialog, click Next to start the Hello, World Wizard. Set the Plug-in Name to **Hello Plug-in**, the Class Name to **org.eclipse-**

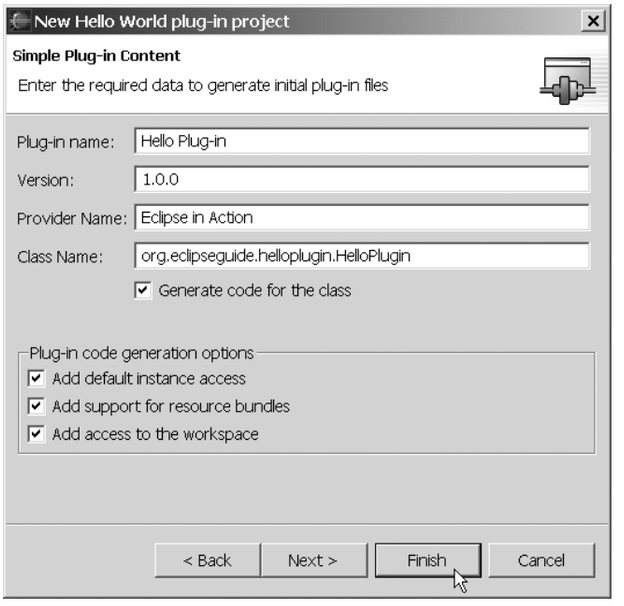

Figure 8.7
This page is common to most plug-in wizards. Use it to fill in the plug-in name and other required data.

guide.helloplugin.HelloPlugin, and the Provider Name to **Eclipse in Action** (see figure 8.7). The next page would let you control the text the example will display; however, the defaults are good, so click Finish to generate the directories, files, and classes necessary for the project.

2. You may get a dialog telling you that the wizard is enabling any needed plug-ins (figure 8.8). This is normal, so click OK.
3. Another dialog asks if you want to switch to the Plug-in perspective (figure 8.9). Click Yes.

That's it! You now have a plug-in project, as shown in figure 8.10.

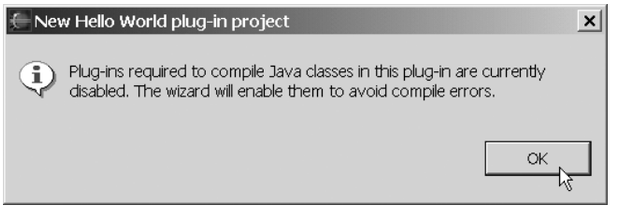

Figure 8.8
The wizard automatically enables plug-ins that this plug-in depends on. You can see the list of enabled plug-ins in the Preferences dialog under Plug-In Development→Target Platform.

Figure 8.9
Plug-in development is best accomplished in the Plug-in perspective. This is optional, however; you can use any perspective you like, as long as it has the views you need.

Figure 8.10
The final result is a plug-in project that prints "Hello, World". Here we have expanded some of the folders so you can see the generated files.

8.3.1 The Plug-in Manifest Editor

The PDE automatically opens the Plug-in Manifest Editor when you first create a plug-in (see figure 8.11). You can bring it up later by double-clicking on plugin.xml. This multipage editor provides convenient access to all the different sections of the plugin.xml file.

Because you'll be spending a great deal of time in the Plug-in Manifest Editor, it's a good idea to familiarize yourself with it now. Its pages are as follows:

- *Welcome*—A quick introduction to the Manifest Editor with links to some of the most important sections. You can turn off this page once you are familiar with the editor.
- *Overview*—Summarizes the plug-in, including the name, version number, extension points consumed and provided, and other information. This is the page you will use most often.
- *Dependencies*—Specifies the plug-ins required for this plug-in.
- *Runtime*—Defines the libraries that need to be included in the plug-in's classpath and whether classes in those libraries should be exported for use by other plug-ins.

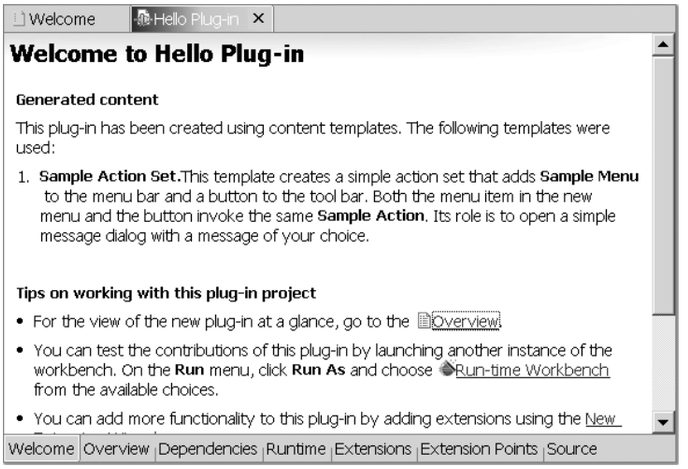

Figure 8.11
When you first open the plugin.xml file, you are greeted with this Welcome screen. It contains hints about how to work with the project, active links to different pages in the Plug-in Manifest Editor, and actions like starting the Run-time Workbench. You can turn off the page when you become more comfortable with the environment.

