CHAPTER 1

The Big Picture

The typical relationship between the programmer and the programming environment is one where the environment nurtures the programmer. In return for this support, the programmer works within the constraints created by the environment.

In the early seventies, Smalltalk was written with a different philosophy. Every user of every application was considered a potential programmer. The classic example is, you could be editing a document and decide you didn't like how the editor worked. Pressing a button would show you the inner workings of the document editor and provide you with tools to change the editor's behavior.

User-as-programmer may seem far-fetched, but Smalltalk saw many cases where non-programming professionals wrote applications uniquely suited to their own needs by learning programming a little at a time and by example. The key to enabling user programming is to structure the environment to encourage learning, giving the user a little payoff for a little investment and a little more payoff for a little more investment.

Eclipse structures the computing experience similarly. Its goals are much the same as the goals of Smalltalk: Give the users an empowering computing experience and provide a learning environment as a path to greater power. In Eclipse, moving up the pyramid requires investment and will be attempted by fewer people (see Figure 1.1).

- **Users—**Daily Eclipse users. Currently this is restricted to programmers, but there is no reason in principle why Eclipse couldn't be used to structure other computing work.

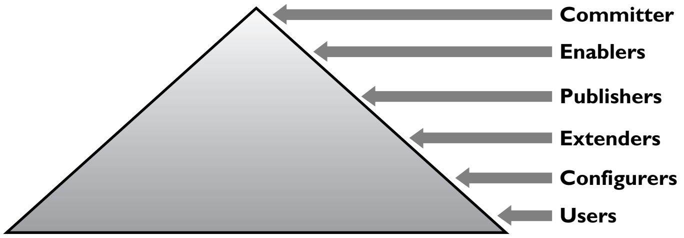


Figure 1.1 Increasing Commitment and Reward

- **Configurers—**Users who customize their experience of Eclipse, either by rearranging perspectives, setting preferences, or deciding which views to show. Configuration is limited to changes envisioned by the original programmer.
- **Extenders—**Programmers who make changes not envisioned by the original Eclipse programmers. Eclipse provides a rich set of places to "plug in" new functionality, because Eclipse itself is built entirely by plugging in functionality.
- **Publishers—**Once you've written something useful, other folks may want it, too. Eclipse is consciously structured so you can easily bundle together extensions so others can load them.
- **Enablers—**Eclipse is built out of "places-to-plug-functionality-in" (extension points) and "functionality-to-be-plugged-in" (extensions). Once you have published a contribution, the next step is to enable others to extend it in ways you don't foresee. You do this by publishing your extension points.
- **Committers—**Eclipse is an open source project. If there is a change you want to make that is outside the scope of the available extension points, you can change the source code itself. Getting your changes incorporated into the global Eclipse release requires that you gain the trust of the existing community of committers. Becoming a committer is outside the scope of this book, although we will look at lots of the Eclipse source code so you can get an idea of what would be involved.

We can also map these levels onto a circle as shown in Figure 1.2.

What makes this circle interesting is the final arrow, from Enabler back to User. In Eclipse, you don't just invest more and more and receive more and more. When you take the step to become an enabler with your own extension points, you create for yourself the opportunity to be nourished by the work of others. Sometime later, someone may extend your contribution in ways you

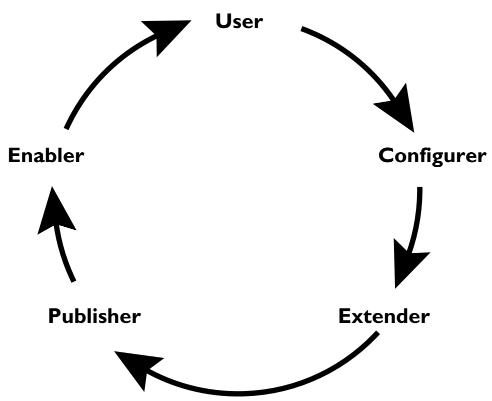


Figure 1.2 The Contribution Circle

find useful. By creating and sharing your own extension points you can get the benefit of their work without further effort on your part.

We intend for this book to be your guide around the Contribution Circle.

1.1 Book Goals

We've been talking about writing a book together for nearly a decade. We share an interest in software design, in the design of software for widespread adaptation (frameworks), and in discovering the "universal" rules that lie behind design. We first met at Bruce Anderson's Architecture Handbook workshop at OOPSLA-93 in Washington, D.C, where we began discussing software design. We discovered that, while we generally shared a common aesthetic of design, we disagreed in enough interesting ways to fuel years of debate and joint exploration. The chance to write this book gave us a good reason to continue and deepen our discussion of design.

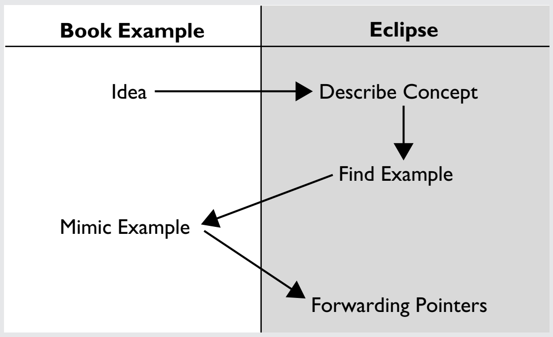
We want to help you around the Contribution Circle as far as you want to go. For you to learn your way around the circle to become a enabler, we need to help you learn two things:

❍ *About Eclipse.* There is far more to Eclipse than would fit into any ten books, but there are some basic concepts and details you simply cannot live without. By following the examples here, you will see these basics in action.

❍ *Learning about Eclipse.* We aren't going to pretend that we have memorized all the details of Eclipse. When we program in Eclipse, we search for examples from which to copy. As we develop our code, we'll tell you how we found our examples.

- *Design principles.* Okay, this is our secret third agenda. As we go along, we'll tell you about the underlying principles of design behind the structure of Eclipse in particular, but also behind platforms of all kinds.

Pedagogical Structure



When we want to describe an Eclipse concept, we will always place it in the context of our example application. We'll start with the idea of what feature we want to add, describe the general concept in Eclipse that fits the idea best, show an example in Eclipse that uses that concept, show how the concept is used in our example, then give forwarding pointers to other places in Eclipse you can look for more examples. This is how we develop in Eclipse ourselves, by copying examples, so it's good practice to see it here.

The book is organized as four concentric circles, each taking you around the Contribution Circle in increasing detail.

1. Circle Zero, Chapter 2 and Chapter 3, gets you set up for plug-in development and takes you as quickly as possible from a tiny plug-in idea to its implementation.
2. Circle One, Chapter 4 through Chapter 12, introduces the content of our plug-in and running test cases, and makes and deploys the simplest possible test-running plug-in.
3. Circle Two, Chapter 13 through Chapter 30, takes the basic test-running plug-in and adds all the capabilities expected of Eclipse contributions, acquiring an interesting twist of metaphor halfway through.

4. Circle Three, Chapter 31 through Chapter 37, tours Eclipse designer-to-designer, highlighting areas of Eclipse worth exploring early. These essays use design and implementation patterns as their basic vocabulary, showing how the patterns play out in a variety of contexts.

1.2 Plug-In

Because this chapter and the following chapters constitute a complete circle, we should have a little bit of background about the plug-in architecture of Eclipse. Eclipse is a collection of places-to-plug-things-in (extension points) and things-plugged-in (extensions). The powerstrip is a kind of extension point. Multiple extensions (in this case, power plugs) can plug into it, and although the extensions are different shapes and have different purposes, they all must share a common interface.

