Contributing to Eclipse

Kent Beck - Erich Gamma

81 pages
Draft 11/15/2002

Table of Contents

To Do

Point folks at the examples in available on www.eclipse.org
SWT snippetsReplace bold code with “changed” ctag
Replace keyword formatting with a “keyword” ctag

Preface

You like writing tools for programmers, sometimes for yourself, sometimes for your buddies. Eclipse seems like an interesting medium for your skills, but how do you get started? Helping you get started contributing to Eclipse is our purpose in writing this book.

Eclipse is good news/bad news for developers who like writing tools for developers. The good news is that the platform is incredibly powerful and the internal developers have followed all the rules in creating the Eclipse Java IDE. The bad news is that Eclipse has a strong world view. If you want to play nicely in Eclipse’s sandbox, you need to learn the rules.

Beginning with Eclipse feels a bit like parachuting blindfolded into Bangkok (this analogy doesn’t apply to Thai programmers.) When you land, you know you need food and shelter, but how are you going to get it? How can you map your clear desires onto the resources available?

Overcoming this feeling of dislocation is the primary goal of Contributing to Eclipse. If you parachuted into Bangkok with a guide, you could say, “I’m hungry,” and your guide would say, “Here’s the kind of place you can get a meal.” Similarly, we can listen to, “I want to build such and so,” and tell you, “This should be its own perspective, that is an object contribution, and you’ll need a new editor for that.”

When you are finished with this book, you won’t have a complete map of Eclipse, but you’ll know at least one place to get each of your basic needs met. You will also know the rules through which you can play well with others. It’s as if you draw a map of Bangkok marked with six streets, a restraunt, a flophouse, and a disco. You won’t know everything, but you’ll know enough to survive, and enough to learn more.

When you learn Eclipse, you’ll spend much more time reading code than writing code. You will have to get used to incredibly productive days in which you spend six hours reading and one hour typing. After you become familiar with Eclipse “culture”, you’ll “just know” how to solve more and more problems. However, you’ll always solve problems by copying the structure of solutions to similar problems, whether by mimicking Eclipse structure, or the structure of your own previous efforts.

As we walk together through our example, we won’t pretend that we perfectly remember all the details. Instead, we’ll show you how we found structure to mimic. Learning to effectively use Eclipse’s search facilities is part of becoming a learner.

When we laid out Contributing to Eclipse, we had a daunting stack of concepts to cover. If we tried to tell you about all 2000 cool ideas in Eclipse, though, we would have a book that would cut off the circulation to your lower extremities. In deference to your feet, we’ve chose the 50 things that are most important for getting you started. Every chapter concludes with “Forward Pointers”, places in the code where you can explore the extension of the concepts covered in the chapter. When we teach Eclipse, this is exactly the structure we use—“Why don’t you look at the org.eclipse.core.runtime manifest?”

Once there was a doctoral student who had to take a qualifying exam about fruit. He only had enough time, though, to learn about cucumbers. When the time came for the exam, the first question was, “Tell us about the tomato.” “You see,” he said, “the tomato, like the cucumber, is actually a fruit. The cucumber is 80% water, has a disease resistant skin, and it used in salads.” Every question that came up, he answered with cucumber facts.

We know cucumbers. Except in our case, there are two cucumbers: patterns and JUnit. You’ll find pattern-y advice throughout the book. In fact, we were uncomfortable with writing until we started writing the Rules. Once we had the concept of Rules, we could proceed happily. The Rules are really patterns in a micro format.

JUnit, our second cucumber, is the basis for the running example. It feels a bit strange to have run our public technical lives based on 8 hours in an airplane over the Atlantic, but there you have it. JUnit is fertile ground for an Eclipse example, because the core is simple—running tests—but the implications, the number of ways to present results, the number of ways to present the tests themselves, are legion.

We did want to warn you, though, that if you’re sick of patterns and JUnit, this is probably not the book for you.

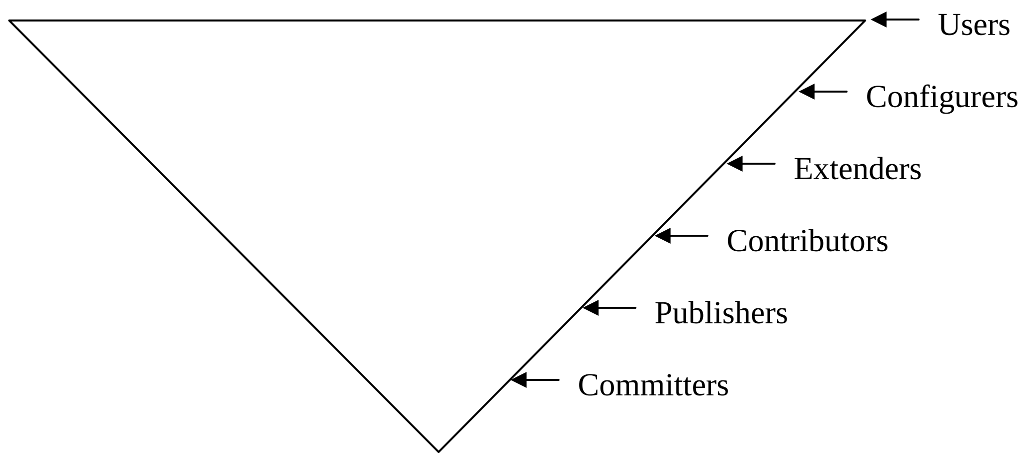
The Big Picture

The usual relationship between programmer and 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 70s, Smalltalk was written with quite 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.

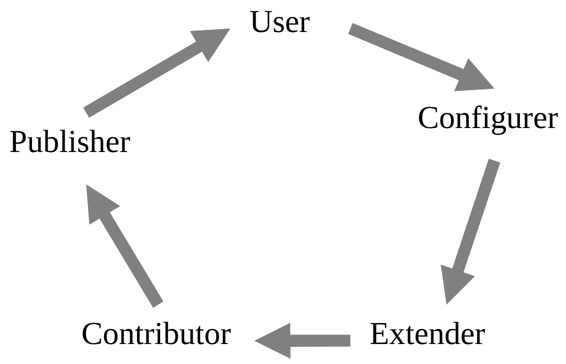
User-as-programmer may seem far fetched, but Smalltalk saw many cases where non-programming professionals wrote applications uniquely suited to their own needs by learning programming a little at a time and by example. The key to enabling user programming is to structure the environment to encourage learning, and to give 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. It's 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 down the pyramid of users requires investment, and will be attempted by fewer people.



- Users—Folks who use Eclipse for their daily work. Currently this is restricted to programmers, but there is no reason in principle why Eclipse couldn't be used to structure other computing work.
- Configurers—Folks who customize their experience of Eclipse, whether by re-arranging perspectives, setting preferences, or deciding which views to show. Configuration is limited to changes envisioned by the original programmer.
- Extenders—Folks who use programming to make changes not envisioned by the original programmers. Eclipse provides a rich set of places to "plug in" new functionality, since Eclipse itself is built entirely by plugging in functionality.
- Contributors—Once you've written something cool, other folks will want it to. Eclipse is consciously structured so you can easily bundle together extensions so others can load them.
- Publishers—Eclipse is built out of "places-to-plug-functionality-in" (extension points) and "functionality-to-be-plugged-in" (extensions.) Once you have written something cool, the next step is to make it available for others to extend 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 build trust in 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:



The Contribution Circle

What makes this circle interesting is the final arrow, from Publisher 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 a publisher of 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 find useful, but you didn't have to do any more work.

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

Book Goals

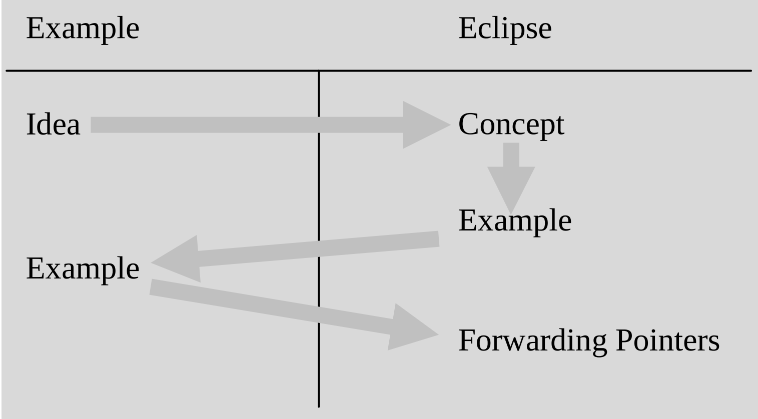
We’ve been talking about writing a book together for nearly a decade. We share an interest in software design, in the design of software for widespread adaptation (frameworks), and in discovering the “universal” rules that lie behind design. We first met at Bruce Anderson’s Architecture Handbook workshop at OOPSLA-?? In ??. There we began discussing software design and 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.

Cutting a long story short (The Great Taligent Debacle, Design Patterns, JUnit, Extreme Programming), IBM asked us to write a book about contributing to Eclipse. Eclipse is a fabulous playground for geeks, but if geeks don’t know about it, they won’t come play. The stars aligned. We had an excuse to write our book about design, but with enough context that we wouldn’t go spiraling off into useless abstract philosophy.

We have one goal for you as the reader of this book—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 publisher, 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 example 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 about Eclipse. When we go program in Eclipse, we search for examples from which to copy. As we develop our example, 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, and behind effect platforms of all kinds (all in our humble opinions, naturally.)

Sidebar: Pedagogical structure



Chapter 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 we want to add, describe the general concept in Eclipse which fits the idea best, show an example in Eclipse itself 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 deeper through the Contribution Circle. The next chapter, Hello World, takes you as quickly as possible from idea to implementation. The second circle, chapters N-M, develop the simplest possible form of a plug-in. The third circle, chapters N-M, take the basic plug-in and prepare it for distribution, including publishing extension points of its own. The final circle, chapters N-M, provide a bit of a cook’s tour of Eclipse, highlighting areas of the API that are worth exploring early in your career as a contributor.

Plug-In

Since this and the following chapter constitute a complete circle, we should have a little bit about the architecture of Eclipse. For now, it is enough for you to think of Eclipse as a collection of places-to-plug-things-in (extension points) and things-plugged-in (extensions). The powerstrip above 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.

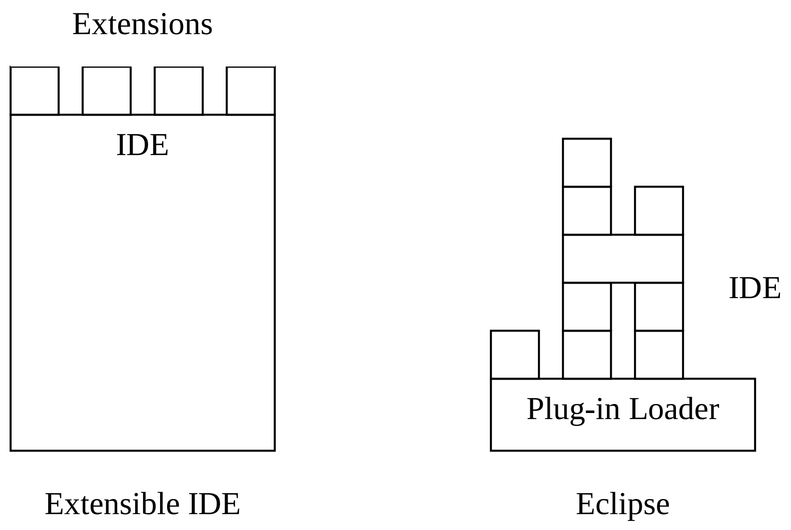
Our Hello World example will consist of an extension plugged into the extension point for buttons on the button bar. When we press the button, a dialog will appear announcing proudly, “Hello World.”

Hello, World

Before we can make our first contribution to Eclipse, we have to talk the least little bit about the architecture of the system. 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 many things are
contributed.



As a consequence of making everything a contribution you will have lots of contributions. Assume that you have a system built out of thousands of tiny 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 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 impacts on the architecture. So here. If you have thousands of contributions, you only have a millisecond to process each contribution.

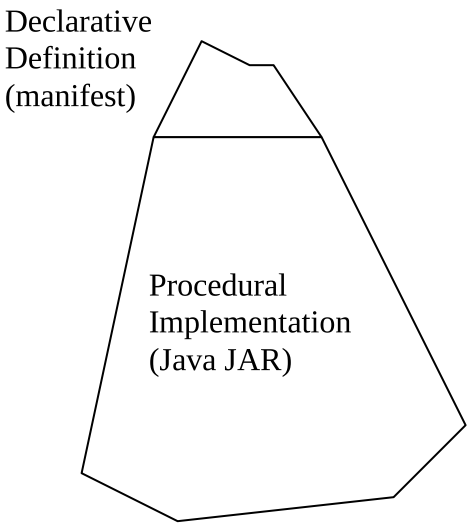
Now we come to a dilemma. The logic contained within a contribution can be substantial. The compiled form of this logic, Java class files (collected in a JAR file), can easily take seconds to load. Making Java classes that are guaranteed to load quickly is more work than most people want to go to. If we want to guarantee snappy startup, 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?

Declaration/Implementation Split

If you know only which contributions are present, 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 XML-based manifest file. The implementation is in Java.


Declarative Definition (manifest)
Procedural Implementation (Java JAR)

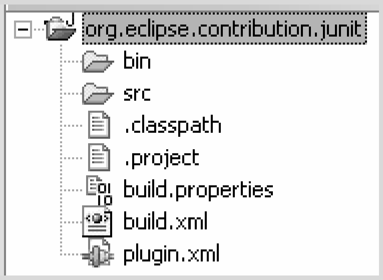
The manifest is the user-visible tip of the iceberg

Sidebar: The Basic Plug-in Structure

Definition: A plug-in is a piece of behavior that is outside the platform. A plug-in is represented as a directory containing:

- A manifest
- Resources, like icons (optional)
- Java code, in a JAR

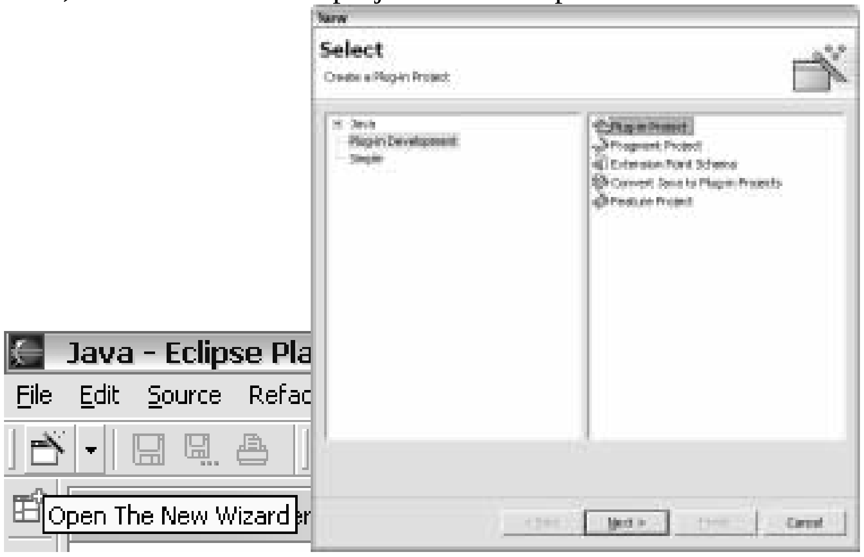
Here is the directory structure for our simple example:



- bin—A directory containing the compiled class files and resources. Do not change the name of this folder if you want to launch your plug-in from Eclipse (i.e. while you're developing the plug-in.)
- src—A source folder containing the Java packages. Complex projects often have more than one source folder.
- .classpath—The project's build classpath.
- .project—Eclipse's project description file.
- build.properties—Information for deploying the plug-in as a JAR.
- build.xml—Generated from build.properties when you execute "Create Plug-in JARs" on plugin.xml. You'll execute this when you want to use the plug-in yourself or give it to someone else.
- plugin.xml—The manifest, a description of the user-visible part of the plug-in.