- *Extensions*—Lists all extension points used by this plug-in.
- *Extension Points*—Lists all extension points defined by this plug-in, along with a cross-reference of who is using those extension points.
- *Source*—A raw XML editor for the plugin.xml file. Often it is useful to make a change in one of the other pages of the editor and then switch to the Source page to see what effect the change had.

8.3.2 The Run-time Workbench

Once you have created a plug-in project, you could compile it, package up a JAR file, copy it to the plugins directory as you did in section 8.1.3, and restart Eclipse. But the PDE provides an easier way in the form of the *Run-time Workbench*, a temporary Eclipse installation created automatically for running and debugging plug-ins.

To run your new plug-in under the Run-time Workbench, select Run→Run As→Run-time Workbench (or use the Run toolbar button). If you haven't done this before, then you will get a notice that Eclipse is completing a new installation, and then a new Eclipse Workbench will appear. This is the Run-time Workbench; it includes all the plug-in projects you are working on in addition to the plug-ins that are part of the standard Eclipse installation. If you look carefully, you will notice a new menu named Sample Menu and a new button in the Eclipse toolbar (see figure 8.12).

Click the button, and Eclipse opens a dialog commemorating your second plug-in (see figure 8.13). We'll look at the Java code behind that button shortly.

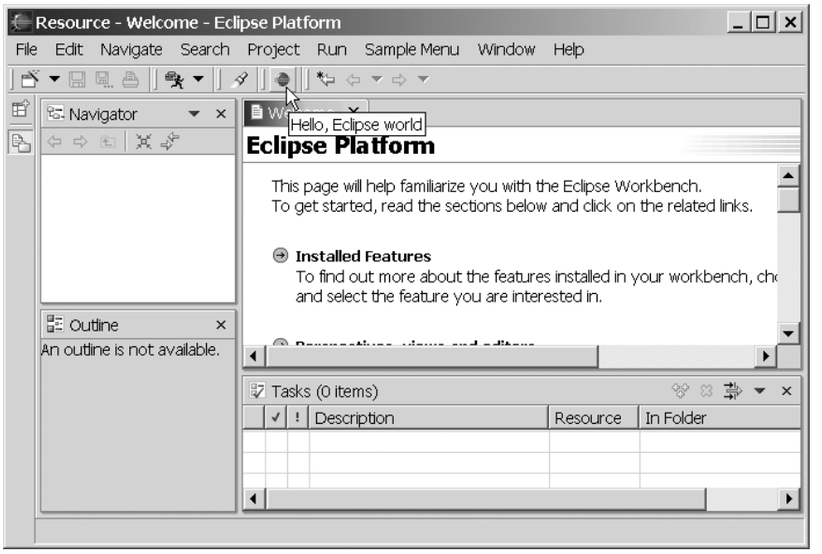

Figure 8.12 Starting the Run-time Workbench opens a new instance of Eclipse with all the plug-ins from your workspace installed. The "Hello, World" plug-in adds a Sample Menu and a toolbar button to the Workbench.

Debugging a plug-in is just as easy. Select Run→Debug As→Run-time Workbench. A second Eclipse Workbench starts up, containing your plug-ins. Debug this as you would any Java application (see chapter 3).

NOTE If you are using JDK 1.4 or higher, you can make small changes to your source code and rebuild, and the changes will be instantly available in the Run-time Workbench. This is called *hot-swapping* or *hot code replace*. Substantial changes (such as adding a new class) may cause a warning to be displayed. If you get this warning, simply close the Run-time Workbench and run it again.

Figure 8.13 This dialog appears when you click the new toolbar button of the "Hello, World" plug-in.

SIDEBAR Eclipse is *self-hosted*, which means it is used to develop itself. The concept of self-hosting got its start in compiler technology. The first version of a compiler is written in a simpler language, such as assembler, or perhaps using a competitor's product. But once the compiler is working, the developer rewrites it in the language being compiled, using the first version to build the second version, the second version to build the third, and so forth. Eclipse developers use Eclipse the same way, and created its plug-in development environment to support this process.

Compilers, being fairly complex programs in their own right, make excellent test cases for compilers. Likewise, by writing Eclipse with Eclipse, the developers can discover and correct any shortcomings in the handling of large projects and optimize the environment for extending the Eclipse Platform.