1.3 Eclipse in a Nutshell

Here is a bit of the Eclipse architecture to get you started. Then we will start programming. We get back to some key architectural elements of Eclipse in Circle Three. If you are more comfortable having an overall picture before looking at details, you may want to read Circle Three now.

Figure 1.3 shows the three layers of Eclipse.

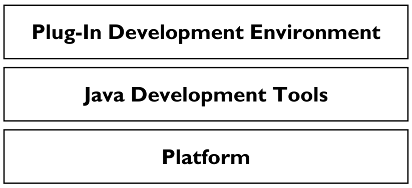


Figure 1.3 The Three Layers of Eclipse

- **Platform—**The Eclipse platform defines the common programming-language-neutral infrastructure.
- **Java Development Tools (JDT)—**The Java development tools add a full-featured Java IDE to Eclipse.

- **Plug-In Development Environment (PDE)—**The PDE extends the JDT with support for developing plug-ins.

The platform consists of several key components that are layered into a user interface (UI)-independent core and a UI layer, as shown in Figure 1.4.

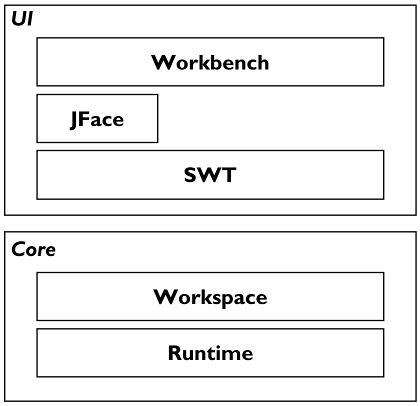
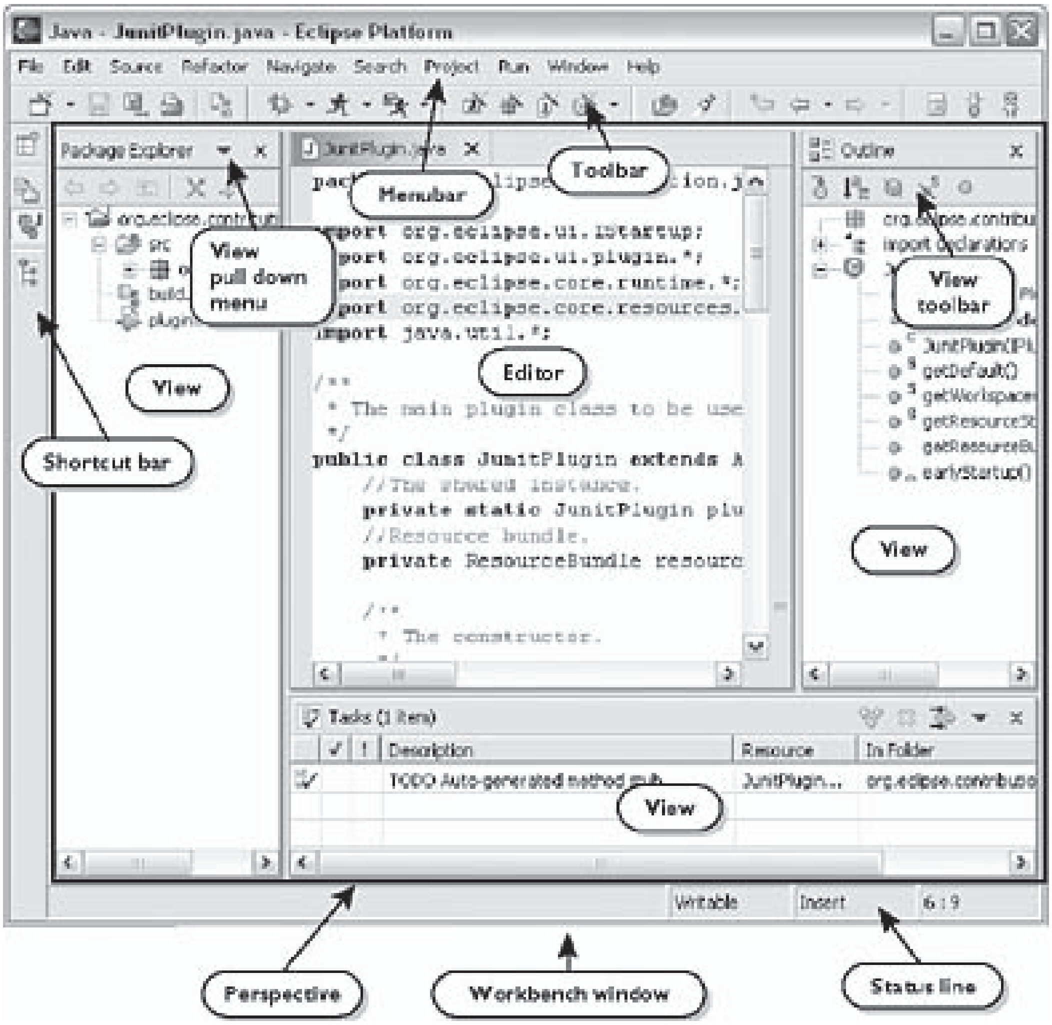


Figure 1.4 Eclipse Architecture Overview

- **Runtime—**The run-time component defines the plug-in infrastructure. It discovers the available plug-ins on start-up and manages the plug-in loading.
- **Workspace—**A workspace manages one or more top-level projects. A project consists of file and folders that map onto the underlying file system.
- **Standard Widget Toolkit (SWT)—**The SWT provides graphics and defines a standard set of widgets.
- **JFace—**A set of smaller UI frameworks built on top of SWT supporting common UI tasks.
- **Workbench—**The workbench defines the Eclipse UI paradigm. It centers around editors, views, and perspectives.

Let's take a brief look at the workbench and its components, shown in Figure 1.5.



```
pac
    port org.eclipse.ui.IStartup;
    port org.eclipse.ui.plugin.*;
    port org.eclipse.core.runtime.*;
    port org.eclipse.core.resources.
import java.util.*;

/**
 * The main plugin class to be use
 */
public class JunitPlugin extends A
    //The shared instance.
    private static JunitPlugin plu
    //Resource bundle.
    private ResourceBundle resourc

    /**
     * The constructor.
```

✓	!	Description	Resource	In Folder
		TODO Auto-generated method stub	JunitPlugin...	org.eclipse.contributio

Figure 1.5 Eclipse User Interface Vocabulary

The Eclipse workbench is presented in one or more *windows*. A workbench window contains a set of workbench *parts*. These can be either *views* or *editors*. A *perspective* defines the visual arrangement of the workbench parts.

Finally, since you are about to start extending Eclipse, you should familiarize yourselves with the Eclipse UI guidelines.[1]

1. *www.eclipse.org/articles/Article-UI-Guidelines/Index.html*

PART I

Circle Zero: Hello World

This circle will take you through the entire contribution process as quickly as possible. We will skip many details in the name of brevity, but fortunately we have three more circles in which to make up for our haste. Our goal is to get you over the inertia of not having made a contribution. Once you've contributed to Eclipse, you'll be ready to learn to contribute well.

Our example will contribute a button to the toolbar. When we press the button, a dialog will appear announcing, "Hello World." First, though, we need to set Eclipse up for plug-in development.

CHAPTER 2

Setting Up Eclipse for Plug-In Development

Developing plug-ins is a little different than developing vanilla Java applications. First, you need to be able to refer to Eclipse's own internal structure to extend that structure. Second, you will spend much of your early hours as an Eclipse contributor reading source code, so you need access to all the source.