Hello Button

First, we have to create a project. The best place to start is the New wizard.



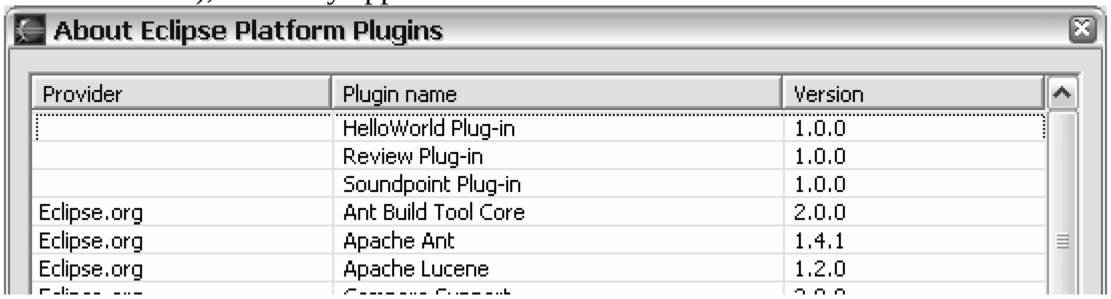
Give the project the name “org.eclipse.contribution.hello” (this is also the name of the plug-in). The next page is fine as is, press Next. On the following page, select “Default Plug-In Structure” before pressing Next. On the final page, press Finish.
Fix this by stepping through the example one more time.

To contribute our greetings to Eclipse, we will start by declaring in the manifest what our contribution looks like. We begin with the “address” of our plug-in, consisting of a name and version number.

```
<plugin
  id="org.eclipse.cobntriution.hello"
  name="HelloWorld Plug-in"
  version="1.0.0">
</plugin>
```

The name is used when presenting information about the plug-in to the user, for example in the about dialog. Internally, the system only works with the id (plug-in ids should follow the Java package naming convention to avoid names clashes.) The version is mandatory because plug-ins can rely on each other (remember that nearly everything is a plug-in) and it important that you can refer to a particular version you depend on.

Now we have a plug-in, but it doesn’t do anything. However, when we install it (more about that later), it already appears.


About Eclipse Platform Plugins

Provider	Plugin name	Version
	HelloWorld Plug-in	1.0.0
	Review Plug-in	1.0.0
	Soundpoint Plug-in	1.0.0
Eclipse.org	Ant Build Tool Core	2.0.0
Eclipse.org	Apache Ant	1.4.1
Eclipse.org	Apache Lucene	1.2.0

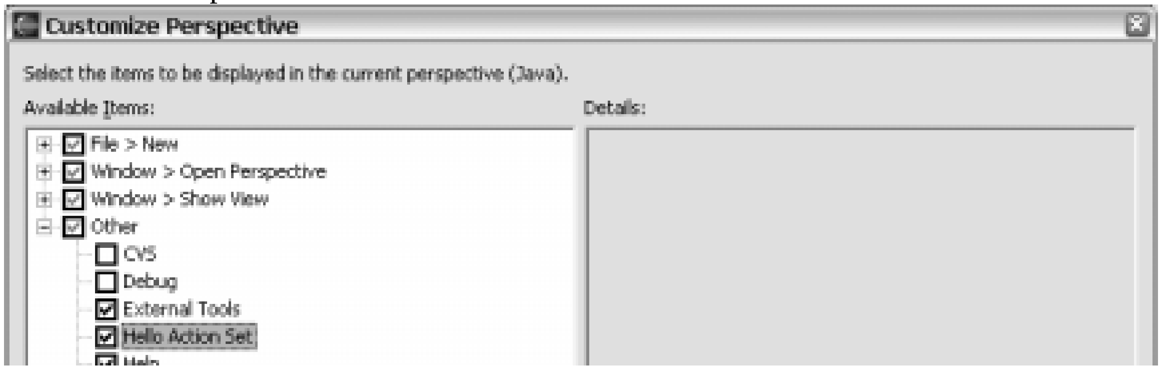
Let’s say that we want our plug-in to be invoked as a button in the toolbar. We insert the following into the manifest:

```
<extension
    point="org.eclipse.ui.actionSets">
  <actionSet
      label="Hello Action Set"
      id="helloWorld.actionSet">
  </actionSet>
</extension>
```

Before we can contribute, we have to know to what we are contributing. That’s the purpose of “<extension” (we are extending Eclipse) and “point=org.eclipse.ui.actionSets” says that we are adding to the sets of actions presented to the user. One set of actions, for example, is for creating Java elements (create class, create package). We’re adding another set of our own.

Actions are grouped into sets so the user can configure which actions are visible for a task. Grouping actions into sets allows the user to show or hide groups of related actions with a single gesture.

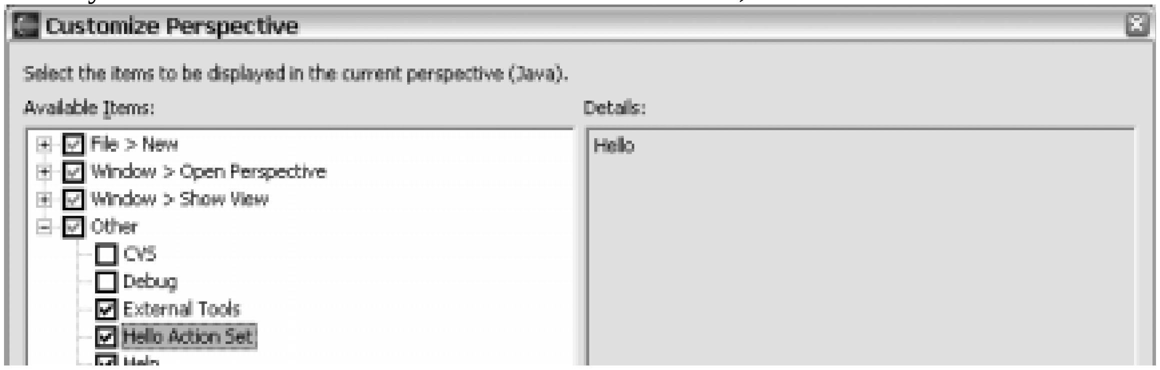
So far we have said we are contributing a new set of actions (in our case, only one). The next bit of the manifest tells Eclipse how the user will see the set of actions (“label=”Hello Action Set””) and how the system will refer to the set of actions internally (“id=”helloWorld.actionSet””). Now we can tell Eclipse to show our action set (even though nothing will happen because there aren’t any actions in it yet.) Selectign Window->Customer Perspective… shows:



We almost have our button on the screen. Next we need to tell the manifest that inside our action set is an action. We insert the following as a child of the action set element.

```
<action
    id="helloWorld.actions.HelloAction"
    label="Hello"
</action>
```

Again we see the label used to communicate to the user and the id used to uniquely identify a contribution. Now when we view the action set, we see an action inside of it:



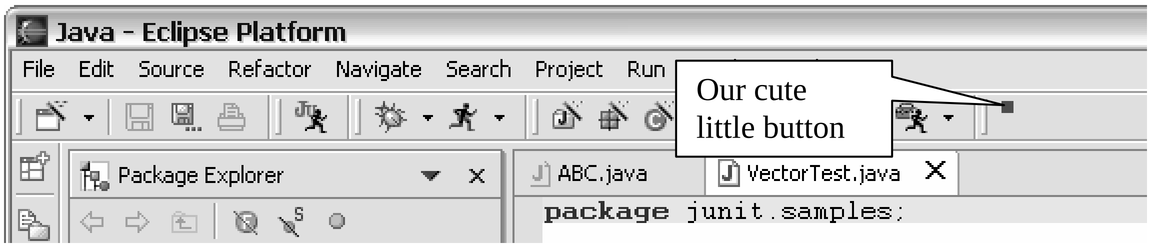
Actions are used to represent menu items and tools in the tool bar. An action element defines the presentation of the action (icon, menu item name, etc). So, before we can see our action as a button, we have to tell Eclipse that the action belongs in the toolbar and where in the toolbar it should go. We extend the action element with an attribute “toolbarPath”.

```
<action
    id="helloWorld.actions.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. This 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. You will have to find a way to think of your contribution as an addition, and it’s Eclipse’s job to harmoniously combine the contributions.

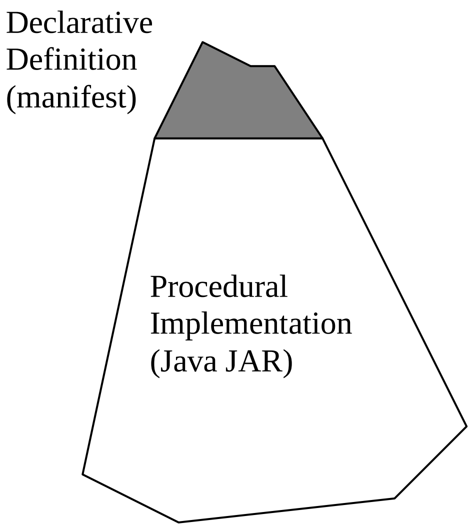


```
package junit.samples;
```

Since we haven’t defined an icon, the button appears as a red dot. This is a good enough presentation for now. You can find all the gory details for making your plug-ins pretty in the online Plug-in Development Environment help or <Jumpstart book>. We’re impatient. It’s been pages and pages, and we still haven’t said hello. How rude.

Before we continue, though, notice that by just specifying the declarative part of our plug-in, the contribution is already visibly part of Eclipse. As you learn to contribute, you will have to get used to this split—when to express an idea in the manifest and when to express it as code.

Saying “Hello”


Declarative Definition (manifest)
Procedural Implementation (Java JAR)

The manifest done, we work on the implementation

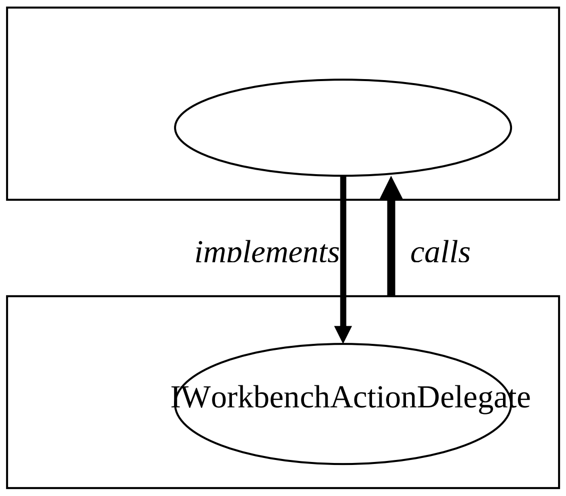
We have finished the user-visible appearance of our plug-in. Now it is time to fill in the implementation side, actually opening a dialog containing the string "Hello". According to Rule #2, contributions are only loaded when they are first invoked. Eclipse waits until the button is pressed, then looks for code to invoke. The code is represented as a Java class, so the name of the class has to be part of the action definition.

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

You can try pressing the button now, even before you define the class. Nothing horrible will happen. Eclipse will just tell you that the operation isn't available since it couldn't find the specified class. Eclipse is looking at the manifest when the button is pressed and saying, "I need to make one of these HelloActions and invoke it."

To make the operation available, 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 IWorkbenchActionDelegate. You as a contributor are required to conform to this interface. This leads to the Conformance Rule:

Conformance Rule: Contributions must conform to expected interfaces



The action must implemented the expected interface

In our case Eclipse expects the action's class to implement these four methods defined by IWorkbenchWindowActionDelegate:

- init(IWorkbenchWindow)—Ignore for now.
- dispose()—Ignore for now.
- selectionChanged(IAction, ISelection)—Ignore for now.
- run(IAction)—Called once each time the button is pressed.

The boring part of the action looks like this:

```
public class HelloAction implements IWorkbenchWindowActionDelegate {
        public void init(IWorkbenchWindow window) {
        }
        public void selectionChanged(IAction action, ISelection selection) {
        }
        public void dispose() {
        }
}
```

We just stub out the methods we are too impatient to understand at the moment. The most important method is run(). When run() is called, we will pop-up our dialog. Here is the code to do that:

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

MessageDialog is a utility for opening some standard dialogs. The third argument is the string to be displayed.

Before we can invoke the class, Eclipse has to have a place to search for the compiled code. The executable part of the plug-in is packaged as a JAR file. The manifest has to say which JAR file to search, like this:

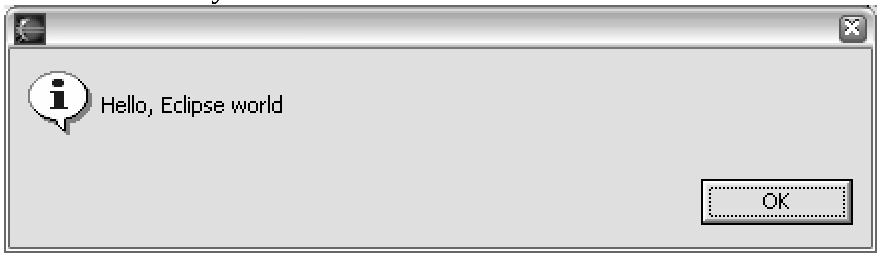
```
<runtime>
  <library name="helloworld.jar"/>
</runtime>
```

The JAR file name is relative to the directory that holds the plug-in.

Finally, our Hello action refers to IWorkbenchWindowActionDelegate and other platform classes. How can Eclipse know where to find these classes? The plug-in must be self contained. Plug-ins can't reliably rely on global resources like the system's class path. We make the dependency of this plug-in on other plug-ins explicit, like this:

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

Org.eclipse.ui is the plug-in containing IWorkbenchWindowActionDelegate and the other action classes. This statement tells Eclipse to look in org.eclipse.ui for classes it doesn't find in plugin's library helloworld.jar.
Now we're ready to run. Et voila!


Hello, Eclipse world

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

HelloAction.java