8.3.3 Plug-in class (AbstractUIPlugin)

The Hello, World Wizard created three files for you: a plug-in manifest (plugin.xml) and two source files (HelloPlugin.java and SampleAction.java). Because all Eclipse plug-ins and extensions follow this pattern (references in the XML file with Java classes to back them up), it's important to understand how it works. Open the Plug-in Manifest Editor and select the Overview page (figure 8.14).

XML

Switch to the Source page. You should see something like this:

```
<plugin
   id="org.eclipseguide.helloplugin"
   name="Hello Plug-in"
   version="1.0.0"
   provider-name="Eclipse in Action"
   class="org.eclipseguide.helloplugin.HelloPlugin">
...
</plugin>
```

Note that the link to the code is provided by the `class` attribute. When the plug-in is activated, this class will be instantiated and its constructor called.

Java

The class that backs up the plug-in definition in the manifest is `HelloPlugin`, contained in the source file HelloPlugin.java (see listing 8.1). In this section, we'll examine this code and explain how all the pieces fit together.

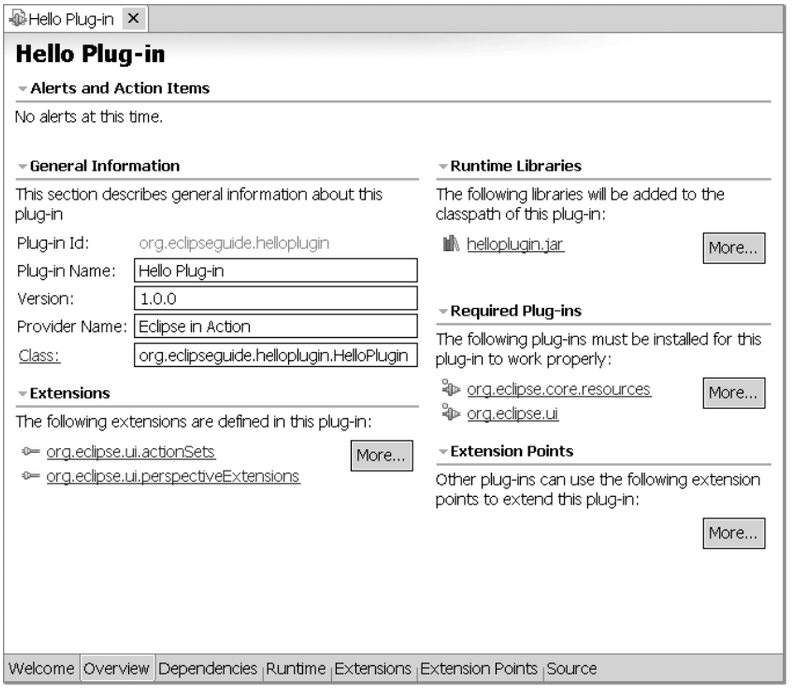

Figure 8.14 The Overview page of the Plug-in Manifest Editor is the central control panel for your plug-in. From here you can get a summary of the plug-in at a glance and access the other pages for more detail.

Listing 8.1 Java class for the "Hello, World" plug-in

```
package org.eclipseguide.helloplugin;                    ❶ Package name

import org.eclipse.ui.plugin.*;
import org.eclipse.core.runtime.*;
import org.eclipse.core.resources.*;
import java.util.*;

/**
 * The main plugin class to be used in the desktop.
 */
public class HelloPlugin extends AbstractUIPlugin        ❷ Plug-in class
{
   //The shared instance.
   private static HelloPlugin plugin;                    ❸ Singleton instance
   //Resource bundle.
   private ResourceBundle resourceBundle;

   /**
```

```
 * The constructor.
 */
public HelloPlugin(IPluginDescriptor descriptor)
{
   super(descriptor);
   plugin = this;
   try
   {
      resourceBundle =
         ResourceBundle.getBundle(
            "org.eclipseguide.helloplugin.HelloPluginResources");
   }
   catch (MissingResourceException x)
   {
      resourceBundle = null;
   }
}

/**
 * Returns the shared instance.
 */
public static HelloPlugin getDefault()
{
   return plugin;
}

/**
 * Returns the workspace instance.
 */
public static IWorkspace getWorkspace()
{
   return ResourcesPlugin.getWorkspace();
}

/**
 * Returns the string from the plugin's resource bundle,
 * or 'key' if not found.
 */
public static String getResourceString(String key)
{
   ResourceBundle bundle =
      HelloPlugin.getDefault().getResourceBundle();
   try
   {
      return bundle.getString(key);
   }
   catch (MissingResourceException e)
   {
      return key;
   }
}

/**
```

❹ **Plug-in constructor**

❺ **Get a resource bundle**

❻ **Return Workspace handle**

❼ **Look up key in resource bundle**

```
     * Returns the plugin's resource bundle,
     */
    public ResourceBundle getResourceBundle()
    {
       return resourceBundle;
    }
}
```

❶ The package name should be the same as the plug-in name, which should be the same as the project name. Packages in the Eclipse Platform start with `org.eclipse`. Although the convention is not always followed, user interface packages generally have `ui` in their name, and non–user interface packages include `core`. If you see a package with `internal` in the name, it is not intended to be used outside the package itself. Internal packages and interfaces can, and often do, change between releases (and even builds of the same release), so stay clear of them.