One way to solve these problems is to load the entire source (around one million lines of code) into the workspace. Managing this huge amount of source can expand the memory footprint of Eclipse. However, we will typically only read existing code and not modify or compile it. Fortunately, PDE offers a quick and space-efficient way to set up a workspace where existing plug-ins cannot be modified, but can be browsed. The trick is to represent existing plug-ins as *binary projects*. Binary projects cannot be modified, but are fully searchable for references and declarations.

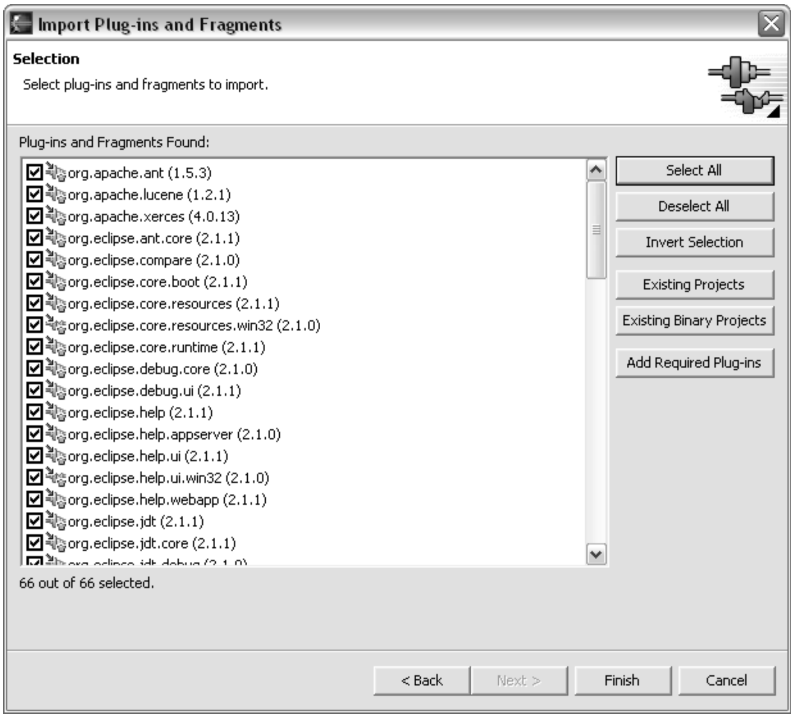
The distinction is between *working with* and *working on*. A project you are working *with* can be represented as a binary project, saving space. A project you are working *on* must be represented as source.

2.1 Setting Up a Workspace

Here is how to set up a workspace with binary projects for all the plug-ins shipped with Eclipse:

1. If you don't want to pollute your existing workspace, start Eclipse with an empty workspace (use `-data new_workspace_location` on the command line).

2. Switch to the Java perspective by selecting **Window > Open Perspective > Java**.
3. Choose **File > Import... > External Plug-ins and Fragments**. Accept the defaults on the next page. On the third page click **Select All** to import all plug-ins from your host workspace.
4. Click **Finish**. The plug-ins will be imported into binary projects, their build class paths initialized, as shown in Figure 2.1.



☑ org.apache.ant (1.5.3)
☑ org.apache.lucene (1.2.1)
☑ org.apache.xerces (4.0.13)
☑ org.eclipse.ant.core (2.1.1)
☑ org.eclipse.compare (2.1.0)
☑ org.eclipse.core.boot (2.1.1)
☑ org.eclipse.core.resources (2.1.1)
☑ org.eclipse.core.resources.win32 (2.1.0)
☑ org.eclipse.core.runtime (2.1.1)
☑ org.eclipse.debug.core (2.1.0)
☑ org.eclipse.debug.ui (2.1.1)
☑ org.eclipse.help (2.1.1)
☑ org.eclipse.help.appserver (2.1.0)
☑ org.eclipse.help.ui (2.1.1)
☑ org.eclipse.help.ui.win32 (2.1.0)
☑ org.eclipse.help.webapp (2.1.1)
☑ org.eclipse.jdt (2.1.1)
☑ org.eclipse.jdt.core (2.1.1)

66 out of 66 selected.

Figure 2.1

2.2 Browsing and Searching Source

You can now browse and search the full Eclipse source. Choose **Navigate > Open Type...** and type in `*`. You should see thousands of classes. You will have around 60 Eclipse plug-in projects in your workspace.

If you do not want to see these binary projects, hide the binary projects in the **Package Explorer** by turning on a filter for binary projects. Select **Filters** from the **Package Explorer** view pull-down menu, as shown in Figure 2.2 and

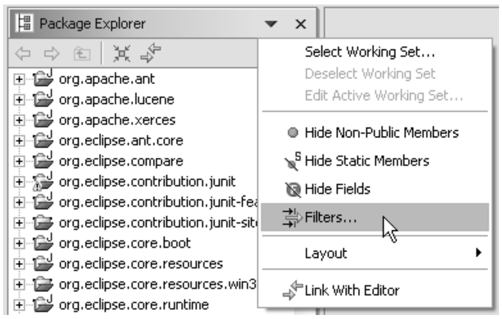
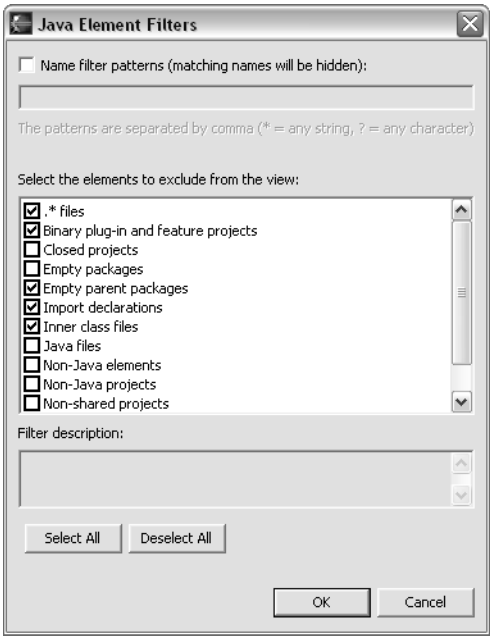


Figure 2.2

select **Binary plug-in and feature projects** in the subsequent dialog, as shown in Figure 2.3.

If you started Eclipse in a clean workspace, the **Package Explorer** will be empty after you invoke the filter because you haven't created any source plug-in projects yet.

One thing to keep in mind when you install a new version of Eclipse: Do not forget to reimport the plug-ins that come with this build. To do so, click the **Existing Binary Projects** button on the import wizard's plug-in selection


Java Element Filters

☐ Name filter patterns (matching names will be hidden):

The patterns are separated by comma (* = any string, ? = any character)

Select the elements to exclude from the view:

- ☑ .* files
- ☑ Binary plug-in and feature projects
- ☐ Closed projects
- ☐ Empty packages
- ☑ Empty parent packages
- ☑ Import declarations
- ☑ Inner class files
- ☐ Java files
- ☐ Non-Java elements
- ☐ Non-Java projects
- ☐ Non-shared projects

Filter description:

Figure 2.3

page. This will replace your existing binary projects with the code for the new versions.

Now Eclipse is ready to say "hello" to the world.

2.3 Forward Pointers

❍ *Workspace set up with binary projects but linked contents*—If you have a large number of plug-ins and you don't want to create a copy of all plug-ins in your workspace, you can uncheck the **Copy plug-in contents into the workspace location** check box when you import the plug-ins. In this case, PDE creates the plug-in projects in your workspace, but it "links" their contents instead of making a copy.

CHAPTER 3

Hello World

Our "Hello World" plug-in will contribute a button to the toolbar. When the button is pressed, we will pop up a dialog box containing the text "Hello World."