```java
package helloWorld.actions;

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

public class HelloAction implements IWorkbenchWindowActionDelegate {
	public void selectionChanged(IAction action, ISelection selection) {
	}
	public void dispose() {
	}
	public void init(IWorkbenchWindow window) {
	}
	public void run(IAction action) {
		MessageDialog.openInformation(null, null, "Hello, Eclipse world");
	}
}
```

plugin.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<plugin
  id="helloWorld"
  name="HelloWorld Plug-in"
  version="1.0.0">

  <runtime>
    <library name="helloWorld.jar"/>
  </runtime>

  <requires>
    <import plugin="org.eclipse.ui"/>
  </requires>

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

```
</plugin>
```

Forward Pointer

- In a plug-in, only the manifest is mandatory. You could have an XML-only plug-in. Help is written this way, as are resource bundles for other languages.
- You cannot add to an existing ActionSet, but you can contribute an ActionSet to the same group as another action set, which has the same effect.

Der Plan

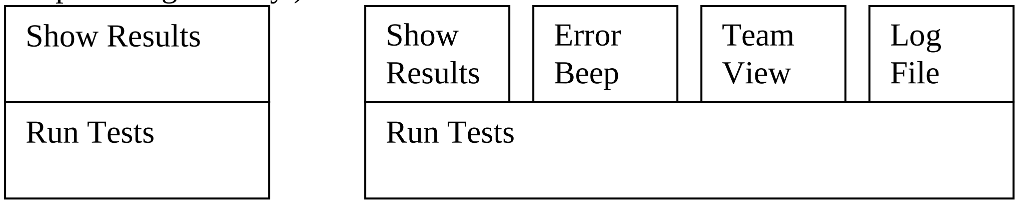
The best way to tell you about Eclipse is to walk you through the development of an example. As we explained with the cute cucumber story in the preface, all we know about any more is JUnit, so that's what we'll use as our example. What is JUnit?

JUnit is a simple framework for writing automated tests. We wrote the first version flying from Zurich to Washington D. C. for OOPSLA-97. We were having fun, making a bit of noise and getting the fish eye from a conservatively but badly dressed man across the aisle. It wasn't until we got to customs and watched him flash a badge and breeze right through that we realized he was probably a federal agent. It's not often you have a brush with death as a programmer.

Enough with the reminiscence. JUnit took off like wildfire, spreading through the Java world like a virus and soon infecting other languages. At last count there were 38 ports of the basic framework.

What makes JUnit interesting as an example is that it is a programming tool written by programmers for programmers, our prototypical Eclipse extension, and it is simple (so we won't have to explain a lot of stuff that isn't Eclipse related). While the user interfaces for JUnit have all been simple, there are lots of cool stuff we can imagine wanting to add.

At first, our Eclipse/JUnit integration will be simple. Eventually, though, we will have added little bits and pieces that add up to a sophisticated interface (and shown you Eclipse along the way.)



Basic JUnit With Cool Stuff

Circle One

This first pass through the Contribution Circle will take us…

Getting Started

What we'll see in this chapter:

- Contributing an action to an object (in this case a menu item).
- Implementing the contribution by looking at an example.

We'd like to get the minimal user interface possible for our example, just to have something complete running. How about this—

1. Select a test case class in the package explorer.
2. Pop-up a context menu.
3. Select "Run Test".
4. Run the tests.
5. Pop up a dialog with the results of running all the tests in the type.

Sidebar: Plug-in Development Environment (PDE)

Because plug-ins are so important to Eclipse, Eclipse has evolved sophisticated supported for developing 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, or working in the workbench that is running the plug-in. For example, if you write to System.out, the text appears in the workbench editing the plug-in, not the workbench in which the plug-in is running. We'll call the workbench used to develop the plug-in the host workbench and the workbench used to execute the plug-in the run-time workbench.

Create a new plug-in project named org.eclipse.contribution.junit. Use the default plug-in structure, but uncheck the button on the final page "Generate code for the class."

Sidebar: When You Need a Plug-in Class

Every plug-in is represented by an instance of a plug-in class. The plug-in class singleton has hook methods for the lifecycle of the plug-in. You can load resources when the plug-in is first loaded. You can clean up when the plug-in is terminated (e.g. when Eclipse is closed). The plug-in is also the source for shared information like preferences, images, and persistent dialog setting.

The plug-in we're defining here doesn't perform any lifecycle actions like loading images, nor does it need any shared information. We rely on the default plug-in class. If later we need a plug-in class, we will define one.

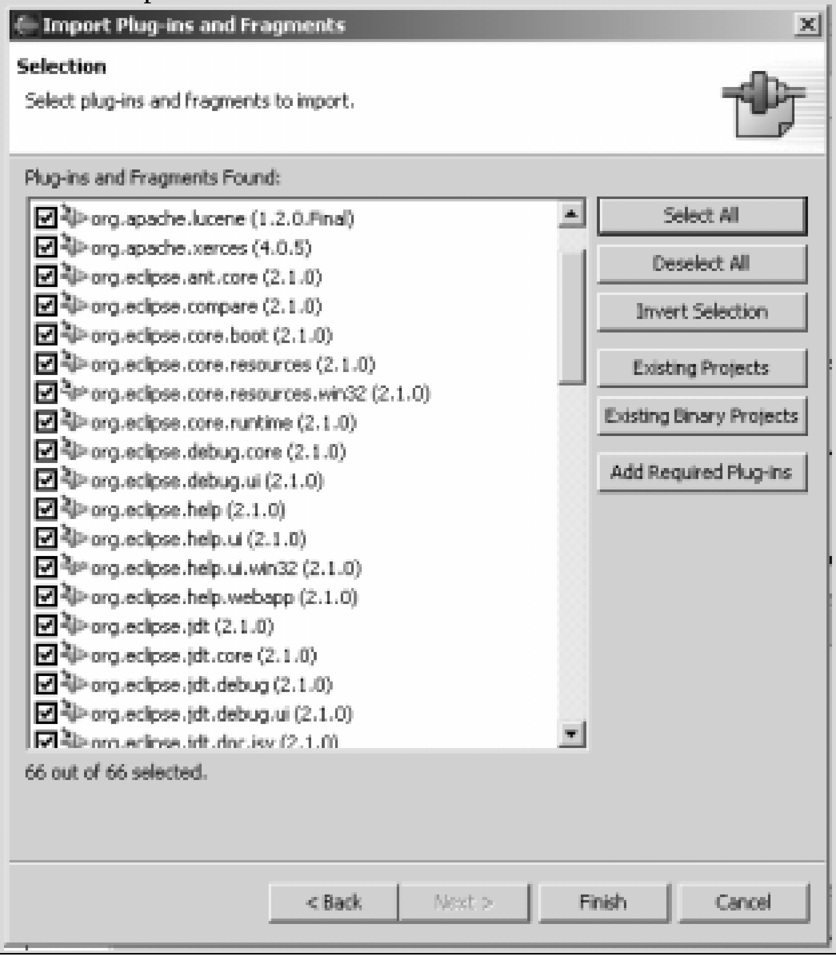
Next, we need to say how the contribution will appear to the user—as a menu item available when a type is selected. When programming in Eclipse, always begin by finding a similar application and copying its structure, the Money See/Monkey Do Rule:

> Monkey See/Monkey Do Rule: Always start by copying the structure of a similar plug-in

Sidebar: Setting Up Eclipse for Plug-in Development

Our approach to Eclipse development is based on mimicking code from the actual Eclipse source. To do so we want to have access to the entire source and we want to be able be able to quickly search for references all over the Eclipse source. One way to achieve this is to load the entire source (around 1 Million lines of code) into the workspace. Obviously managing this huge amount of source has its price. However, we will typically only read existing code and do not need the option to modify and compile it. Fortunately PDE offers a quick and space efficient way to set-up a workspace where existing plug-ins are not modifiable but fully browseable. The trick is to represent existing plug-in as *binary projects*. Binary projects cannot be modified but are fully searchable for references and declarations. They have the other nice property that your build class paths in the workspace become independent of the fact whether a project is represented in source or binary. It is therefore easily sharable within a team. Here is how to set-up a workspace with binary projects for all the plug-ins shipped with Eclipse:

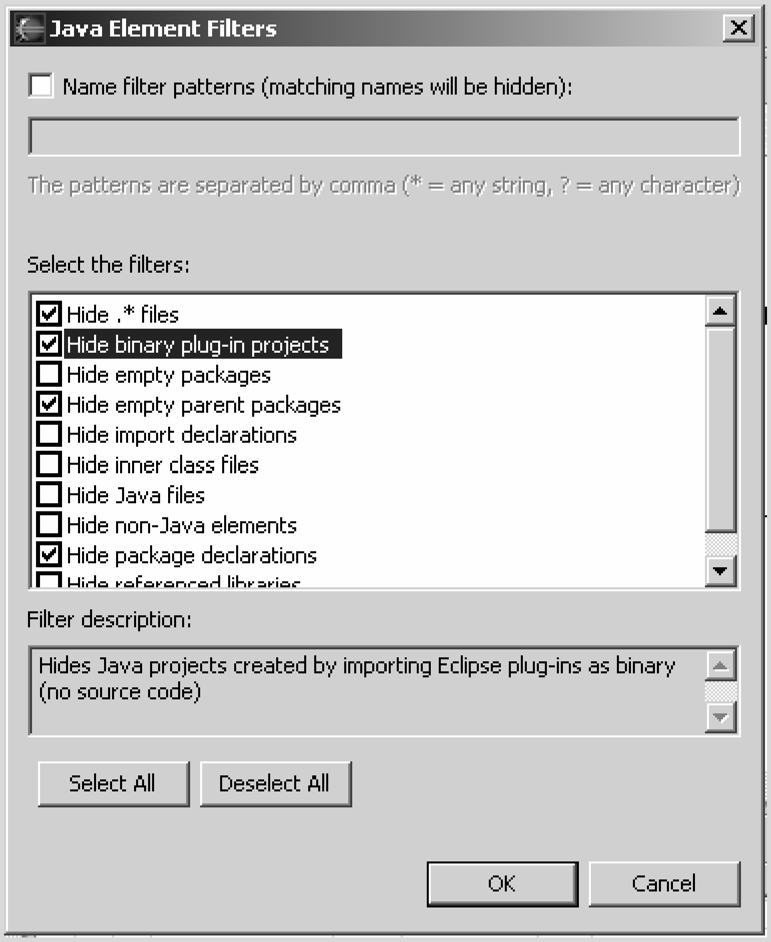
1. Start-up eclipse with an empty new workspace.
2. File>Import>External Plug-ins and Fragments, accept the defaults on the first page. On the second page press “Select All” to import all plug-ins from your host workspace.



```
☑ org.apache.lucene (1.2.0.Final)
☑ org.apache.xerces (4.0.5)
☑ org.eclipse.ant.core (2.1.0)
☑ org.eclipse.compare (2.1.0)
☑ org.eclipse.core.boot (2.1.0)
☑ org.eclipse.core.resources (2.1.0)
☑ org.eclipse.core.resources.win32 (2.1.0)
☑ org.eclipse.core.runtime (2.1.0)
☑ org.eclipse.debug.core (2.1.0)
☑ org.eclipse.debug.ui (2.1.0)
☑ org.eclipse.help (2.1.0)
☑ org.eclipse.help.ui (2.1.0)
☑ org.eclipse.help.ui.win32 (2.1.0)
☑ org.eclipse.help.webapp (2.1.0)
☑ org.eclipse.jdt (2.1.0)
☑ org.eclipse.jdt.core (2.1.0)
☑ org.eclipse.jdt.debug (2.1.0)
☑ org.eclipse.jdt.debug.ui (2.1.0)
☑ org.eclipse.jdt.doc.isv (2.1.0)
66 out of 66 selected.
```

3. Press finish, the plug-ins will be imported into binary projects, their build class path initialized.

You can now easily browse and search the full Eclipse source. A consequence of creating binary projects for all plug-ins you end-up with 60 projects in your workspace. If you do not want to see these binary projects or to reduce clutter you can easily hide binary projects the Packages Explorer by turning on a filter for binary projects. Select Filters from the Package Explorer View menu and check “Hide binary plug-in projects in the following dialog:


Java Element Filters

[ ] Name filter patterns (matching names will be hidden):

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

Select the filters:

- [✓] Hide .* files
- [✓] Hide binary plug-in projects
- [ ] Hide empty packages
- [✓] Hide empty parent packages
- [ ] Hide import declarations
- [ ] Hide inner class files
- [ ] Hide Java files
- [ ] Hide non-Java elements
- [✓] Hide package declarations
- [ ] Hide referenced libraries

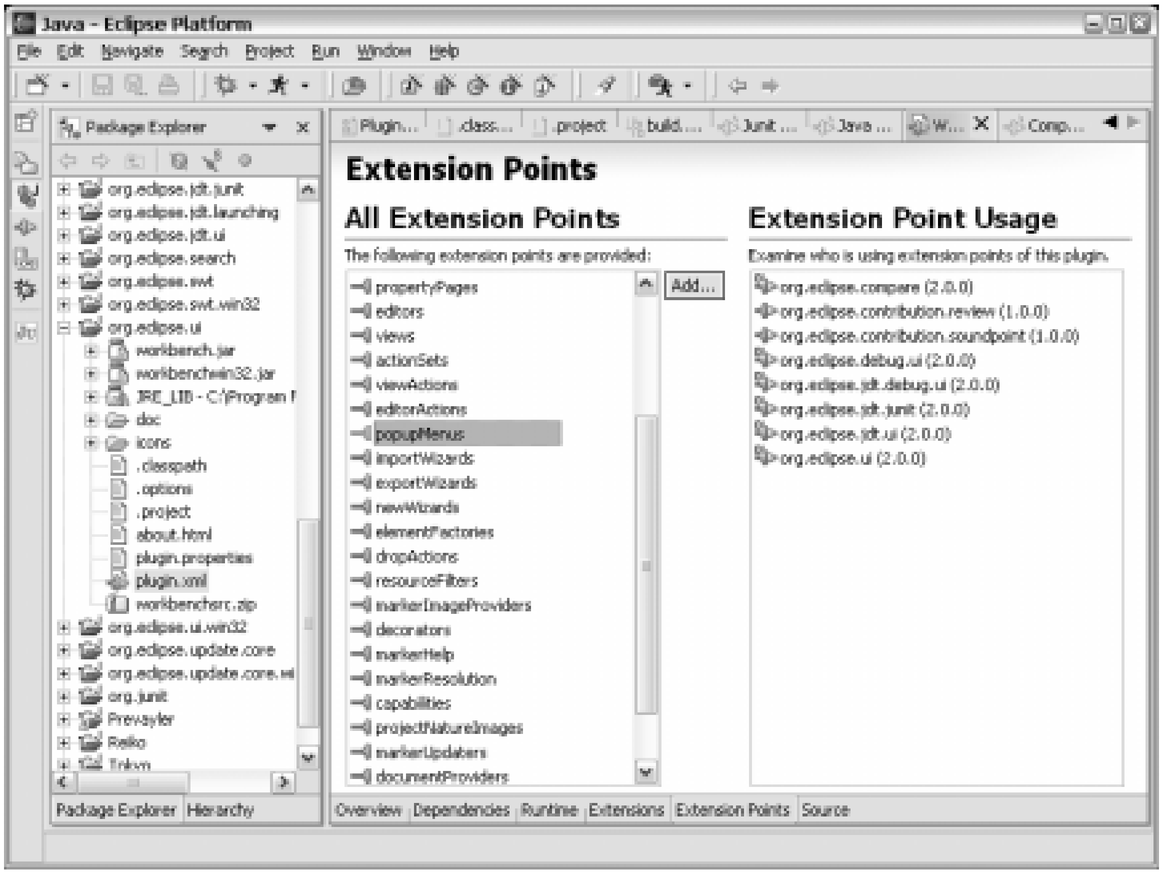
Filter description:

Hides Java projects created by importing Eclipse plug-ins as binary (no source code)

As result of turning on the filter the Packages Explorer View becomes empty since we haven’t created any source plug-in projects yet.

Notice that PDE also supports a simpler workspace setup for plug-in development. This setup doesn’t require importing existing plug-ins into the workspace. Instead the plug-in in your host workspace are used to resolve such plug-ins references. This setup is simpler and you do not have to know about binary plug-in projects. However, in this case the plug-ins are represented in the workspace and hence they are not indexed for searching. Searching for examples of API use with reference searches is therefore not possible. For this reason we recommend the setup with binary projects. *Erich will fill this in*

Object Contributions are the way you add menu items to selections. Where can we find some examples of object contributions? Org.eclipse.ui is where the extension point is defined for pop-up menus. By looking at the extension points defined in the manifest file, we can see what the extension points are and (more to our purpose here) where they are used:


Java - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Package Explorer
Plugin... .class... .project build.... Junit ... Java ... W... Comp...
Add...
Overview Dependencies Runtime Extensions Extension Points Source
Package Explorer Hierarchy

Package Explorer:

- org.eclipse.jdt.junit
- org.eclipse.jdt.launching
- org.eclipse.jdt.ui
- org.eclipse.search
- org.eclipse.swt
- org.eclipse.swt.win32
- org.eclipse.ui
  - workbench.jar
  - workbenchwin32.jar
  - JRE_LIB - C:\Program F
  - doc
  - icons
  - .classpath
  - .options
  - .project
  - about.html
  - plugin.properties
  - plugin.xml
  - workbenchsrc.zip
- org.eclipse.ui.win32
- org.eclipse.update.core
- org.eclipse.update.core.wi
- org.junit
- Prevayler
- Reiko
- Tokyo

Extension Points

All Extension Points

The following extension points are provided:

- propertyPages
- editors
- views
- actionSets
- viewActions
- editorActions
- popupMenus
- importWizards
- exportWizards
- newWizards
- elementFactories
- dropActions
- resourceFilters
- markerImageProviders
- decorators
- markerHelp
- markerResolution
- capabilities
- projectNatureImages
- markerUpdaters
- documentProviders

Extension Point Usage

Examine who is using extension points of this plugin.

- org.eclipse.compare (2.0.0)
- org.eclipse.contribution.review (1.0.0)
- org.eclipse.contribution.soundpoint (1.0.0)
- org.eclipse.debug.ui (2.0.0)
- org.eclipse.jdt.debug.ui (2.0.0)
- org.eclipse.jdt.junit (2.0.0)
- org.eclipse.jdt.ui (2.0.0)
- org.eclipse.ui (2.0.0)

By double clicking on an example (org.eclipse.compare, just to pick one at random), we can see how to declare an extension to the pop-up menus. The whole extension is more complicated than we need, but this part of it seems relevant:

```
<extension point="org.eclipse.ui.popupMenus">
…
      <objectContribution
            id="org.eclipse.compare.AddFromHistoryAction"
            objectClass="org.eclipse.core.resources.IContainer" adaptable="true">
            <action
                  id="addFromHistoryAction"
                  label="%addFromHistoryAction.label"
                  tooltip="%addFromHistoryAction.tooltip"
                  menubarPath="replaceWithMenu"
                  enablesFor="1"
                  class="org.eclipse.compare.internal.AddFromHistoryAction">
            </action>
      </objectContribution>
</extension>
```

Ignore most of this for now. When copying an example, you are looking for structure, much of which you'll typically delete because it doesn't apply, or at least not yet. Here are the bits we know we need (looking at the on-line help, Platform Plug-in Developer Guide>>Reference>>Extension Points Reference>>Workbench>>org.eclipse.ui.popupMenus, for a reminder):

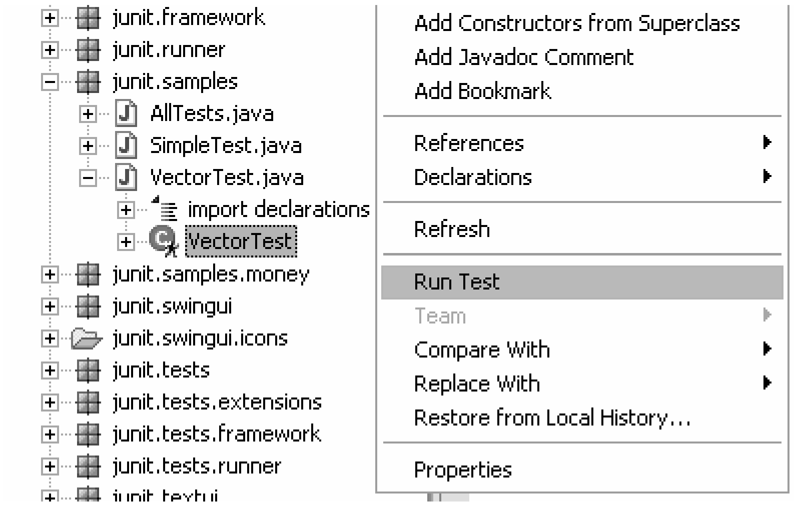
<extension point="org.eclipse.ui.popupMenus">	This says that we will extend a pop-up menu
<objectContribution	We extend it for all objects of a given type
id="org.eclipse.contribution.junit.runtest"	Unique ID
objectClass="org.eclipse.jdt.core.IType">	All selected IType's will have this item
<action	This is the menu item.
id="org.eclipse.contribution.junit.runtest.action"	Unique ID
label="Run Test"	How the item will appear in the menu
enablesFor="1">	The item is only enabled when there is exactly one element selected.
</action>	
</objectContribution>	
</extension>	

What is an IType? Eclipse (actually org.eclipse.jdt.core) defines a set of interfaces representing the elements of a Java program. In there you will find types representing projects, packages, compilation units and, in this case, types (classes and interfaces).

Now we can run the plug-in. We expect it to show up, but not do anything. If it doesn't show up, make sure you've entered the correct fully qualified type name (Kent screwed this up when writing.) Run it by selecting Run>>Run As>>Runtime Workbench. It will take a few seconds for the new workbench to appear. Select a type (not just the compilation unit), pop up the menu, and voila.

In general, when you make a contribution, you need to limit its availability to cases where it can possibly help. You don't want a context menu with 200 items, most of which wouldn't actually work with the current selection. This is the Relevance Rule:

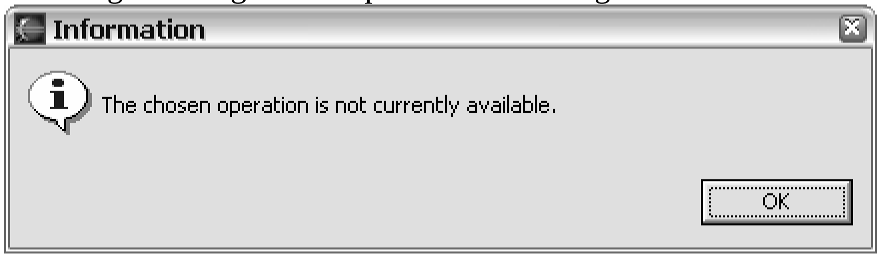
Relevance Rule: Contribute only when you can successfully operate



Object contributions are a bit optimistic by nature. Our “Run Test” item will show up for any type, not just test cases. The tradeoff is that if the filter for whether the menu item appears is only on the type, the plug-in code does not need to be loaded until the menu item is actually selected. The manifest file is declarative, so it can’t be the home for logic. While such a coarse filter might seem too optimistic, it works well in practice (see Further Reading for more filtering options).

Notice that we have violated the Relevance Rule by having “Run Test” show up for types that don’t have any tests. Sometimes rules are in conflict. Here, the conflicting rule is the Lazy Loading Rule. The only way to determine if “Run Test” is really relevant is to run the plug-in. We satisfy the Lazy Loading Rule at the expense of the Relevance Rule. Design is all about breaking the right rule at the right time.

Selecting the test gives us a polite error message.


The chosen operation is not currently available.

Eclipse prints a programmer-friendly error message in the console of the host workbench:

```
Could not create action.
Reason:
Executable extension definition for "class" not found.
```

This reminds us that we need to define a class for the action to invoke, which we’ll do in the next chapter. This polite, non-catastrophic behavior in the face of error leads us to the Safe Platform Rule:

> Safe Platform Rule: As the provider of an extension point, you must protect yourself against misbehavior on the part of extenders.

Before doing that, let’s review:

- Contributed an action labelled “Run Test” to all Java elements of type IType. So far we only have the menu item, not its implementation.
- We began working on our action by finding an example in Eclipse of a similar action. When we implement the logic of the action, we will also start with a similar action implementation.

Forward Pointers

- You can specify where items appear. See users of menubarPath for examples. (Use file search for all plugin.xml files for “menubarPath”.)
- Internationalization. Labels beginning with “%” will be used as a key into plugin.properties file. Replacing this file allows you to change the language of the plug-in. (Use file search for all plugin.properties files.)
- Better filtering. The object contribution offers an optional nameFilter attribute. Its value is a name pattern (e.g. “*Test.java”.) The most sophisticated filters are written with action expressions (search on-line help for “action expression”.)
- Class filter applies to all subtypes. If you look at the IMember hierarchy (ctrl-shift-h “IType”, you’ll see a large graph. If you wanted an action to be available for all members (fields, types, methods, and initializers), you would define the class attribute to be IMember.
- Working set. Looking at all projects at once can be overwhelming. In the Package Explorer, you can define a working set only containing projects you are interested in at the moment.
- Other object contributions. You can also contribute to the property pages for a given object type. See org.eclipse.ui.propertyPages.

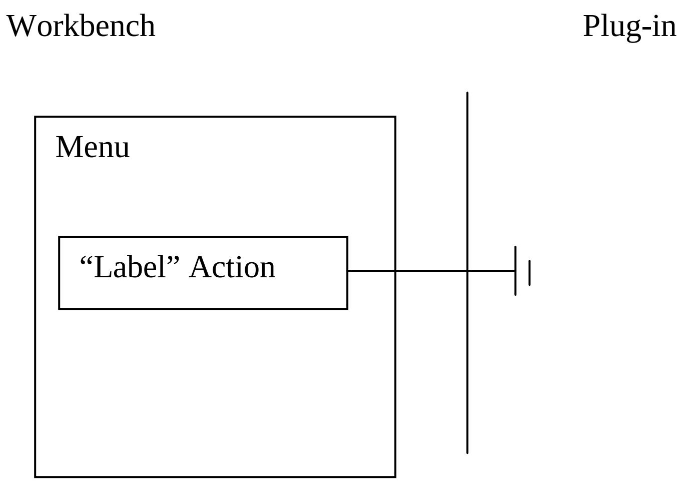
Class Action

In this chapter, we will see:

- How the workbench lazily creates the Action used in our plug-in.
- How an Action operates.
- Plugin dependency and build classpath management.

The class we are about to define will take a selected type, run the tests in that type, and display the results.

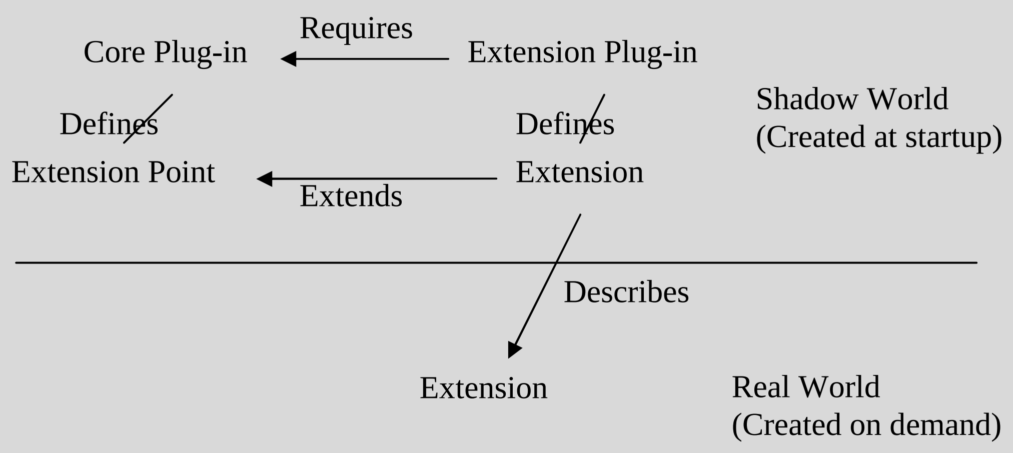
One of the consequences of the Lazy Loading Rule is the logic of the action won't be available when the user first sees the action in the user interface. All that exists before our action has been selected is a generic proxy for the menu item's action. The generic proxy action in the workbench (an example of the Proxy pattern) uses the information in the manifest to decide whether an item should appear for a particular selection, and whether the item should be enabled.



Before the action has been invoked

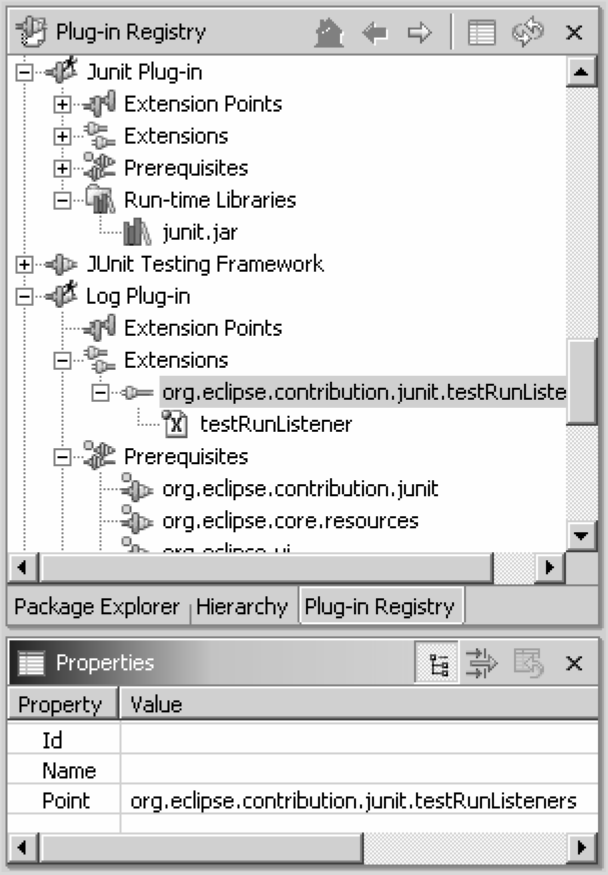
Sidebar: The Shadow World

On startup, the platform builds a shadow of all the plug-ins by reading all the manifest files. The contents of this shadow world are available to you through the class Platform.



The Plug-in Registry

You can see and explore the shadow world by opening the PDE Plugin-in Registry view with Window>>Show View>>Other…>>PDE Runtime>>Plugin-in Registry:



```
Junit Plug-in
    Extension Points
    Extensions
    Prerequisites
    Run-time Libraries
        junit.jar
JUnit Testing Framework
Log Plug-in
    Extension Points
    Extensions
        org.eclipse.contribution.junit.testRunListe
            testRunListener
    Prerequisites
        org.eclipse.contribution.junit
        org.eclipse.core.resources
        org.eclipse.ui
```

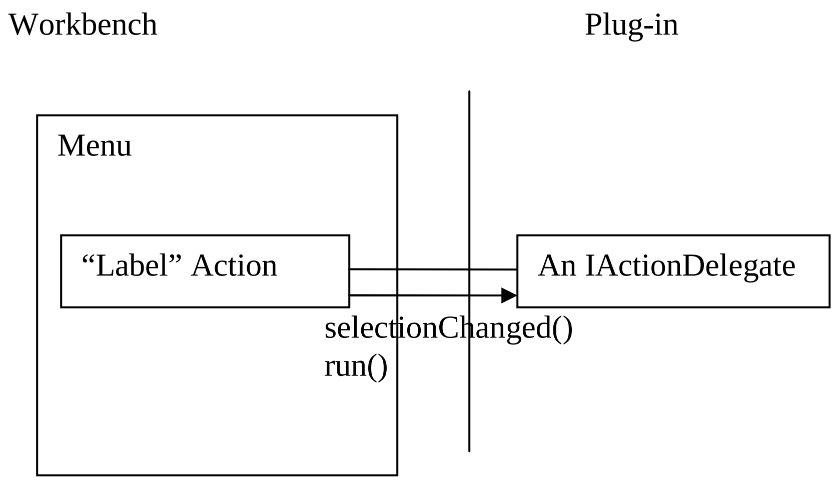
Property	Value
Id	
Name	
Point	org.eclipse.contribution.junit.testRunListeners

The above image shows that our Log Plug-in extends the testRunListeners extension point. The Properties view shows the details of the extension.

What object in the plug-in should be invoked when the menu item is selected? We need to specify the class of the object in the manifest, like our example from the previous chapter:

```
<action
        id="addFromHistoryAction"
        label="%addFromHistoryAction.label"
        tooltip="%addFromHistoryAction.tooltip"
        menubarPath="replaceWithMenu"
        enablesFor="1"
        class="org.eclipse.compare.internal.AddFromHistoryAction">
</action>
```

When the menu item is selected, the generic action will create an instance of AddFromHistoryAction. The action will be passed the current selection, and then told to run.



Invoking the action

How does AddFromHistoryAction implement these two methods? The Conformance Rule tells us that the action must conform to an expected interface. In Hello World, the interface was IWorkbenchWindowDelegate. As an alternative, actions can also conform to the IActionDelegate interface:

```
public interface IActionDelegate {
        public void run(IAction action);
        public void selectionChanged(IAction action, ISelection selection);
}

public class AddFromHistoryAction implements IActionDelegate…
```

AddFromHistoryAction stores the selection in selectionChanged, because the selection is not passed to run():

```
        public void selectionChanged(IAction a, ISelection s) {
                fSelection= s;
```

```
}
```

The work of the AddFromHistoryAction occurs in run(). The details don't concern us here, just the fact that the selection is used:

```
public void run(IAction action) {
        …fSelection…
}
```

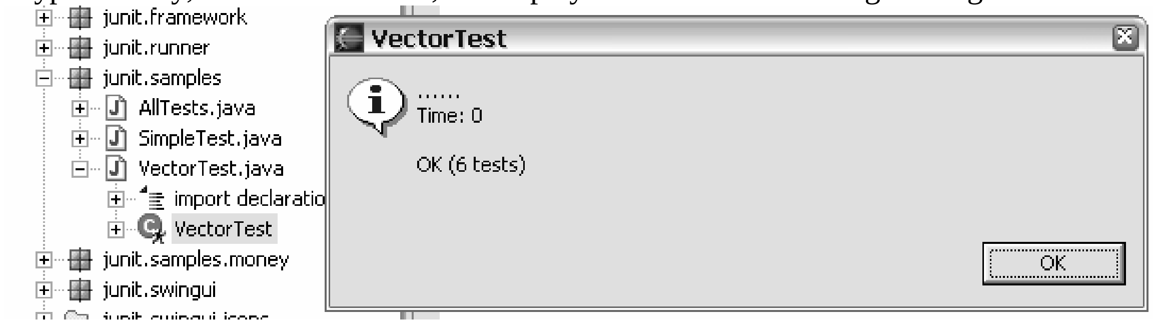
(Once again, we're applying the Monkey See/Monkey Do Rule to copy the structure for our application.) To implement the running of the tests, we also need to save the selection.

```
ISelection fSelection;
public void selectionChanged(IAction action, ISelection selection) {
        fSelection= selection;
}
```

The real action happens in run(). Imagine first that we have a method runTest(IType) that returns the output of running the tests in an IType (the details are too ugly and uninteresting to include here, but you can find the full source in the appendix.) Given this handy dandy method, here is the code for run():

```
public void run(IAction action) {
        if (! (fSelection instanceof IStructuredSelection))
                return;
        IStructuredSelection selection= (IStructuredSelection) fSelection;
        IType type= (IType) selection.toArray()[0];
        String output= runTest(type);
        MessageDialog.openInformation(null, null, output);
}
```

run() only works if called after the selection has been set to a IStructuredSelection (instead of, say, a text selection), so the guard clause ignores other selections. Then we get the IType out of the selection (selections can contain multiple entries, but because of the manifest we are guaranteed there will only be one selection.) We run the tests in the IType. Finally, as in Hello World, we display the results in a message dialog:



```
......
Time: 0

OK (6 tests)
```

Sidebar: Class Path Management with PDE:

PDE can help us to update the Java Build class path as we develop. When you refer to types from another plugin you have to state in the manifest that you require this plugin. In addition, since you want to compile your code, you also have to inform the Java tooling's build classpath about this fact. PDE can help you to keep the manifest and the build class path in synch. PDE offers different ways to do so, ranging from fully automatic to user initiated. We mostly use the user initiated way, since we want to know when the build class path changes. You can trigger a build class path either by in the manifest editor's Dependency page or by selecting the manifest file and executing "Update Classpath…".

The run() code above refers to IType, a type defined in the org.eclipse.jdt.core plug-in. As a consequence we have to add this plugin to our required plug-in list in the manifest. To do so we add a dependency on org.eclipse.jdt.core in the "requires" section of the manifest:

```
<requires>
  <import plugin="org.eclipse.core.resources"/>
  <import plugin="org.eclipse.ui"/>
  <import plugin="org.eclipse.jdt.core"/>
</requires>
```

After adding this line, we update the Java build class path (see sidebar: Class Path Management with PDE). Our mysterious runTest() code creates a dependency on one more plug-in, org.eclipse.jdt.launching, so we must add it as a dependency and update the class path before we can compile:

```
<requires>
  <import plugin="org.eclipse.core.resources"/>
  <import plugin="org.eclipse.ui"/>
  <import plugin="org.eclipse.jdt.core"/>
  <import plugin="org.eclipse.jdt.launching"/>
</requires>
```

We can imagine all sorts of wonderful things to add to the basic user interface, but if we spent our time doing that, we wouldn't be able to show you other cool stuff about Eclipse. Since our overall goal is to close the circle of programming tools, the next step, now that we have extended Eclipse, is to provide opportunities for further extension of what we've done. We'll do that in the next chapter. First, reviewing:

- We saw how our RunAction wasn't created until the menu item was selected.
- We saw how the selection is passed as part of selectionChanged(), but used in run(). These are the only two methods needed to act as the action behind a menu item.
- When implementing the logic of our action, we needed to declare that our plug-in was dependent on other plug-ins, and update the build classpath accordingly.

Forward Pointers

- IWorkbenchWindowActionDelegate is another interface your contributed Action can conform to. It is required when you contribute to the toolbar or the global menubar.
- Dynamic enablement and label changes. Once your plugin is loaded you can update the enablement state or the label precisely from code. To do so you get the generic IAction passed in to both ActionDelegate methods.

- When does selectionChanged get called? Due to the lazy loading it might not be clear when exactly your ActionDelegate's selectionChanged method is called. An easy to find it out is to add a System.out.println system in selectionChanged.
- Unrooted dialog—It's bad practice to have an unrooted dialog (we didn't pass in the parent Shell as the first parameter.) The best way to get at a Shell is from an IWorkbenchSite. If you cannot get at one the common solution in Eclipse (search for users of MessageDialog) is to have a utility method in your plug-in class that returns the active workbench shell.
- Other examples of the Lazy Loading Rule. Preference pages are also loaded lazily. Look carefully. The first time you click on a preference node, there is a brief pause while Eclipse loads the preference page. Subsequently the page loads instantly.
- Launch Configurations. The code above launches the tests manually. Eclipse provides infrastructure for launching and debugging processes. See org.eclipse.debug.core.launchConfigurationTypes.
- Long-running operations. The way we run our test can potentially block the UI thread for longer than is comfortable for a user. In particular, the user has no way to cancel the tests. Search for users of the type IRunnableWithProgress for examples of how to present cancellable operations.

Opening

Our goal in this chapter is to turn the basic activity of running a test into a resource for other functionality that depends on tests running.

> Invitation Rule: Whenever possible, let others contribute to your contributions

In this chapter, you'll see:

- Defining an extension point
- Defining an contribution interface
- Defining an extension

Now that we have the basic test running functionality, we want to expand it. If we were to expand it directly in the same plug-in, there would be no way for you to a) accept only functionality you wanted and b) see how to create extension points. As we near the end of the first section, we need to finish moving around the Contribution Circle.

Good programmers become intensely aware of their own processes, whether they are aware of how they use the available keystrokes and menus, how they design, or how they relate to other programmers. One interesting bit of self-awareness is how often we run our tests. Our idea is to write a log file entry every time tests are run. Later, we can analyze the log results to see how often we run tests and eventually gather all kinds of statistics. This is only one of the many extensions we could pick first—we might also like to see how our team members are testing, we might like aural feedback of test progress, we might like to notify a particular person when a particular test fails.

> Explicit Extension Rule: Declare explicitly where a platform can be extended

How are we going to provide extension points? We haven't been able to work test-driven so far, but we can ask Eclipse for feedback as we go. Instead of starting with the extension point, let's start with how the logging extension wants to look. First, create a new plug-in project called org.eclipse.contribution.junit.log (because it's sort of part of our JUnit contribution.) Where can we find a similar extension?

We know we want to have different classes we can notify when the tests run. A similar extension point is used in org.eclipse.core.resources.IMoveDeleteHook. We found this by browing the extension point reference in the online help for 'class "extension point"'. A moveDeleteHook is declared like this (copied right from the help documentation example section):

```
<extension point="org.eclipse.core.resources.moveDeleteHook">
  <moveDeleteHook class="org.eclipse.team.internal.MoveDeleteHook"/>
</extension>
```

What shall we call our extension point. In general, an extension point should accept many extensions (the move/delete hook, unfortunately for our purposes, doesn't). The general rule goes under the name of the Diversity Rule:

> Diversity Rule: Extension points accept multiple extensions

The Diversity Rule is a consequence of the Sharing Rule. If you have added an extension to an extension point, it should have no effect on our ability to add an extension to the same point. It's when implementing an extension point that you have to take the Diversity Rule into account.

Why isn't a run-time listener good enough? If you know anything about JUnit, you have seen that you can dynamically add listeners. However, how would such a listener know to add itself? The Lazy Loading Rule tells us that the first time the tests run, our new intended listener won't even have been loaded. The extension point mechanism avoids this by constructing the shadow world on startup. Your extension is guaranteed to exist in potential, even though your code hasn't been loaded. We'll see in a moment how the potential is realized.

We want an extension point for an object that listens to tests being run, but the extension point's name should be plural. How about "testRunListeners"? But extension points ids must be unique. So, we add the following extension declaration to the manifest of our logging plug-in project:

```
<extension point="org.eclipse.contribution.junit.testRunListeners">
  <testRunListener class="org.eclipse.contribution.junit.log.LogListener"/>
</extension>
```

This says that when an extension point named testRunListeners is invoked, an instance of LogListener should be created (by convention, this attribute is always called "class"). To make this work, we also need to invoke the Conformance Rule. LogListener will implement an interface that we have yet to define, but which will be invoked by our basic testing plug-in.

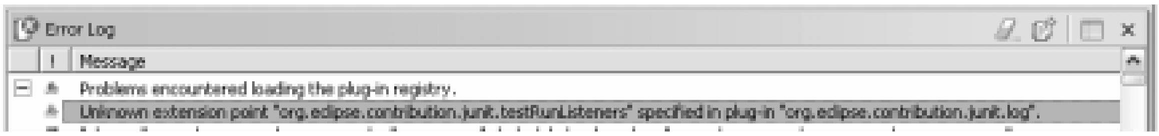
Now we can run the runtime workbench and see how Eclipse behaves.

Sidebar: Diagnosing Plug-ins

Sometimes when you have trouble with plug-ins, the problem is really in the internals of the plug-in logic. For these problems, the usual debugging methods suffice: running the workbench under the debugger and setting breakpoints, print statements, and test-driven development. What if the problem is in the manifest?

For these errors you have two primary resources. The first is the console in the host workbench. Message sent to System.out appear there. The second resource is the error log view in the runtime workbench. View the log by selecting Window>>Show View>>Other…>>PDE Runtime>>Error Log. Double clicking an error gives you a chance to see both the conditions under which you created the run-time workbench, and (under Status Details) the details of the error.

The error log in the run-time workbench tells us, not that we're surprised, that junit.log has asked for an extension point that doesn't exist.