❷ This is where the plug-in class is created. There are two types of plug-ins: those with user interfaces and those without. `AbstractUIPlugin` is the base class for all UI type plug-ins, and `Plugin` is the base class for the rest.

❸ A Singleton pattern is used to ensure there will be only one instance of the plug-in's class.

❹ The plug-in's constructor is passed an `IPluginDescriptor` object, which has methods such as `getLabel()` that return information from the plug-in registry. You can get a reference to this descriptor later by using the `getDescriptor()` method. Note that all Eclipse interfaces begin with the letter `I`.

❺ See the name of the bundle in the `getBundle()` call? Once created, the properties file for this bundle goes in your org.eclipseguide.helloplugin project and is named HelloPluginResources.properties. You can manage it by hand or by using the Externalize Strings Wizard (Source→Externalize Strings).

❻ The `IWorkspace` interface is the key to the Eclipse Platform's resource management. It has methods to add, delete, and move resources; most important, it has the `getRoot()` method to return the *workspace root* resource, the parent of all the projects in the workspace. This is a Singleton object (only one in the system).

❼ The wizard has created a standard Java resource bundle for you to look up natural language strings. For example, to get the translated string for a greeting, you could call the method `HelloPlugin.getResourceString("%greeting")`.

Actually, two bundles are at work. The first one is associated with the plug-in externally and may be referenced in the plug-in manifest, plugin.xml. Properties for the external bundle are kept in the file plugin.properties. Generally speak-

ing, you will never use that one in the plug-in code. The second bundle, referenced here, is internal to the plug-in and is kept in the plug-in's JAR file.

8.3.4 Actions, menus, and toolbars (IWorkbenchWindowActionDelegate)

An *action* is the non–user interface part of a command that can be run by a user, usually associated with a UI element like a toolbar button or menu. Actions are referenced in the plug-in manifest and defined as Java classes. Figure 8.15 shows what this extension looks like in the manifest editor's Extensions page.

You will probably find using the Extensions page more convenient and less error-prone than editing the XML in the Source page. However, because XML is a more compact representation than a series of screenshots of property pages, we will show the raw XML for most examples in this chapter and the next. Keep in mind, though, that there is a one-to-one correspondence between the two. Also, you can switch back and forth between the pages of the manifest editor at any time; a change in one is reflected in all the others.

XML

The first extension defined by the plug-in is an action set. An *action set* is a menu, submenu, or draggable group of toolbar buttons that appears in the user interface.

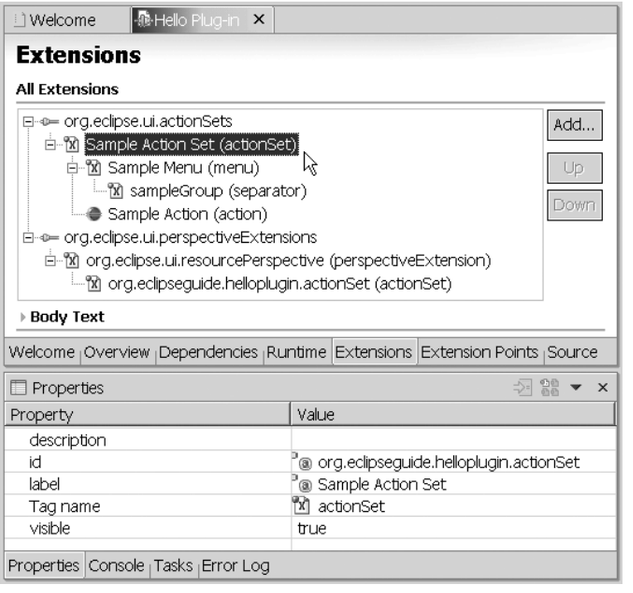

Figure 8.15
You can use the Extensions page of the Plug-in Manifest Editor to add new extensions to your plug-in. Properties and their values are viewed and modified through the Properties view. Required properties such as `id` and `label` are marked with an icon. If you prefer, you can edit the raw XML representation in the Source page.

Listing 8.2 shows the definition in the plug-in manifest (compare this to figure 8.15).