The developers at Eclipse.org share a consistent set of rules for design. Here's the first rule of Eclipse:

CONTRIBUTION RULE Everything is a contribution.

The whole of Eclipse—the Java development tools, the CVS repository explorer, every single tool—is contributed. That is, none of them is "built into" Eclipse. There is no monolithic tool to which a few things are added. There is a tiny little kernel to which many things are contributed, as shown in Figure 3.1.

As a consequence of making everything a contribution you will have lots of contributions. The Java environment and the Eclipse base together are more than 60 large plug-ins. The IBM WebSphere Application Development environment, for example, adds another 500 plug-ins. Assume that you have a system built out of thousands of contributions. If you want the system to start up this century, you can't do much work per contribution on start-up. In particular, the end user should not pay in start-up time for plug-ins that are installed but not used.

While we speak of performance as being the last thing you should pay attention to in development, performance often has profound impact on the architecture. Eclipse is shaped by the need to process thousands of contributions at start-up, yielding a budget of a few milliseconds each.

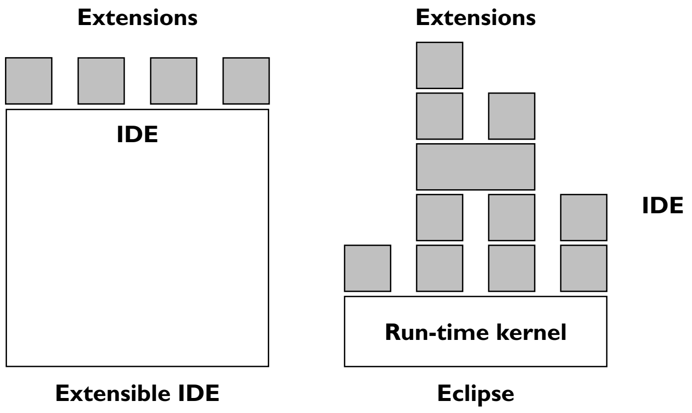


Figure 3.1 Some Extensions Versus All Extensions

Now we come to a dilemma. The logic contained within a contribution can be substantial. The compiled form of this logic, Java class files (collected in a Java Archive [JAR] file), can easily take seconds to load. Making Java classes that are guaranteed to load quickly is more work than most people want to do. If we want to guarantee snappy start-up, we can't load classes. This leads to the Lazy Loading Rule:

> **LAZY LOADING RULE** Contributions are only loaded when they are needed.
>
> Good rule, but how is it implemented?

3.1 Declaration/Implementation Split

If you only know which contributions are present, even if you haven't loaded their implementation you can already give the user a picture of what operations are available. The plug-in architecture implements this split between declaration and implementation by declaring the "shape" of a contribution in an Extensible Markup Language (XML)-based manifest. The implementation of the contribution is in Java (see Figure 3.2).

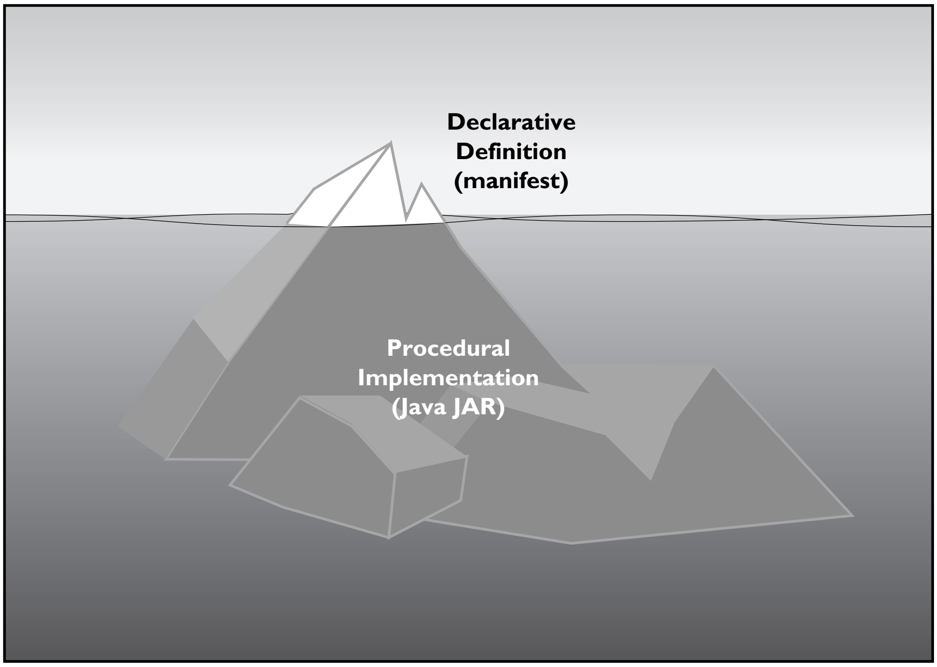

Declarative Definition (manifest)
Procedural Implementation (Java JAR)

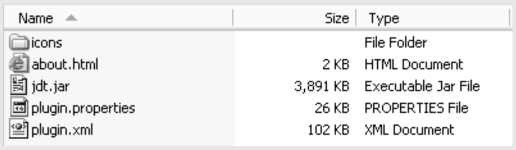
Figure 3.2 The Manifest Describes Your Plug-In's Contribution

The Basic Plug-In Structure

A plug-in is a piece of behavior that is outside the run-time kernel. A plug-in is represented as a directory containing

- plugin.xml—The manifest, a description of the contributions of the plug-in
- Resources, like icons (optional)
- Java code, in a JAR (optional)

The directory structure for the plug-in `org.eclipse.jdt.ui` is shown below.



Name	Size	Type
icons		File Folder
about.html	2 KB	HTML Document
jdt.jar	3,891 KB	Executable Jar File
plugin.properties	26 KB	PROPERTIES File
plugin.xml	102 KB	XML Document

3.2 Hello Button

We want to contribute a button to Eclipse. First we have to create a project. Eclipse has wizards to automate tedious work, but if you haven't done it before, the work is hardly tedious. We'll build our examples as much as possible by hand, then you can use the wizards once you understand what they are doing for you. We'll only use the wizard to give us the basic structure, and we'll explain what it did for us (Figures 3.3–3.7).

1. Open the new wizard.

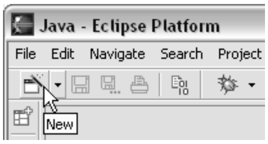


Figure 3.3

2. Select **Plug-in Project**.

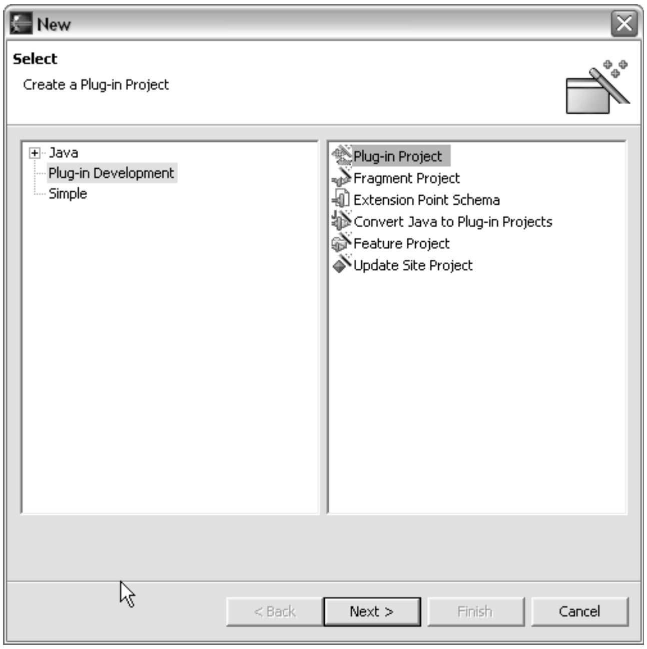
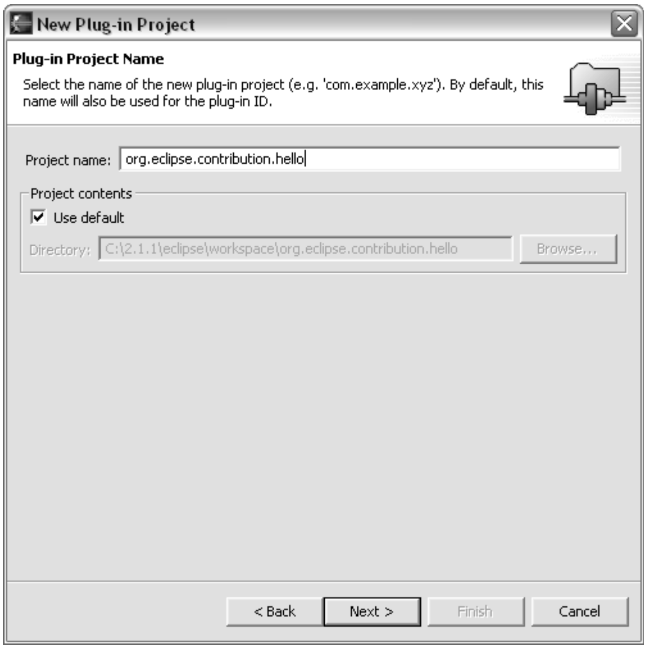


Figure 3.4

3. Define the name of the plug-in project.


New Plug-in Project

Plug-in Project Name

Select the name of the new plug-in project (e.g. 'com.example.xyz'). By default, this name will also be used for the plug-in ID.

Project name: org.eclipse.contribution.hello

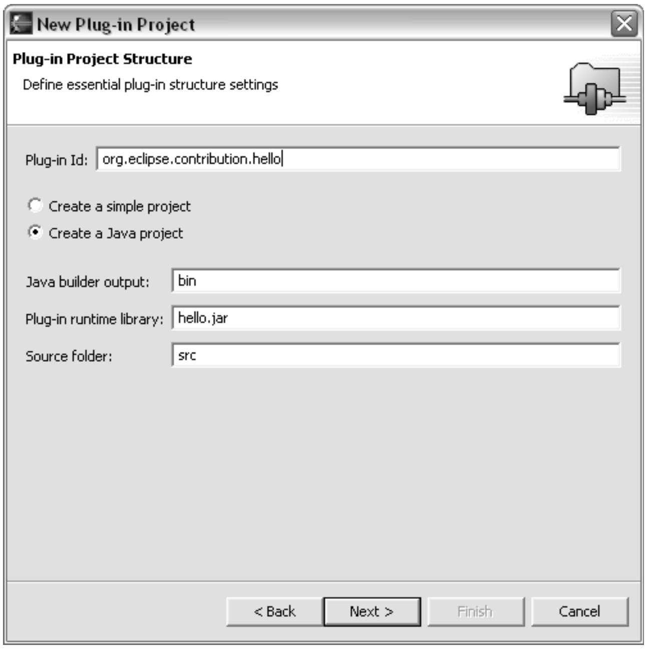
Project contents

[✓] Use default

Directory: C:\2.1.1\eclipse\workspace\org.eclipse.contribution.hello

Figure 3.5

4. Define the settings related to the plug-in structure.


New Plug-in Project

Plug-in Project Structure

Define essential plug-in structure settings

Plug-in Id: org.eclipse.contribution.hello

○ Create a simple project
● Create a Java project

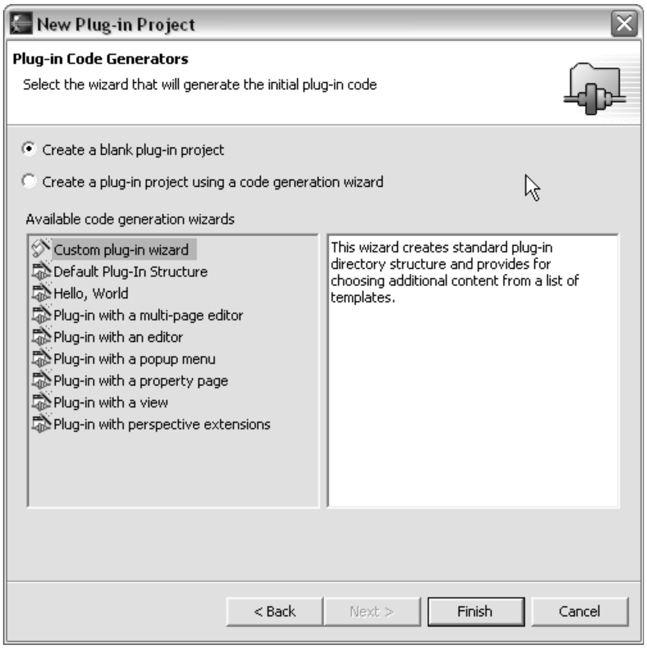
Java builder output: bin

Plug-in runtime library: hello.jar

Source folder: src

Figure 3.6

5. Create a blank plug-in project without using any of the code generation wizards.


New Plug-in Project
Plug-in Code Generators
Select the wizard that will generate the initial plug-in code
Available code generation wizards
< Back
Next >
Finish
Cancel

(•) Create a blank plug-in project
( ) Create a plug-in project using a code generation wizard

Custom plug-in wizard
Default Plug-In Structure
Hello, World
Plug-in with a multi-page editor
Plug-in with an editor
Plug-in with a popup menu
Plug-in with a property page
Plug-in with a view
Plug-in with perspective extensions

This wizard creates standard plug-in directory structure and provides for choosing additional content from a list of templates.

Figure 3.7

Note that on the last step we selected **Create a blank plug-in project** so we could have the joy of filling in as many details as possible by hand. When you click **Finish**, answer **No** to the dialog asking you if you want to switch to the plug-in development perspective. We prefer to do our plug-in development in the Java perspective.

The result is the basic plug-in project structure shown in Figure 3.8.

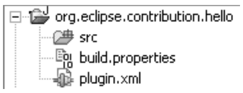


Figure 3.8 Plug-In Project Structure

We have a nearly empty manifest file describing the appearance and structure of the plug-in, a *build.properties* file that tells the Plug-in Development Environment where to find the source code for building the JAR, and an empty *src* folder.

The manifest file, *plugin.xml* (shown in its entirety in Section 3.3.2), contains the top-level plug-in description. Opening *plugin.xml* will show you the manifest editor, a friendly interface to all the details in the manifest. Since we want to explore the whole manifest, click on the **Source** tab to show the raw XML. You'll see something like this:

org.eclipse.contribution.hello/plugin.xml
```
<plugin
  id="org.eclipse.contribution.hello"
  name="org.eclipse.contribution.hello"
  version="1.0.0">
</plugin>
```

You'll also see a run-time entry in the manifest that defines where Eclipse will look for classes in this plug-in.

IDs in manifests are globally unique. Names, however, are expected to be read by humans. Our first change is to make the name of our plug-in a bit friendlier:

org.eclipse.contribution.hello/plugin.xml
```
<plugin
  id="org.eclipse.contribution.hello"
  name="Hello World"
  version="1.0.0">
```

The version is mandatory because plug-ins can rely on each other (remember that nearly everything is a plug-in) and you can refer to the particular version of a plug-in you depend on.