Listing 8.2 The actionSet extension

```
<extension point="org.eclipse.ui.actionSets">
   <actionSet
         label="Sample Action Set"
         visible="true"
         id="org.eclipseguide.helloplugin.actionSet">
      <menu
            label="Sample &amp;Menu"
            id="sampleMenu">
         <separator
               name="sampleGroup">
         </separator>
      </menu>
      <action
          label="&amp;Sample Action"
          icon="icons/sample.gif"
          class="org.eclipseguide.helloplugin.actions.SampleAction"
          tooltip="Hello, Eclipse world"
          menubarPath="sampleMenu/sampleGroup"
          toolbarPath="sampleGroup"
          id="org.eclipseguide.helloplugin.actions.SampleAction">
      </action>
   </actionSet>
</extension>
```

❶ Fully qualified unique ID
❷ Menu ID need not be unique
❸ Placeholder for items/submenus
❹ Points to code
❺ Adds action to menu bar
❻ Adds action to toolbar

❶ Each extension, and indeed just about everything in the plug-in manifest, has a *fully qualified ID* that, by convention, starts with the plug-in ID. It doesn't matter what you call these IDs, as long as you pick unique names.

❷ Of course, there are exceptions, such as menus. They typically use short names like `group1` to achieve some level of consistency between menus. For example, a File menu might have a `group1` section and a Windows menu might also have a `group1` section.

❸ Menus can contain *groups* and *separators*. Separators are simply groups that are drawn with thin lines between them. All menus have a section named `additions`, which is the default place new items are added if you don't specify a location. This particular menu has two levels: `sampleMenu` (the parent menu) and `sampleGroup` (the child group).

❹ As with plug-ins, the `class` property points to the code.

5. The `menubarPath` property indicates the action is being added to a menu bar (in this case, the top-level bar of the Workspace). The paths look like directories, going from higher-level parent menus or groups to lower-level child ones.
6. The `toolbarPath` property indicates that this action is also being added to a toolbar. There is one toolbar called `Normal`, but the name is usually omitted from the path.

Java

Now let's move over to the Java side and dig into the code for the `SampleAction` class, shown in listing 8.3.

Listing 8.3 The SampleAction class

```
package org.eclipseguide.helloplugin.actions;

import org.eclipse.jface.action.IAction;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.ui.IWorkbenchWindow;
import org.eclipse.ui.IWorkbenchWindowActionDelegate;
import org.eclipse.jface.dialogs.MessageDialog;

/**
 * Our sample action implements workbench action delegate.
 * The action proxy will be created by the workbench and
 * shown in the UI. When the user tries to use the action,
 * this delegate will be created and execution will be
 * delegated to it.
 * @see IWorkbenchWindowActionDelegate
 */
public class SampleAction implements IWorkbenchWindowActionDelegate
{
   private IWorkbenchWindow window;
   /**
    * The constructor.
    */
   public SampleAction()
   {
   }

   /**
    * The action has been activated. The argument of the
    * method represents the 'real' action sitting
    * in the workbench UI.
    * @see IWorkbenchWindowActionDelegate#run
    */
   public void run(IAction action)
   {
      MessageDialog.openInformation(
         window.getShell(),
```

1. JFace and UI imports
2. Code pointed to by manifest
3. Main Workbench window
4. Perform action
5. Open dialog

```
            "Hello Plug-in",
            "Hello, Eclipse world");
    }

    /**
     * Selection in the workbench has been changed. We
     * can change the state of the 'real' action here
     * if we want, but this can only happen after
     * the delegate has been created.
     * @see IWorkbenchWindowActionDelegate#selectionChanged
     */
    public void selectionChanged(
       IAction action,
       ISelection selection)
    {
    }

    /**
     * We can use this method to dispose of any system
     * resources we previously allocated.
     * @see IWorkbenchWindowActionDelegate#dispose
     */
    public void dispose()
    {
    }

    /**
     * We will cache window object in order to
     * be able to provide parent shell for the message dialog.
     * @see IWorkbenchWindowActionDelegate#init
     */
    public void init(IWorkbenchWindow window)
    {
       this.window = window;
    }
}
```

❻ **Can be used to free system resources**

❶ JFace is a high-level wrapper on top of the Standard Widget Toolkit (SWT) used for the Eclipse UI. JFace deals in concepts like dialogs and viewers, whereas SWT deals in windows, canvases, and buttons. (For more details on SWT and JFace, see appendixes D and E.)

❷ A *proxy* stands in for an object until the real object is available. In this case, there is a proxy for the toolbar button (not seen here) created by the Workbench based solely on the information in the plug-in manifest. Remember that the plug-in (including this code) isn't even loaded until after the button is clicked. When the user finally clicks the button, this *delegate* is created and passed the action to run. It works just like a relay race. The first runner is the proxy, and the baton he car-

ries is the action. The baton is passed to the second runner, the delegate, who finishes the race.