Plug-In Development Environment (PDE)

Because plug-ins are so important to Eclipse, Eclipse has evolved tools for developing Java projects that are plug-ins. You will see a wizard for creating plug-in projects, specialized editors for the manifest file *(plugin.xml)*, and support for running a second workbench with plug-ins under development.

Running with PDE will be confusing at times because you have to remember whether you are working in the workbench that is editing the plug-in (host workbench) or working in the workbench that is running the plug-in under development (run-time workbench). For example, if you write to `System.out` from inside a plug-in under development, the text appears in the workbench editing the plug-in, not the workbench in which the plug-in is running.

Now we have a plug-in, but it doesn't do anything. We will be able to see it when we start a run-time workbench (see Figure 3.9).

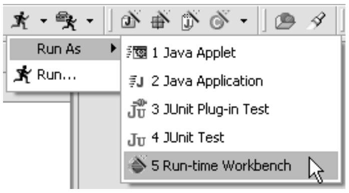
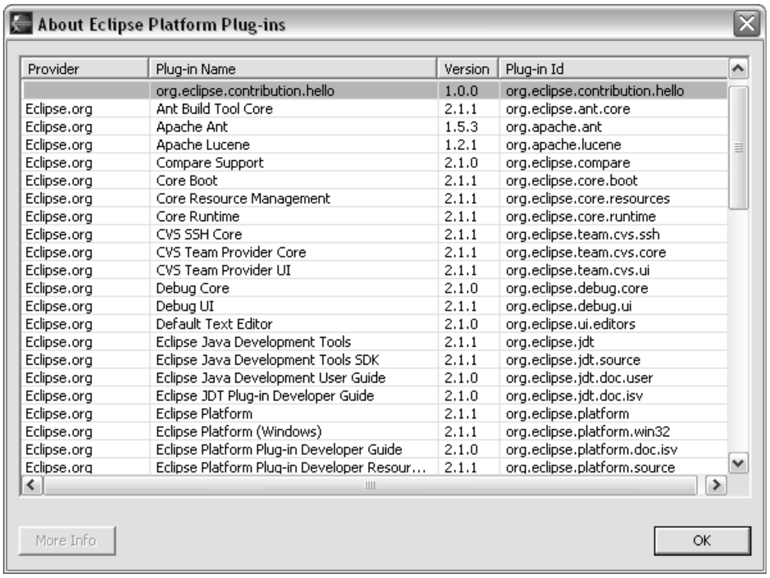


Figure 3.9

Clicking **Run As > Run-time Workbench** will bring up another instance of Eclipse, but this one (the run-time workbench) has our new plug-in loaded. You can verify that our plug-in is present by choosing **Help > About Eclipse Platform > Plug-in Details** in the run-time workbench. Our plug-in is at the top of the list, as shown in Figure 3.10.



Provider	Plug-in Name	Version	Plug-in Id
	org.eclipse.contribution.hello	1.0.0	org.eclipse.contribution.hello
Eclipse.org	Ant Build Tool Core	2.1.1	org.eclipse.ant.core
Eclipse.org	Apache Ant	1.5.3	org.apache.ant
Eclipse.org	Apache Lucene	1.2.1	org.apache.lucene
Eclipse.org	Compare Support	2.1.0	org.eclipse.compare
Eclipse.org	Core Boot	2.1.1	org.eclipse.core.boot
Eclipse.org	Core Resource Management	2.1.1	org.eclipse.core.resources
Eclipse.org	Core Runtime	2.1.1	org.eclipse.core.runtime
Eclipse.org	CVS SSH Core	2.1.1	org.eclipse.team.cvs.ssh
Eclipse.org	CVS Team Provider Core	2.1.1	org.eclipse.team.cvs.core
Eclipse.org	CVS Team Provider UI	2.1.1	org.eclipse.team.cvs.ui
Eclipse.org	Debug Core	2.1.0	org.eclipse.debug.core
Eclipse.org	Debug UI	2.1.1	org.eclipse.debug.ui
Eclipse.org	Default Text Editor	2.1.0	org.eclipse.ui.editors
Eclipse.org	Eclipse Java Development Tools	2.1.1	org.eclipse.jdt
Eclipse.org	Eclipse Java Development Tools SDK	2.1.1	org.eclipse.jdt.source
Eclipse.org	Eclipse Java Development User Guide	2.1.0	org.eclipse.jdt.doc.user
Eclipse.org	Eclipse JDT Plug-in Developer Guide	2.1.0	org.eclipse.jdt.doc.isv
Eclipse.org	Eclipse Platform	2.1.1	org.eclipse.platform
Eclipse.org	Eclipse Platform (Windows)	2.1.1	org.eclipse.platform.win32
Eclipse.org	Eclipse Platform Plug-in Developer Guide	2.1.0	org.eclipse.platform.doc.isv
Eclipse.org	Eclipse Platform Plug-in Developer Resour...	2.1.1	org.eclipse.platform.source

Figure 3.10

Next we need to contribute a button. Here's how we specify our button's appearance:

org.eclipse.contribution.hello/plugin.xml

```
<extension point="org.eclipse.ui.actionSets">
  <actionSet
    id="org.eclipse.contribution.hello.actionSet"
    label="Hello Action Set">
    <action
      id="org.eclipse.contribution.hello.HelloAction"
      label="Hello">
    </action>
  </actionSet>
</extension>
```

Each button is supported by an `Action`, the object that will be invoked when the button is pressed. The buttons in the toolbar are grouped into *action sets*, sets of related actions, as shown in Figure 3.11. For example, the buttons that create Java elements are an action set. The above declaration states that we are contributing a new action set, `point="org.eclipse.ui.actionSets"`, which contains a single action, labelled "Hello". Note once again that the IDs are globally unique but the names of elements are intended for human consumption.

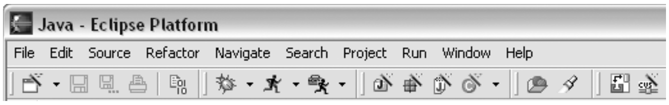
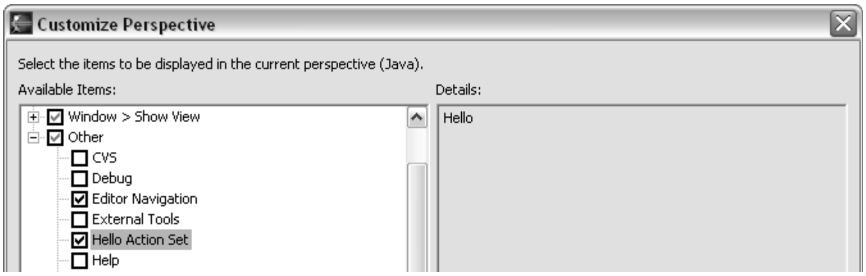

Java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help

Figure 3.11

When we start a new run-time workbench (clicking the **Running Guy** will rerun what was run previously), we can see our action set in **Window > Customize Perspective... > Other** as shown in Figure 3.12.



Select the items to be displayed in the current perspective (Java).

Available Items:
- [✓] Window > Show View
- [✓] Other
  - [ ] CVS
  - [ ] Debug
  - [✓] Editor Navigation
  - [ ] External Tools
  - [✓] Hello Action Set
  - [ ] Help

Details:
Hello

Figure 3.12

Notice that the values of the `label` elements in the declaration are used to present our contribution to the user.

Selecting our action set and clicking **OK** doesn't cause a button to appear. Why? To appear as a button, each action has to be associated with a toolbar path, a hint to Eclipse as to where to put the action. In our case, we don't want to put the action near any other particular actions, so we can make up a toolbar path:

org.eclipse.contribution.hello/plugin.xml

```
<action
  id="org.eclipse.contribution.hello.HelloAction"
  label="Hello"
  toolbarPath="helloGroup">
</action>
```

If there were already actions with the toolbar path `helloGroup`, our Hello action would appear nearby. Since this action is the only one with this toolbar path, our button appears in a group by itself. Because we didn't specify an icon, the button appears as the default red square, as shown in Figure 3.13.

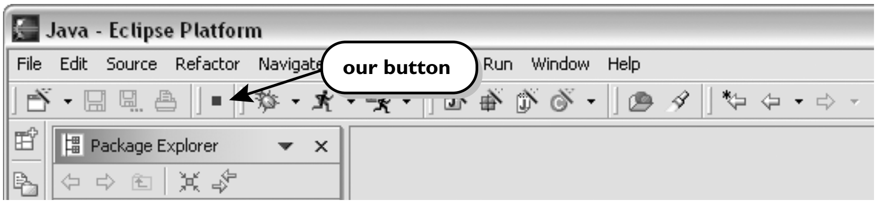


Figure 3.13 Our Button Appears as a Red Square

The button placement behavior leads us to the Sharing Rule:

SHARING RULE Add, don't replace.

When you contribute to Eclipse, your contributions will be added to the contributions already in place. There isn't a way to replace existing functionality. It's your job to find a way to think of your contribution as an addition to the existing functionality and it's Eclipse's job to harmoniously combine the contributions.

Before we implement the functionality behind the button, notice that we have been able to present our contribution to the user purely declaratively. The manifest defines how the contribution appears; the Java code defines how it behaves.

3.3 Saying "Hello"

We have finished the user-visible appearance of our plug-in. Now it is time to fill in the implementation side, actually opening a dialog containing the string "Hello". According to the Lazy Loading Rule, contributions are only loaded when they are first invoked. Eclipse waits until the button is clicked, then looks for code to invoke. The code is represented as a Java class, so the name of the class has to be part of the definition of the action.

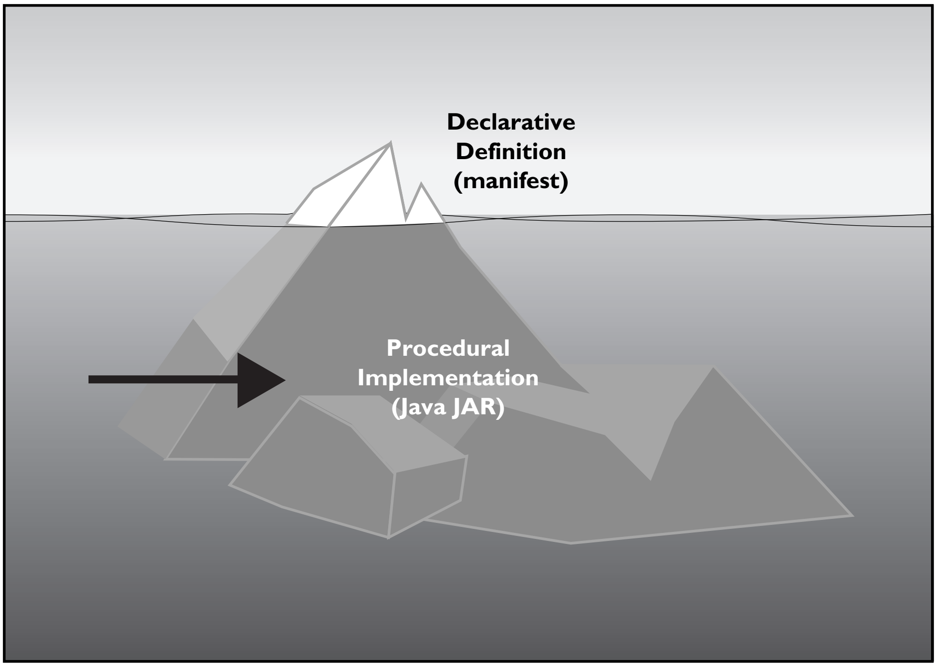


Figure 3.14 The Appearance Is Finished, Now the Implementation

Each action has the name of a Java class associated with it. When the action is invoked, an instance of that class is created to process the button click. The Java class is represented as an element of the action declaration:

org.eclipse.contribution.hello/plugin.xml
```
<action
  id="org.eclipse.contribution.hello.HelloAction"
  label="Hello"
  toolbarPath="helloGroup"
  class="org.eclipse.contribution.hello.HelloAction">
</action>
```

When we start the run-time workbench and click our button, the console in the host workbench (the one where we are developing the plug-in) tells us that the class can't be found:

```
Could not create action delegate for id: org.eclipse.contribution.
hello.HelloAction
Reason:
Plug-in org.eclipse.contribution.hello was unable to load class org.
eclipse.contribution.hello.HelloAction.
```

To make the action work, we need to create a class called `org.eclipse.contribution.hello.HelloAction`. How will the action be invoked? Eclipse needs a protocol common to all actions. This protocol is defined in the interface `IWorkbenchWindowActionDelegate`.[1] As an extender you are required to conform to this interface, the Conformance Rule (see Figure 3.15):

CONFORMANCE RULE Contributions must conform to expected interfaces.

Before we can define an implementor of `IWorkbenchWindowActionDelegate`, we have to help Eclipse find it from within our plug-in. Eclipse doesn't use the usual classpath mechanism of Java to find classes. Instead, each plug-in has its own class lookup path. This "classpath" is defined by a plug-in declaring which other plug-ins it depends on. At runtime, these prerequisite plug-ins will be searched whenever a class needs to be found. This mechanism is more efficient than Java's classpath, and more predictable, since each plug-in can precisely specify the context in which it is intended to be run.

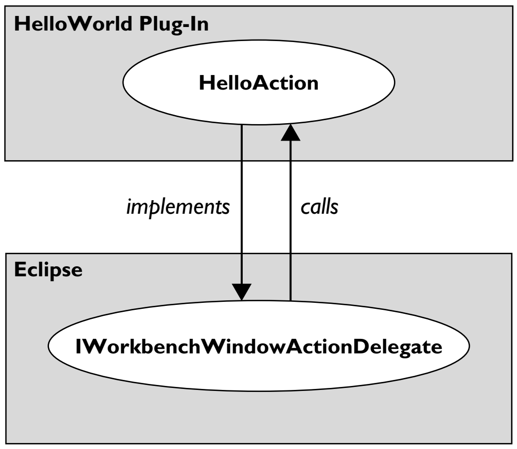


Figure 3.15 Eclipse Calls Our Contribution Through the Expected Interface

1. The convention in Eclipse is to name interfaces beginning with an "I."

The interface we want to implement, `IWorkbenchWindowActionDelegate`, is defined in the `org.eclipse.ui` plug-in, so our plug-in needs to depend on it. We add the following to our manifest:

org.eclipse.contribution.hello/plugin.xml
```
<requires>
  <import plugin="org.eclipse.ui"/>
</requires>
```

At this point, we also have to update the build class path. Go to the **Dependency** tab of the manifest editor. Select **Compute Build Path** from the context menu. PDE will now look in `org.eclipse.ui` for classes referenced by our code.

Now we can define our `HelloAction`.

1. Create a package `org.eclipse.contribution.hello` in the source folder `src` of our project.
2. Create the class `HelloAction` implementing the interface `IWorkbench-WindowActionDelegate`.