Because the "Hello, World" button is in the Workbench toolbar and the menu item is in the main Workbench menu, this code implements the Workbench window action delegate interface. There are similar interfaces for View, Editor, and Object action delegates, which are all based on `IActionDelegate`. `IActionDelegate` only has two methods—`run()` and `selectionChanged()`—which you will implement a little further down in this class.

❸ `IWorkbenchWindow` is an interface used for the top-level window of the Workbench. It contains a collection of `IWorkbenchPages` that in turn hold all the views, editors, and toolbars. Some common methods you'll use in `IWorkbenchWindow` include `close()` and `getWorkbench()`.

❹ The `run()` method is where the actual work gets done. The `IAction` interface has methods for getting and setting the user interface style of the button or menu it is associated with, and maintains a list of *listeners* that are called when any of its properties change.

❺ `MessageDialog` is one of a group of JFace utility classes that perform common operations. The static method `openInformation()`, as you might guess, opens an information dialog (as opposed to an error, warning, question, or other type of dialog). Its first argument is a *shell*, which is a low-level SWT window. (You'll find that many classes have a `getShell()` method, and you will use it often.) The `openInformation()` method's second argument is the title of the dialog that will be shown, and the final argument is the text that will be displayed on the main area of the dialog.

❻ Because SWT works more closely to the underlying window system than other APIs (notably Swing), it is sometimes necessary to free up system resources in `dispose()` methods that are explicitly called. Garbage collection cannot be relied on for this purpose because it is run at unpredictable times. This is one of the more controversial requirements of SWT, but it is not as painful as you might think.

8.3.5 Plug-ins and classpaths

One "gotcha" that continues to bite plug-in developers (new and old alike) is the way plug-in classpaths work. Plug-ins can only use classes exported by other plug-ins. For security reasons, plug-ins ignore the normal classpath settings at runtime, causing `ClassNotFound` exceptions even when the code compiled just fine.

Because of this restriction, if you want to use an external JAR file (one that is not in your workspace) inside a plug-in, you must bring it into your workspace.

Typically you do so by wrapping the JAR file in its own plug-in and making any other plug-ins that need the library depend on the new plug-in. The Eclipse Platform includes many examples, such as the `org.junit`, `org.apache.ant`, and `org.apache.xerces` plug-ins, which are simple wrappers around the JUnit, Ant, and Xerces libraries, respectively.

Wrapping a JAR file is one of the simplest plug-ins you can create, because no code is involved. The next example walks you through the necessary steps.

8.4 The log4j library plug-in example

As you recall from chapter 3, log4j is a free logging API created for the Apache Jakarta project. When you wanted to use it in a normal Java program from within Eclipse, you created a classpath variable for it and then referenced that variable inside the Java Build Path for the project. But let's say you need to use the library inside a plug-in, so you need to create a wrapper plug-in for it. To create the wrapper for log4j, start with a blank template from the New Plug-in Project Wizard:

1. Select File→New→Project to bring up the New Project Wizard (figure 8.3).
2. Select the Plug-in Project Wizard and click Next to open the New Plug-in Project Wizard.
3. Enter the name for the plug-in, **org.apache.log4j**, and click Next.
4. Leave the fully qualified ID as it is, making sure the option to Create a Java Project is selected, and click Next again.
5. Select the option to Create a Blank Plug-in Project (see figure 8.6). Click Finish to generate the plug-in.

Now, you need to customize the plug-in to contain the log4j library and include the proper export instructions so other plug-ins can use it. To do this, follow these steps:

1. In the new project directory, delete the src directory (right-click on it and select Delete), because there will be no source code in the project itself.
2. Copy the log4j JAR file (for example, log4j-1.2.8.jar) from the place you installed it in chapter 3 into the top level of the project directory. To do this, you can use File→Import→File System or, if you're using Windows, drag the file from your file explorer into the project.
3. Rename the JAR filename to remove the version number by right-clicking on it, selecting Refactor→Rename, and entering the new name, **log4j.jar**. The version number is specified in the plug-in manifest and is

appended to the plug-in directory name, so you don't also have to append it to the JAR filename.

4. Edit the project properties (right-click on the project and select Properties). Select Java Build Path, and then select the Libraries tab. Click Add Jars, navigate into the project, and select log4j.jar. Click OK.
5. While still in the project properties dialog, select the Order and Export tab. Put a check mark next to log4j.jar and click OK to save. This setting lets other plug-ins use this library at compile time.
6. Open the Plug-in Manifest Editor (double-click on plugin.xml) and switch to the Overview page. Set the Plug-in Name to **Apache Log4J**, change the Version to match the version number of the log4j package (for example, **1.2.8**), and fill in the Provider Name with **Eclipse in Action**.
7. Still in the manifest editor, switch to the Runtime page. Verify that log4j.jar is in the library list. Select it and turn on the option to Export the Entire Library. This setting lets other plug-ins use the library at runtime. You can also use this trick to make it look like classes from many other plug-ins come from a single plug-in.
8. Delete the reference to the src/ folder on the Runtime page under Library Content by right-clicking on it and selecting Delete. Again, because you are not building the plug-in from source code, you don't need a source folder.
9. Switch to the Source page of the manifest editor and admire your handiwork. When you are done, the XML in the Source page should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
   id="org.apache.log4j"
   name="Apache Log4J"
   version="1.2.8"
   provider-name="Eclipse in Action">

   <runtime>
      <library name="log4j.jar">
         <export name="*"/>
      </library>
   </runtime>
</plugin>
```

10. Press Ctrl-S to save, and then close the Plug-in Manifest Editor.

8.4.1 Attaching source

Users of your plug-in will undoubtedly want to view log4j's source code at some point, perhaps while debugging or in order to understand how to use its classes. To make that functionality available, you have to place a zip file containing the source in the top-level directory of the plug-in (zip is used even on UNIX). In order for Eclipse to find the zip file automatically, it must have same name as the JAR file, but with src.zip appended to the end (for example, foosrc.zip goes with foo.jar). In normal plug-ins, the PDE makes this file for you from your own source code. But because you are not building the code for this library, you must make other arrangements:

1. Create a log4jsrc.zip file containing the source. The log4j distribution doesn't include a source zip file, but it does have a directory containing the source. Locate the top of the source tree in the distribution's src/java directory and put the org subdirectory and everything under it into the zip using the `jar` utility (`jar -cvf log4jsrc.zip`) or your favorite zip program, such as WinZip. When you're done, the zip file must have an internal structure like this:

```
org
+---apache
    +---log4j
        +---chainsaw
        +---config
        +---helpers
        +---etc...
```

2. Copy the new log4jsrc.zip file into your project (using File→Import→File System).
3. Associate the source zip to the log4j JAR file by right-clicking on log4j.jar and selecting Properties→Java Source Attachment (see figure 8.16). Click the Workspace button to locate the zip file. Doing so lets you view the source in your own plug-in projects.

8.4.2 Including the source zip in the plug-in package

When the time comes to make your plug-in available for other people to use, you need to package it in a zip file organized exactly as it should be organized under the plugins directory. Eclipse uses the properties in build.properties to tell it which files should be packaged and which ones should be ignored. Obviously, you want the source zip to be included, so follow these steps:

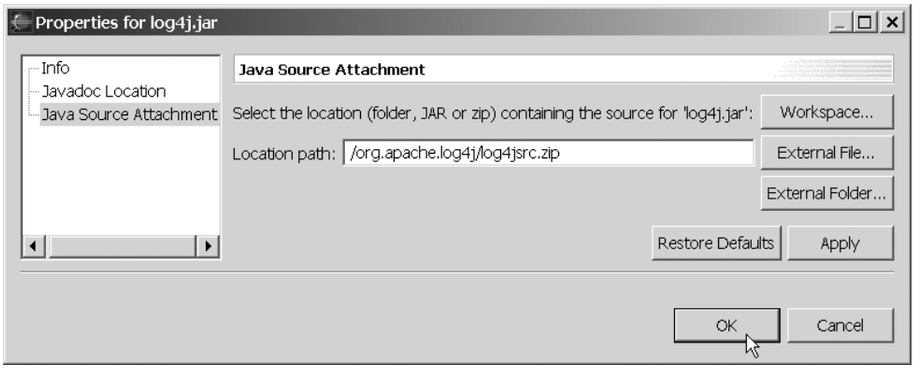

Figure 8.16 Set the Java Source Attachment property on a JAR file to make its source visible in Eclipse. The recommended method is to click the Workspace button to locate a path relative to the workspace, as shown here.

1. Open the Properties Editor on build.properties by double-clicking on it. Note that you can't edit build.properties and plugin.xml at the same time, because the manifest editor needs to write the properties file. So if you get an error that the file is in use or read only, close the manifest editor and try opening build.properties again.
2. Add the zip file to the `bin.includes` property. To do this, select the `bin.includes` property on the left to display the values for that property on the right. Click the Add button under Replacement Values and type **log4jsrc.zip**. Later, when it's time to deploy the plug-in, this property will be used to pick which files from your project are included. Internally, `bin.includes` is a variable name used in an Ant script that does the deployment.

TIP You can use Ant patterns to include many files at once. For example, use a wildcard like ***.jar** to include all files ending with *.jar*. The pattern ****** (for example, ****/*.gif**) matches any number of directory levels, and a trailing slash (for example, **lib/**) matches a whole subtree.

---

3. While you're editing the properties file, remove the `source.log4j.jar` property by right-clicking on it and selecting Delete. Because there is no source code in the project except the zip file you just imported, this property is unnecessary. Press Ctrl-S to save. The build.properties file should now look like figure 8.17.

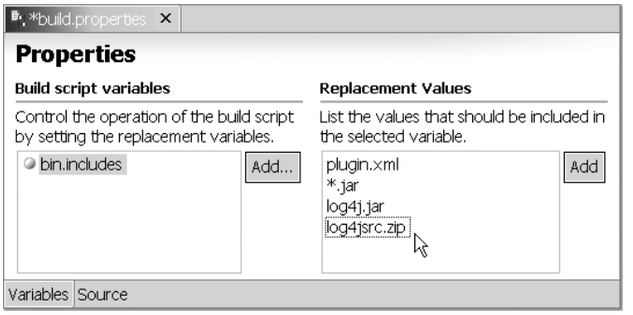

Figure 8.17 Add the zip file containing the source to the `bin.includes` property. Everything listed here will be included in the final plug-in package when the time comes to deploy it.

There you have it—a plug-in that wraps the log4j JAR file and that can be referenced from other plug-ins. You'll use this plug-in for the examples in chapter 9.

8.5 Deploying a plug-in

Once you've created a plug-in in your workspace, you can run and debug it using the Run-time Workbench. But how do you install the plug-in or give it to someone else so they can install it? This process is called *deployment*. You can create deployable zip files with Ant, but the PDE supplies an Export Wizard to make it even easier. To demonstrate this process, let's create the zip file for the log4j library plug-in you just built:

1. Select File→Export to start the Export Wizard, and then select Deployable Plug-ins and Fragments and click Next. The Export dialog shown in figure 8.18 opens.
2. Select the plug-in(s) to export and enter the filename of the zip file you want to create.
3. Click Finish to create the file.

Now you have a plug-in zip file that others can install.

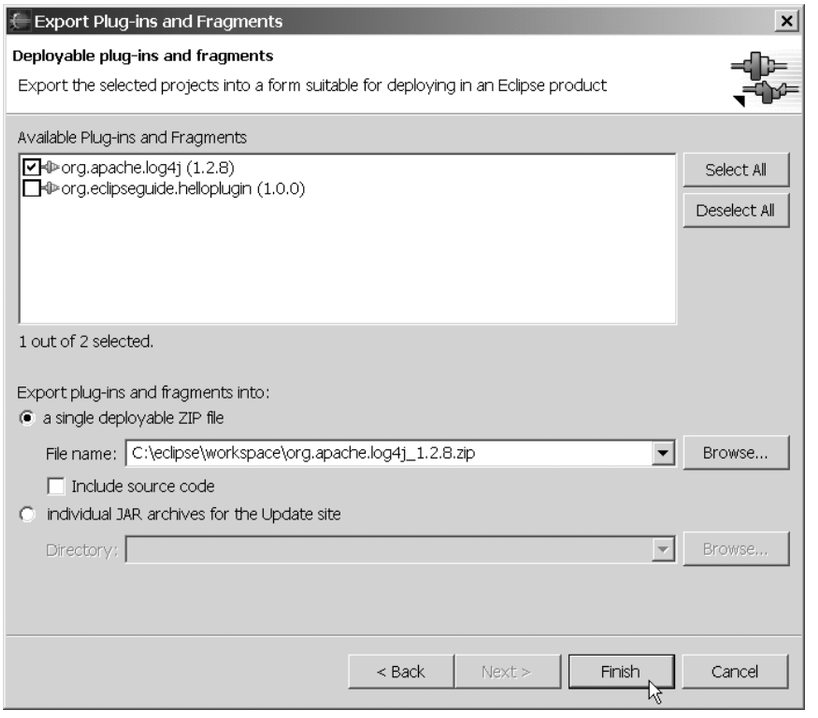

Figure 8.18 You can use the Deployable Plug-ins and Fragments Wizard to create zip files that others can install. Specify the plug-in and the name of the zip file to create and click Finish to create the file.

8.6 Summary

Every component of the Eclipse Workbench—every view, every editor, every menu—is defined in a plug-in. The Eclipse designers took great care to expose a fully functional public API for all plug-in writers to use. Because of this even playing field, high quality plug-ins you provide cannot be distinguished from plug-ins that were originally part of the Platform.

The convergence of an object-oriented polymorphic introspective language (Java), a universal data exchange format (XML), open-source tools (Ant, JUnit), design patterns, and agile programming techniques (such as refactoring) make Eclipse a unique and fun environment in which to program. Wizards and templates greatly lessen the learning curve, and the open source community built around Eclipse provides plenty of examples (and support) for the Eclipse programmer.