Eclipse fills in default implementations for the four methods in the signature of `IWorkbenchWindowActionDelegate`:

org.eclipse.contribution.hello/HelloAction
```
public void init(IWorkbenchWindow window) {
}
public void selectionChanged(IAction action, ISelection selection) {
}
public void dispose() {
}
public void run(IAction action) {
}
```

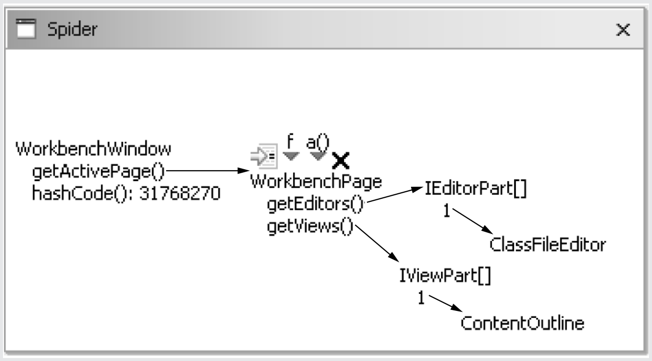
Extension implementations usually have a zero-argument constructor and are declared public because the extension objects will be created by reflection. In this case, we are lucky. Because we inherit from `Object` and define no other constructor, the default constructor is generated automatically. In most cases, we will have to define a zero-argument constructor explicitly.

When we bring up the run-time workbench and click the little red square nothing happens. We want to bring up a dialog with a cheery greeting. We replace the implementation of `HelloAction.run()` with the following:

org.eclipse.contribution.hello/HelloAction
```
public void run(IAction action) {
  MessageDialog.openInformation(null, null,
    "Hello, Eclipse world");
}
```

Spider

In the book we make use of the Spider to draw diagrams of live objects. We contributed Spider while we wrote this book to help us understand and illustrate how Eclipse works. For example, here are some objects behind a `WorkbenchWindow` captured with the Spider:

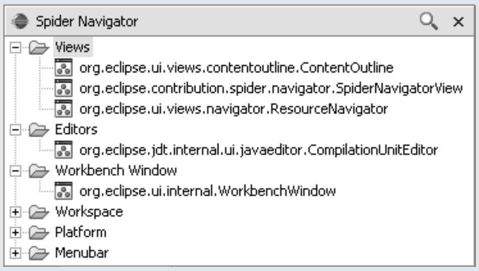


In the Spider diagram we see that a `WorkbenchWindow` has an active `WorkbenchPage` which has a collection of `IEditorPart`s and `IViewPart`s.

When you click an object in Spider, a set of handles pops-up around the object. In the above example you see four handles attached to the `Work-benchPage`. Clicking a handle allows you to invoke an action on the selected object. From left to right the following handles are shown:

- Goto source—locates the source of the object's class and opens it in an editor.
- Expand a field—shows a pop-up menu with all the fields of the object. Selecting a field adds its value to the drawing.
- Expand an attribute—shows a pop-up menu with all no-argument methods in the object. Selecting a method invokes it and adds the returned value to the drawing.
- Delete—deletes the object from the drawing

To find a starting point for exploration the Spider provides a Spider Navigator. It shows a set of top-level Eclipse objects that serve as entry points for exploration.


Spider Navigator
- Views
  - org.eclipse.ui.views.contentoutline.ContentOutline
  - org.eclipse.contribution.spider.navigator.SpiderNavigatorView
  - org.eclipse.ui.views.navigator.ResourceNavigator
- Editors
  - org.eclipse.jdt.internal.ui.javaeditor.CompilationUnitEditor
- Workbench Window
  - org.eclipse.ui.internal.WorkbenchWindow
- Workspace
- Platform
- Menubar

Double clicking an object in the Spider Navigator adds it to the Spider drawing.

You can download the Spider from *www.javaspider.org*. Once you have the Spider plug-in installed you can start to explore Eclipse by opening the Spider perspective (**Window > Open Perspective > Spider**). The Spider perspective shows both the Spider Navigator and the Spider drawing view.

Now when we start the run-time workbench and click the button, we see the message we've been expecting, shown in Figure 3.16.

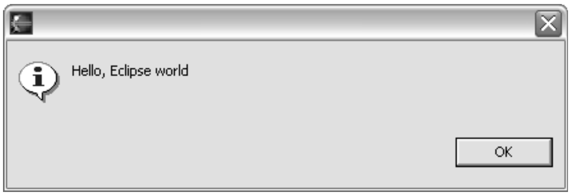

Hello, Eclipse world

Figure 3.16

Before leaving our delightful little example, let's see how the Lazy Loading Rule plays out. Here are the objects of the toolbar item after the `HelloAction` has been made visible, but before the button has been clicked. The `WWinPluginAction` (for "Workbench Window") is the Proxy for our action (see Figure 3.17).

When we click the button, the proxy creates the delegate and forwards the request to the delegate, which causes the dialog to appear. Afterwards, our action delegate has been loaded, as shown in Figure 3.18.

Following the Lazy Loading Rule, our `HelloAction` class is not loaded until the first time it is invoked. Then the class is loaded, the instance is created, and the instance is invoked.

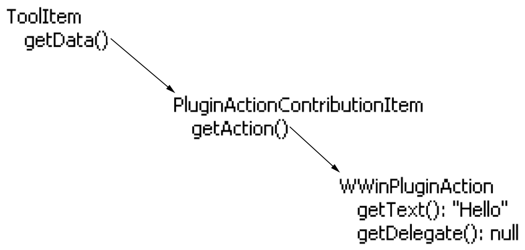


Figure 3.17 The Proxy Action

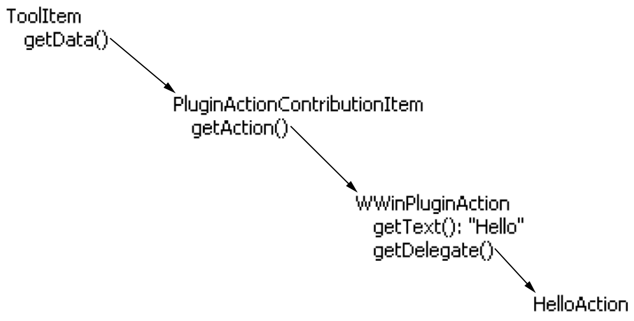


Figure 3.18 After Clicking the Button, the Real Action Has Been Loaded

With a dozen lines of Java and a dozen lines of specification in XML we were able to contribute to Eclipse.

The complete contents of the two files are shown in Section 3.3.1 and Section 3.3.2.

3.3.1 HelloAction.java

```
package org.eclipse.contribution.hello;

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

public class HelloAction implements IWorkbenchWindowActionDelegate {

  public void dispose() {
  }

  public void init(IWorkbenchWindow window) {
  }
  public void run(IAction action) {
    MessageDialog.openInformation(null, null,
      "Hello, Eclipse world");
  }

  public void selectionChanged(IAction action,
    ISelection selection) {
  }

}
```

3.3.2 *plugin.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
  id="org.eclipse.contribution.hello"
  name="Hello World"
  version="1.0.0">

  <runtime>
    <library name="hello.jar"/>
  </runtime>
  <requires>
    <import plugin="org.eclipse.ui"/>
  </requires>

  <extension
    point="org.eclipse.ui.actionSets">
    <actionSet
      label="Hello Action Set"
      id="org.eclipse.contribution.hello.actionSet">
      <action
        label="Hello"
        class="org.eclipse.contribution.hello.HelloAction"
        toolbarPath="helloGroup"
        id="org.eclipse.contribution.hello.HelloAction">
      </action>
    </actionSet>
  </extension>
</plugin>
```

3.4 Forward Pointers

- Plug-ins don't need to contain code. Documentation plug-ins are written as a combination of XML and HTML.