```
Problems encountered loading the plug-in registry.
  Unknown extension point "org.eclipse.contribution.junit.testRunListeners" specified in plug-in "org.eclipse.contribution.junit.log".
```

Extension points must be declared before they are used. This provides an additional level of checking for manifest errors. It's unfortunately easy to mis-type an extension point id. So, now we need to declare the extension point in the plug-in org.eclipse.contribution.junit. We can look at the manifest for org.eclipse.core.resources (we saw this project name when looking at the help documentation for extension points.) There we see:

```
<extension-point name="%hookName" id="moveDeleteHook"/>
```

Notice that the id has no unique prefix. Eclipse, somewhat inconsistently, prepends the plug-in id as a prefix for all declared extension points. Following this example, we add the following to the org.eclipse.contribution.junit manifest:

```
<extension-point id="testRunListeners" name="Test Run Listeners"/>
```

Now the runtime workbench starts without a whimper. Our next job is to digest the extension that has been declared. Following the Lazy Loading Rule, our base plug-in will only load extensions when it is about to call them. Before we look for where else this happens in the system, where should this code live?

You remember how in the chapter on Getting Started, we deliberately did not create a Plugin class? It turns out that you almost always need a Plugin class, but we didn't want to confuse you just then. One of the reasons we mentioned for possibly needing a plug-in class was for storing shared information. The list of extensions is just such shared information. Now we have to create a plug-in class.

The first duty of the plug-in class is to be accessible to the other classes in the plug-in. It's the home for shared information. Out we pull handy-dandy Singleton. Yes, it creates a "global" variable, but here the whole purpose is to share information between objects, and the actual scope of the information is only within the plug-in, not the whole system (we'll see in the next chapter how to further reduce the visibility.)

The simplest plug-in superclass is org.eclipse.core.runtime.Plugin. We'll use it until we decide we need a more powerful godfather. The Singleton implementation is straight out of the book[1]. The plug-in class is defined in org.eclipse.contribution.junit.

```
public class JUnitPlugin extends Plugin {
	private static JUnitPlugin fgPlugin;

	public JUnitPlugin(IPluginDescriptor descriptor) {
		super(descriptor);
		fgPlugin= this;
	}

	public static JUnitPlugin getDefault() {
		return fgPlugin;
	}
}
```

[1] Gamma, Erich, et. al, *Design Patterns*, millions of copies sold.

(We don't have to make our plug-in dependent on org.eclipse.core.runtime because every plug-in is implicitly dependent on org.eclipse.runtime. This is inconsistent with the rest of the platform, where dependencies are explicit.)

We have to update the manifest, otherwise the plug-in class won't be instantiated:

```
<plugin
  id="org.eclipse.contribution.junit"
  name="Junit Plug-in"
  version="1.0.0"
  provider-name=""
  class="org.eclipse.contribution.junit.internal.JUnitPlugin">
```

Where does extension loading happen in the system? First, how do we find examples of extension loading? Now, for the first time in the book, we have to invoke a little magic. We happen to know that the method IExtensionPoint.getConfigurationElements() is invoked to get the descriptions of the extensions to be loaded. We can search for all references to this method. One that looks promising is org.eclipse.jdt.launching.LaunchingPlugin.

<<<We should pick another example, the use of a HashMap is just bogus. This means in the case of an ambiguity the last retrieved extension wins. This doesn't make sense and should be replaced with a List (actually we have fixed this in the latest)>>>

```
private void initializeVMConnectors() {
	IExtensionPoint extensionPoint=
		Platform.getPluginRegistry().getExtensionPoint(…);
	IConfigurationElement[] configs= extensionPoint.getConfigurationElements();
	…
	fVMConnectors = new HashMap(configs.length);
	for (int i= 0; i < configs.length; i++) {
		try {
			IVMConnector vmConnector=
				(IVMConnector)configs[i].createExecutableExtension("class");
			fVMConnectors.put(vmConnector.getIdentifier(), vmConnector);
		} catch (CoreException e) {
			…
		}
	}
	…
}
```

The process of loading extensions is to:

1. Get the extension point from the platform (remember that the platform on start-up reads all the manifest files, so our extension point will be there).
2. Get the descriptions of the extensions (IConfigurationElements). These are also created by reading the manifest files, looking for <extension point=…>.
3. For each extension description, create an object whose class will be based on the contents of the class attribute. Validate that the defined attributes are complete.
4. Save the newly created extensions in a collection. (in the example a Map, but we only need a List).

Since our extensions will be loaded only when they are first accessed, we can lazily instantiate the test run listeners. First, the boilerplate of lazy instantiation:

```
private List fListeners;

private List getListeners() {
	if (fListeners == null)
		fListeners= computeListeners();
	return fListeners;
}
```

Before we can load our extensions, the Conformance Rule tells us that we have to have an interface for the extensions to conform to. We'll call it ITestRunListener, and define it in org.eclipse.contribution.junit:

```
public interface ITestRunListener {
}
```

It doesn't need any operations at the moment, since we are only instantiating it, not calling it.

Now we copy the structure from our example to compute the listeners:

```
private List computeListeners() {
	IExtensionPoint extensionPoint=
Platform.getPluginRegistry().getExtensionPoint("org.eclipse.contribution.junit.testRu
nListeners");
	IConfigurationElement[] configs= extensionPoint.getConfigurationElements();

	List listeners= new ArrayList(configs.length);
	for (int i= 0; i < configs.length; i++) {
		try {
			ITestRunListener listener= (ITestRunListener)
configs[i].createExecutableExtension("class");
			listeners.add(listener);
		} catch (CoreException e) {
			e.printStackTrace();
		}
	}
	return listeners;
}
```

The Safe Platform Rule tells us as the implementor of an extension point to be tolerant of fallible extenders. That's why we catch the CoreException. Ignoring it completely is bad form, but it will do for now until we talk about serviceability. We are careful to catch it separately for each extension we create, so if one extension doesn't work for some reason, the rest will load correctly.

Sidebar: Extension Instantiation

Frameworks have to solve the problem of how to create client objects. By definition, you don't know what objects you'll be creating when you write the framework. ConfigurationElement.createExecutableExtension() is a flavor of Factory Method. The important difference is that Factory Method creates a static dependency on the extension class. Here, we use Java’s class loading facility to load and instantiate the extension class, without having any static dependency between the extension creator and the extension.

When we run now, we get an error in the host workbench console:

```
org.eclipse.core.runtime.CoreException[2]: java.lang.ClassNotFoundException:
        org.eclipse.contribution.junit.log.LogListener…
```

We haven’t yet created our LogListener class. The Conformance Rule tells us that the LogListener should conform to ITestRunListener. We define the LogListener in our new plug-in project, org.eclipse.contribution.junit.log:

```
public class LogListener implements ITestRunListener {
}
```

Now if we run, the plug-ins load fine, but nothing happens. When we run a test in the RunAction, we want to ask the JUnitPlugin to notify the extensions. We’ll start with modifying the RunAction:

```
public void run(IAction action) {
    if (! (fSelection instanceof IStructuredSelection))
        return;
    IStructuredSelection selection= (IStructuredSelection) fSelection;
    IType type= (IType) selection.toArray()[0];
    String output= runTest(type);
    JUnitPlugin.getDefault().fireTestRunFinished(type);
    MessageDialog.openInformation(null, type.getElementName(), output);
}
```

Sidebar: Programming by Intention

Eclipse provides wonderful support (if we do say so ourselves) for this style of programming, where you start with expected usage and work backwards to the implementation. When we enter a reference to an undefined method, one of the Quick Fixes is to create a stub for the method.

Now we have to add the “fire” method to notify the extensions. We loop through the extensions, invoking the (as-yet-undefined) method testRunFinished():

```
    public void fireTestRunFinished(IType testClass) {
        for (Iterator all= getListeners().iterator(); all.hasNext();) {
            ITestRunListener each= (ITestRunListener) all.next();
            each.testRunFinished(testClass);
        }
    }
```

We have to declare the method in the interface:

```
public interface ITestRunListener {
    void testRunFinished(IType testClass);
}
```

We also have to implement the method in LogListener:

```
public void testRunFinished(IType testClass) {
    System.out.println(testClass.getFullyQualifiedName());
}
```

For the moment, we don't bother writing a persistent log entry anywhere. When we run our menu item, though, the name of the test is printed on the host workbench console.

We're almost done with the circle. Our final task is to prepare our contribution for publication in the wide wide world. Before we do that, we'll review. We:

- Defined an extension point, testRunListeners. We did this by defining its use, but you can define the point and extension in either order.
- Defined an extension and its implementation, LogListener.

Forward Pointers

- User Arbitration—If extensions share a limited resource (like space on an icon or space in a perspective), the user is given the responsibility to decide which extensions will be active.
- Loading extensions can be time consuming, so always show the busy cursor during loading. See users of BusyIndicator for examples of use.
- Documenting extension points. Use an HTML template similar to that used by the help documentation.
- Defining an extension point schema file (.exsd). PDE offers a guided way to define an extension in the manifest editor. This requires us to tell PDE more about our extension and its structure. This information is captured in an XML schema file with the suffix .exsd. PDE provides a wizard to simplify this task.
- IExectuableExtension. If you need additional arguments for the extensions created by the factory, define them in the XML, declare your extension to implement IExecutableExtension, and the parameters will be passed to your newly created object.
- Status. All plug-in objects have a log, to which you can append an IStatus. Look for references to getLog() for examples.
- Safe Runnable. Each extension notification should be invoked in a SafeRunnable to ensure that all notifications are sent, even if one has a problem.

API

In this chapter, we'll prepare our plug-in structure for public consumption. We will:

- Prepare our plug-in for local deployment
- Describe the rules for API design
- Apply the rules by hiding externally irrelevant detail

We're almost all the way around the Contribution Circle. We've extended the environment, invited contributions to our contribution, and contributed further. Are we finished? Not quite. First, we have to deploy our plug-in into Eclipse.

<<EG this addition is required to really close the cycle for the first time - the current library section in the plugin.xml has a bug>>

Before distributing the plug-ins, let's try to use them ourselves. Up to now we have only run our plug-ins in development mode. During development we don't have to package our classes as a JAR file with the name specified in the library section of the manifest file. The classes are found magically in the project's "bin" folder. This is convenient. It significantly shortens the development cycle and removes the JAR creation step. However, before making the plug-in available we have to run them from a JAR file as well. Therefore, for a quick test let's create the two library JAR files (junit.jar and log.jar) into their projects of our run-time workspace. We can then copy the project folders from the run-time workspace into the host workspace's plug-in folder and restart the workbench. Obviously this is not a clean packaging of our plugins, but it is good enough for the quick sanity check. To create the JARs we use the Eclipse JAR exporter:

1. Select the src folder of the org.eclipse.contribution.junit project.
2. File>>Export>>JAR file.
3. Enter the junit.jar in the org.eclipse.contribution.junit project of the run-time workspace as the export destination.
4. Press Finish.

Do the same in org.eclipse.contribution.junit.log to create the log.jar file.
Now that we have created the JARs we copy the projects from the run-time workspace into the "plugins" folder of your host workspace. Shutdown, restart Eclipse and try it…bommer… nothing happens and the error log shows us the somewhat scary error message that the class ITestRunListener was not found:

```
java.lang.NoClassDefFoundError: org/eclipse/contribution/junit/ITestRunListener
      at java.lang.ClassLoader.defineClass(ClassLoader.java:502)
…
org.eclipse.contribution.junit.internal.JUnitPlugin.computeListeners(JUnitPlugin.java:
50)
```

What happened? Eclipse allows you to define which classes from your library JAR to export to other plug-ins. Eclipse uses the defensive default that none of your classes are exported. However, our logging plug-in requires the ITestRunListener class from org.eclipse.contribution.junit and hence we get the class not found exception. To fix this we have to adjust the library element of the manifest in org.eclipse.contribution.junit. In the manifest we have to specify that we intend to export all the classes in the junit.jar library. As we will see in the next section where we talk about API considerations, exporting all classes is fair play. You can do adjust the library definition either with the PDE manifest editor in the Runtime tab ("Export the entire library") or by editing the plugin.xml file directly:

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

By setting the export name pattern to "*" we are exporting all our classes.

We've discharged our obligation to communicate with the computer (i.e. our code works), but we haven't thought much about how to communicate with our fellow programmers. We need to take what we have and arrange it so that readers will know what is important and what is not, what is stable and what is likely to change.

The Invitation Rule tells us to invite contributions. One way to invite contributions is to publish extension points, *a la* the Explicit Extension Rule. Another form of contribution is to surface some of your programmatic interface for others to use. Published APIs should also be explicit:

Explicit API Rule: Separate the API from internals

Freedom is the designers stock in trade. If you publish all functionality as potentially useful APIs, when you want to change the internal structure, you break all clients. Without carefully defining the API, as time goes on, you as a designer have less and less freedom.

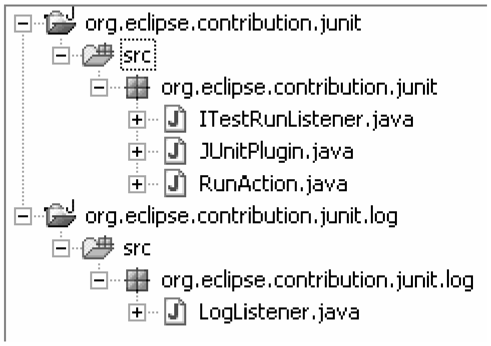
Stability Rule: Once you invite someone to contribute, don't change the rules

Once you have published an API, it is rude to change it without a compelling reason. If I have written a plug-in based on your version 1, I'd like it to work without modification on your version 2. Keeping the API stable is to both of our benefits. Except if it's wrong. The need to balance stability and growth leads us to the third API-related rule:

Revelation Rule: Reveal the API a little at a time

The convention for combining these rules in Eclipse is for every plug-in intended for external use to have two name spaces. If a package name contains "internal", the classes it contains are not intended to be used outside the plug-in. Other packages can be considered published, available for subsequent contributors' use.

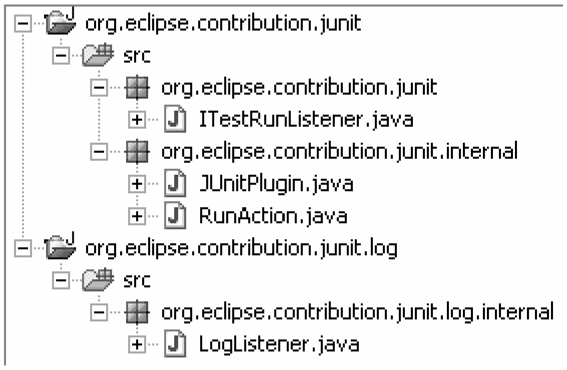
Here is our current package structure:



```
org.eclipse.contribution.junit
  src
    org.eclipse.contribution.junit
      ITestRunListener.java
      JUnitPlugin.java
      RunAction.java
org.eclipse.contribution.junit.log
  src
    org.eclipse.contribution.junit.log
      LogListener.java
```

Which of these types are intended for use outside its defining plug-in? Only ITestRunListener. It is intended to be shared between the two plug-ins. The rest of the types should be in internal packages.

We can move the types easily enough, dragging the types to their new packages:



```
org.eclipse.contribution.junit
  src
    org.eclipse.contribution.junit
      ITestRunListener.java
    org.eclipse.contribution.junit.internal
      JUnitPlugin.java
      RunAction.java
org.eclipse.contribution.junit.log
  src
    org.eclipse.contribution.junit.log.internal
      LogListener.java
```

Unfortunately, Eclipse only automatically updates package references in Java code. The references in the manifest must also be updated before our plug-ins work again. In the org.eclipse.contribution.junit manifest, the declaration of the RunAction must be updated to reflect the new fully qualified class name:

```
<action
    label="Run Test"
    class="org.eclipse.contribution.junit.internal.RunAction"
    enablesFor="1"
    id="org.eclipse.contribution.junit.runtest.action">
</action>
```

Notice that we don't feel the need to change the id. As long as the id is unique, it will work fine. The location of the plug-in class must also be updated:

```
<plugin
  id="org.eclipse.contribution.junit"
  name="Junit Plug-in"
  version="1.0.0"
  provider-name=""
  class="org.eclipse.contribution.junit.internal.JUnitPlugin">
```

The declaration of the log listener extension must also be updated:

```
<testRunListener class="org.eclipse.contribution.junit.log.LogListener"/>
```

We've seen the three rules of API design, intended to give other plug-ins appropriate access to our contribution without tying our hands. Instead of four public types, we have only one. One of our plug-ins has no published interface at all.

This completes the first Contribution Circle. We've taken a basic idea, a test runner, implemented it, extended it, and simplified it for publication. In the next section, we'll take a deeper look at all the resources available to you as you contribute to Eclipse, and open your contributions for further contribution.

Sidebar: non transivity of prerequisites

Where does this really go?

Forward Pointers

- In a manifest you can enforce the separation into API and internal by only exporting API classes. Any attempt to use a non-exported class will result in a runtime error. Notice, that this support is not used in the Eclipse platform. It was considered to be too restrictive. In particular, just receiving a class not found exception when using an internal class at run-time is rather harsh. It also isn't consistent the Revelation Rule.
- If you need to publish the plug-in class, don't publish the whole thing. Only publish a Façade (an object with a restricted subset of the whole plug-in protocol that forwards messages to the real plug-in.)
- Consider documenting your API using Javadoc.

<<<we should describe some of the Eclipse API rules somewhere: the idea of "soft final" the subclasser/implementor contract etc.

<<<should add an explanation what we mean by stable API, i.e., API contract compatibility>>>

Circle Two

The first circle has taken us from the most basic interface and model for JUnit to an extensible plug-in. The last chapter is perhaps the most important, though. In spending the time to clean up our code, to bring it into alignment with our rules, we gave ourselves the chance to learn the lessons of our recent experience.

Designing anything new is a bit like taking a big jump. When you land, chances are you aren't going to be perfectly in balance. If you're not in balance, your chances of a successful second jump diminish drastically. Taking one more pass through the code, making sure that you've expressed all you can express, abstracted all you can profitably extract, eliminated all you can safely eliminate, gives you the chance to get your feet firmly back underneath you. Once you've done that, you're ready for the next leap.

We've found the value of design rationalization at all times scales.

- After a few minutes, you can learn about what you've just done by arranging it nicely.
- After a few hours, cleaning up before checking in is good for you and the rest of your team.
- At the end of a week, taking the time make sure the package, API, and plug-in structure is clean will help you in the following week.
- At the end of a release, exploring and beginning large-scale restructuring will give you something profitable to do as you begin the next push.

At the end of this circle our design rationalization will involve preparing our plug-ins for distribution as a feature.

We have two goals with this second pass through the Contribution Circle.

- Make the user interface friendlier.
- Make our plug-ins fit for public consumption.

In the process you'll watch as the inter-plug-in structure becomes clearer, and you'll see examples of the most important functionality Eclipse has to offer contributors:

- Views
- Menus
- Preferences
- Logging and serviceability
- Internationalization
- Help

At the end of this circle, you'll be ready to develop some serious tools, but you still won't have a grasp on everything Eclipse has to offer. For that matter, at the end of Circle Three you still won’t have a grasp on *everything* Eclipse has to offer, but you’ll be ready to learn on your own.

A View with Room

The first thing our JUnit contribution needs is a proper user interface. We'd like a view that let's us pick a test, run it, and see the results. In this chapter we'll see:

- How to contribute a view.
- How to construct a view out of widgets.
- How to connect our view to an existing plug-in.

Sidebar: Why not a perspective?

We also might contribute a perspective as a way of introducing JUnit into Eclipse. We chose a view instead. Why? We have to look at the role of tests in development to answer this question.

A perspective is intended to big a context for solving a whole problem. Thus, there is a perspective for Java programming which helps in manipulating Java programs and a debug perspective for diagnosing problems. If we were to provide an environment for project management, we would likely create a project management perspective. The resources used in project management are strongly connected to each other, but only weakly connected to programming resources.

In a different perspective it is difficult to seamlessly integrate functionality. Since JUnit testing is expected to be part of programming, we need to contribute a lighterweight concept, like a view, that fits into the existing Java perspective.

Integration Rule: Integrate your contribution with existing contributions.

A view displays information in a special format. How do we contribute a view? We'll look for an example. For instance, how is the Task view contributed?

Searching for "*TaskList*" yields several classes all found in the org.eclipse.ui plug-in. Looking at the manifest file shows this declaration:

```
<extension point="org.eclipse.ui.views">
   …
  <view
      name="%Views.TaskList"
      icon="icons/full/cview16/tasks_tsk.gif"
      category="org.eclipse.ui"
      class="org.eclipse.ui.views.tasklist.TaskList"
      id="org.eclipse.ui.views.TaskList">
  </view>
</extension>
```

Views need a name, an id, and a class (we'll ignore the rest for the moment—you can read the online help for details). We'll create a new plug-in for our user interface, org.eclipse.contribution.junit.ui. Then we can add the view contribution to the manifest:

```
<extension point="org.eclipse.ui.views">
  <view
      name="JUnit"
```

```
        class="org.eclipse.contribution.junit.ui.JUnitView"
        id="org.eclipse.contribution.junit.ui.view">
    </view>
  </extension>
```

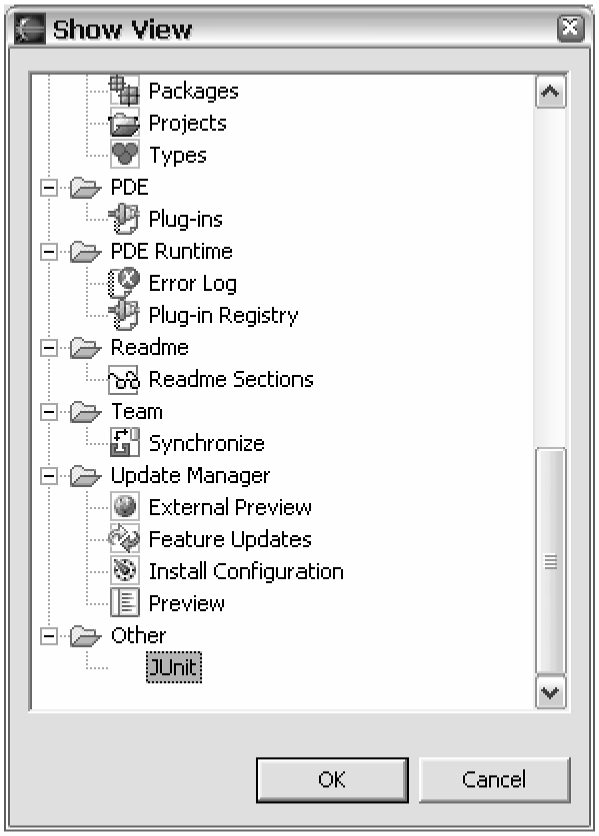
Sidebar: Why another plug-in?

Lumpers and splitters

Also XP style, the simplest thing that could possible work, start with a single plugin and then split.

Need to describe the alternative of using a fragment instead of a plugin and its consequences

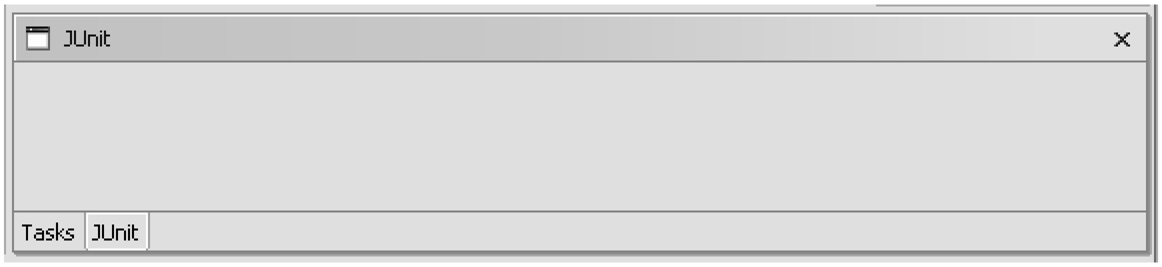
We can run the runtime workbench and see our new view by executing Window>>Show View>>Other…>>Other:



This violates the Integration Rule,i.e., the JUnit views shows up under Other rather than trying to integrate into an existing category. Since it is used for Java development, categorizing it under Java would make sense (however there already is a JUnit view in there). To define the category we add a category attribute to declaration of the view in the manifest:

```
          <extension point="org.eclipse.ui.views">
                <view
                      …
                      category="org.eclipse.jdt.ui.java"
                …
            </view>
        </extension>
```

Selecting the item just results in an error. We need to implement our class. Views need to conform to IViewPart. The easiest way to do this is to subclass ViewPart. We do this. The default implementation yields:



Our view doesn't display anything, but there it is next to the task view, the console, and so on. The name of the view appears both in the title and the tab.

The next job is to get the view to display the results of running the tests. If we could make JUnitView an ITestRunListener, we could use the notification to display the results. Unfortunately, ITestListener only notifies listeners of the name of the test, not the results. We need to add a parameter to the notification. To do this we need to move our mysterious test running method to JUnitPlugin and change the interface a bit. First, moving runTest(). We can define runTest() easily enough in JUnitPlugin instead of RunAction. Invoking it, we write:

```
public void run(IAction action) {
    if (! (fSelection instanceof IStructuredSelection))
        return;
    IStructuredSelection selection= (IStructuredSelection) fSelection;
    IType type= (IType) selection.toArray()[0];
    String output= JUnitPlugin.getDefault().runTest(type);
    JUnitPlugin.getDefault().fireTestRunFinished(type);
    MessageDialog.openInformation(null, type.getElementName(), output);
}
```

Any time you have two calls in a row to a different object, you likely should move some responsibility to the called object. So here. We can bundle the call to runTest() and fireTestRunFinished() together, firing the notification automatically every time tests are run. This simplifies RunAction:

```
public void run(IAction action) {
    if (! (fSelection instanceof IStructuredSelection))
        return;
    IStructuredSelection selection= (IStructuredSelection) fSelection;
    IType type= (IType) selection.toArray()[0];
    String output= JUnitPlugin.getDefault().runTest(type);
    MessageDialog.openInformation(null, type.getElementName(), output);
}
```

Now we can add a parameter to fireTestRunFinished:

```
public void fireTestRunFinished(IType testClass, boolean success) {
			for (Iterator all= getListeners().iterator(); all.hasNext();) {
				ITestRunListener each= (ITestRunListener) all.next();
				each.testRunFinished(testClass, success);
			}
		}
```

This requires that we add a parameter to ITestRunListener and all of its implementors (Eclipse now supports this as an atomic refactoring):

```
	ITestRunListener
		void testRunFinished(IType testClass, boolean success);
	LogListener
		public void testRunFinished(IType testClass, boolean success) {
			System.out.println(testClass.getFullyQualifiedName() + "  " + success);
		}
```

The rest of the test listeners, extensions of the point org.eclipse.contribution.junit.testListeners, are connected the first time a test is run. Our first thought was to make JUnitView another extension:

```
	<extension point="org.eclipse.contribution.junit.testRunListeners">
	  <testRunListener class="org.eclipse.contribution.junit.ui.JUnitView"/>
	</extension>
```

However, if we did this, we would have one instance of JUnitView created as part of extending the view extension point, and another instance created when the extensions of testRunListener were created. Instead, we need to dynamically add the view as a listener when it is created:

```
	public class JUnitView extends ViewPart implements ITestRunListener {
		public void createPartControl(Composite parent) {
			JUnitPlugin.getDefault().addListener(this);
		}
		public void testRunFinished(IType testClass, boolean success) {
		}
	}
```

Notice, that in this case the lazy loading isn't an issue since we only want to be registered as a listener when the view is visible. In the case of the log listener we want to be notified in every case, even the plug-in isn't loaded yet.

<<<To do: side bar: what can activate a plug-in.1) an extension point that the plugin implements is implemented 2) another plug-in calls a method in the plugin.>>

Style hint (could come later): when implementing a listener in an API class, consider implementing the listener in an inner class so that you do not have to expose the implemented listener methods as API.

We must also add a dependency on the junit plug-in:

```
	<requires>
	  <import plugin="org.eclipse.core.resources"/>
	  <import plugin="org.eclipse.ui"/>
	  <import plugin="org.eclipse.contribution.junit"/>
	</requires>
```

Finally, we need to be able to dynamically add listeners to JUnitPlugin:

```
JUnitPlugin
public void addListener(ITestRunListener listener) {
	getListeners().add(listener);
}
```

We can verify that the view is getting updates by printing to the console when the view is notified that a test ran. What we really want to do is change the color of the view depending on whether the test ran, and to display the name of the test. We can display the name of the test by adding a Label to the view:

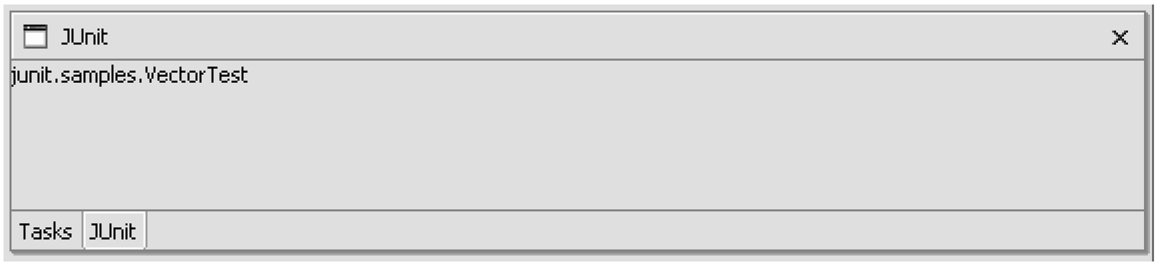
```
private Label fTest;

public void createPartControl(Composite parent) {
	fTest= new Label(parent, SWT.NONE);
	JUnitPlugin.getDefault().addListener(this);
}
```

When the view is notified that a test ran, it changes the contents of the label to the name of the test:

```
public void testRunFinished(IType testClass, boolean success) {
	fTest.setText(testClass.getFullyQualifiedName());
}
```

Now when we run a test, the name is displayed:



```
junit.samples.VectorTest
```

Changing the color takes a bit of poking around. We ended up looking at references to the class org.eclipse.swt.graphics.Color. You can create colors by specifying red, green, and blue components. Using this, we get:

```
public void testRunFinished(IType testClass, boolean success) {
	fTest.setText(testClass.getFullyQualifiedName());
	int background = success ? SWT.COLOR_GREEN : SWT.COLOR_RED;
	fTest.setBackground(fTest.getDisplay().getSystemColor(background));
}
```

Running it, sure enough, the background of the view turns a shocking primary color.

Next we need to make it possible to run tests from the JUnit view. First, however, we can review:

- We extended the view by providing an extension to org.eclipse.views and implementing a subclass of ViewPart.
- We made the view listen to tests, but dynamically, not by extending testRunListeners.

- We created and updated a widget in the view.

Could add an inset on "Eat your own dog food" section. When developing tools for ourselves we are in the deluxe position to use them as we develop and by doing so to uncover problems early on.

An easy way to export a plugin from your run-time workspace so that it can be copied into the host workspace is to use Ant. Here is a simple script that exports our plugin to an export directory. It makes sure that the plugin is up to date and creates the corresponding JAR files:

```
<project name="org.eclipse.contribution.junit" default="export" basedir="..">
	<target name="init">
		<tstamp/>
		<property name="destdir" value="../../plugin-export" />
		<property name="plugin"  value="org.eclipse. contribution.junit" />
		<property name="dest"  value="${destdir}/${plugin}/>
	</target>

	<target name="build" depends="init">
	<eclipse.incrementalBuild project="${plugin}" kind="incr"/>
	</target>

	<target name="export" depends="build">
		<mkdir dir="${destdir}" />
		<delete dir="${dest}" />
		<mkdir dir="${dest}" />

		<jar
			jarfile="${dest}/junit.jar"
			basedir="bin"
		/>
		<copy file="plugin.xml" todir="${dest}"/>
	</target>
</project>
```

Forward Pointers

- Perspective Extensions: we can define a perspective extension to define where the view should appear in an existing perspective (see the extension point org.eclipse.ui.perspectiveExtensions)
- SWT managing system resources.

Run, Test, Run

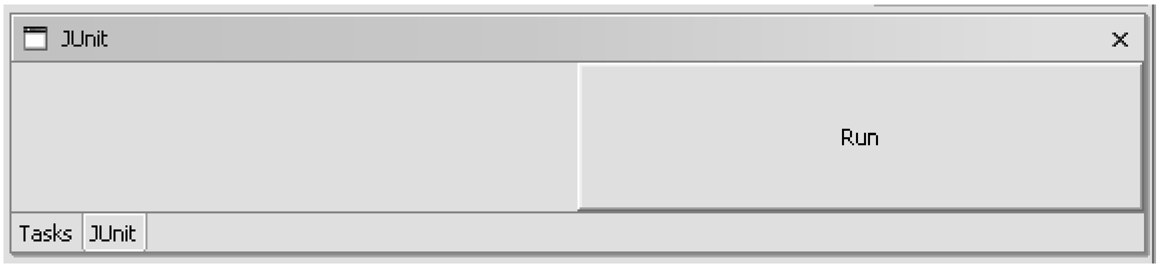
Now that we can see the test that ran, and its results, we'd like to be able to run the same test again. In this chapter, we will:

- Add a button to the view.
- Connect the button's action to the view.

Looking at the definition of Button, it looks like all we need to do to get a button is add it:

```
public void createPartControl(Composite parent) {
	fTest= new Label(parent, SWT.NONE);
	Button run= new Button(parent, SWT.PUSH);
	run.setText("Run");
	JUnitPlugin.getDefault().addListener(this);
}
```

Sure enough, we get a button (the layout is ugly, but we'll learn about that later.)



The label is on the left, the button on the right. Pressing the button doesn't do anything. How do we connect a button's action? We found more than a thousand references to Button. Here's one picked at random, in org.eclipse.team.internal.ui.IgnorePreferencePage:

```
addButton = new Button(buttons, SWT.PUSH);
…
addButton.addListener(SWT.Selection, new Listener() {
	public void handleEvent(Event e) {
		addIgnore();
	}
});
```

this is using what we call "untyped listeners, i.e., you get a generic handleEvent callback. The passed in argument has the generic Event type. The recommended way is to use a typed listener with a specific event method. In this case the listener that should be used should be a SelectionListener and look something like:

```
// Add listeners
fUsePatchFileButton.addSelectionListener(
	new SelectionAdapter() {
		public void widgetSelected(SelectionEvent e) {
			fShowError= true;
			setEnablePatchFile(!getUseClipboard());
```

```
                                        updateWidgetEnablements();
                                }
                        }
                );
```

Listeners with typed events should be familiar to all Java programmers. It is the same event handling model as is used in AWT and Swing. Copying the above, we get:

```
        public void createPartControl(Composite parent) {
                fTtest= new Label(parent, SWT.NONE);
                Button run= new Button(parent, SWT.PUSH);
                run.setText("Run");
    ->change to selection listener
                run.addListener(SWT.Selection, new Listener() {
                        public void handleEvent(Event e) {
                                runTest();
                        }
                });
                JUnitPlugin.getDefault().addListener(this);
        }

        private void runTest() {
                System.out.println("Run");
        }
```

We try it and see “Run” in the console of the host workbench. Now all we have to do is tell JUnitPlugin to run the test that just ran. Before we can do that, we have to save the Type:

```
        private IType fTestClass;
        public void testRunFinished(IType testClass, boolean success) {
                fTestClass= testClass;
                fTest.setText(testClass.getFullyQualifiedName());
                int background = success ? SWT.COLOR_GREEN : SWT.COLOR_RED;
                fTest.setBackground(fTest.getDisplay().getSystemColor(background));
        }
```

Now when we want to run, we pass the same Type back to the JUnit plug-in:

```
        private void runTest() {
                JUnitPlugin.getDefault().runTest(fTestClass);
        }
```

We try it, and it runs. But why can’t we change the test class from the view? Why, indeed. We’ll do that in the next chapter. Reviewing, we:

- Added a button to our view.
- Connected the button’s action back to the view.
- Ran the tests successfully when the button was pressed.

Forward Pointers

- Layout see http://www.eclipse.org/articles/Understanding%20Layouts/Understanding%20Layouts.htm
- Using JFace to implement a View

Invitation to Menu

In this chapter, we will see:

- How to create a view menu so that others can contribute to it. View menus are complicated enough that they are generally created programmatically. However, we would still like to accept contributions.
- How to call a dialog to ask the user to select a type.

The Invitation Rule says that where possible, we should invite others to contribute to our contributions. View menus are an excellent example, especially since we need to create one to allow us to choose a new type. If we create an extension point, then create our menu as an extension, others will also be able to extend our menu.

The contents of context-sensitive menus can be extremely complex:

- Different combinations of items based on the selection or combination of selections.
- Complex enable/disable logic.

Trying to express all the combinations in the manifest would result in either hideously complicated XML, or in menus that had far too many items. View menus are mostly constructed in code, and they are not declared as extension points in the manifest.

Because we can't declare view menus as extension points declaratively, we have to tell Eclipse about our menu so it can still act as an extension point. Where will we find an example to copy? Searching for references to registerContextMenu(), we find ccvs.ui.HistoryView:

```
MenuManager menuMgr = new MenuManager();
Menu menu = menuMgr.createContextMenu(tableViewer.getTable());
menuMgr.addMenuListener(new IMenuListener() {
…
tableViewer.getTable().setMenu(menu);
getSite().registerContextMenu(menuMgr, tableViewer);
```

Adapting this to our application, we get:

```
public void createPartControl(Composite parent) {
	test= new Label(parent, SWT.NONE);
	…
	MenuManager menuMgr = new MenuManager();
	Menu menu= menuMgr.createContextMenu(test);
	test.setMenu(menu);
	menuMgr.add(new Separator(IWorkbenchActionConstants.MB_ADDITIONS));
	getSite().registerContextMenu(menuMgr, getSite().getSelectionProvider());
}
```

(Without the separator for named "additions", we get an error in the runtime workbench. The separator provides a place for contributed menu items to go.) <<<the rule of the workbench is: If the menubarPath attribute is not specified, the Workbench will first

attempt to add the action in the group "additions". If the group "additions" does not exist, then the action is added to the end of the context menu.>>> KB: I tried it. Without a menubarPath, the item doesn't appear.

Looking at the online help for registerContextMenu() (search for "registerContextMenu"), we see that we need to add a viewer contribution, if we want to contribute our menu items declaratively.M (most commonly view menu items are completely contributed in code.). The reasons for this are:

- We use the declarative approach to get lazy loading. In the case of a view menu the view is already loaded hence there are no wins with regard lazy loading
- The declarative approach in XML is less powerful than Java code hence if there is a choice we prefer to do it in code. In particular it is easier to set a breakpoint in Java code than in XML.
- A context menu should be as short as possible and it should be highly context sensitive. Again this is easier to achieve in code.

Searching for examples (file search in plugin.xml for "viewerContribution"), we find the following in org.eclipse.jdt.ui:

```
<extension
      point="org.eclipse.ui.popupMenus">
      <viewerContribution
            id="org.eclipse.jdt.ui.CompilationUnitEditorPopupActions"
            targetID="#CompilationUnitRulerContext">
```

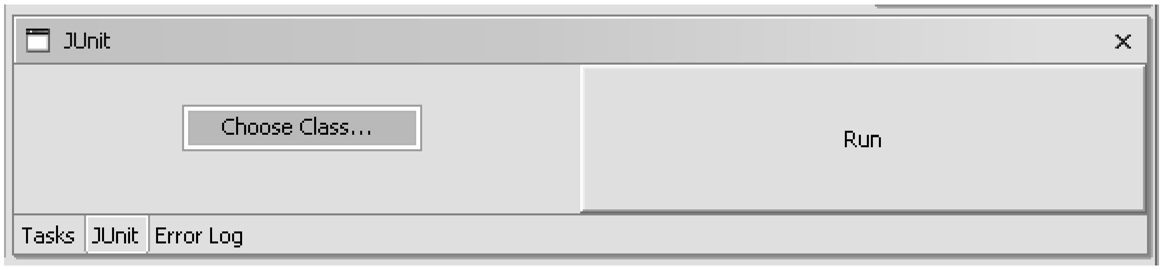
->unhappy example – it violates the naming conventions for targets, we should have another example, see the contribution article

```
            <action
                  id="org.eclipse.ui.texteditor.TaskRulerAction"
                  menubarPath="add"
                  label="%AddTask.label"
                  tooltip="%AddTask.tooltip"
         helpContextId="org.eclipse.ui.AddTask_action_context"
                  class="org.eclipse.ui.texteditor.TaskRulerAction">
            </action>
      </viewerContribution>
</extension>
```

We want a menu item called "Choose Class…". Copying the structure above (ignoring the bits we aren't going to use now), we get:

```
<extension point="org.eclipse.ui.popupMenus">
      <viewerContribution
            id="org.eclipse.contribution.junit.ui.view.items"
            targetID="org.eclipse.contribution.junit.ui.view">
            <action id="org.eclipse.contribution.junit.ui.view.ChooseClass"
                  menubarPath="add"
                  label="Choose Class..."
                  class="org.eclipse.contribution.junit.ui.ChooseClass">
            </action>
      </viewerContribution>
</extension>
```

The targetID should be the same fully qualified name as the id of the view declared a bit earlier in the manifest. When we call registerContextMenu(), Eclipse searches for extensions of popupMenus whose targetID matches the id of the view for which we are registering the menu. At this point we can pop up the menu.



Eclipse politely tells us that we need to define ChooseClass. The online help suggests that we make sure ChooseClass implements IViewActionDelegate. After defining the class, we can use the context menu item Source>>Override Methods… to create stubs for the methods in IViewActionDelegate:

```
public class ChooseClass implements IViewActionDelegate {

	public void init(IViewPart view) {
	}

	public void run(IAction action) {
	}

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

}
```

Now we can choose our menu item and nothing happens. How will we let the user choose a type? Where in the system are types choosen? JavaUI provides us with a handy set of utilities for Java-related, well, user interfaces. Searching for users of JavaUI.createTypeDialog() gives us a handful of uses. GotoTypeAction seems worth copying:

```
Shell shell= JavaPlugin.getActiveWorkbenchShell();
SelectionDialog dialog= null;
try {
	dialog= JavaUI.createTypeDialog(shell, new ProgressMonitorDialog(shell),
		SearchEngine.createWorkspaceScope(),
IJavaElementSearchConstants.CONSIDER_TYPES, false);
	…
Object[] types= dialog.getResult();
if (types != null && types.length > 0) {
	gotoType((IType) types[0]);
}
```

To ask the user for a type, we:

- Create the dialog.

- Fill in some parameters.
- Open it.
- Make sure it wasn’t cancelled.
- Make sure something was selected.
- Call our code.

This results in:

```
public void run(IAction action) {
	Shell shell= JavaPlugin.getActiveWorkbenchShell();
	SelectionDialog dialog= null;
	try {
		dialog= JavaUI.createTypeDialog(shell, new
ProgressMonitorDialog(shell),
			SearchEngine.createWorkspaceScope(),
IJavaElementSearchConstants.CONSIDER_TYPES, false);
	} catch (JavaModelException e) {
		// TODO: Better error handling
	}
	dialog.setTitle("Choose Test Class");
	dialog.setMessage("Class to Run");
	if (dialog.open() == IDialogConstants.CANCEL_ID)
		return;
	Object[] types= dialog.getResult();
	if (types == null || types.length == 0)
		return;
	JUnitPlugin.getDefault().runTest((IType) types[0]);
}
```

It runs. When the menu item is selected, the action is run. The JUnit plug-in is told to run the test. After the tests run our view is told that the tests ran, and the label turns green (if we’re lucky.)

The next thing we notice is that when we bring the workbench up the next time, our carefully selected test class is gone. This is impolite. How can we save view state? We’ll see in the next chapter. Reviewing this chapter, we saw:

- How to dynamically create menus using the MenuManager.
- How to invite contributions to dynamically created menus using registerContextMenu.
- How to implement a menu item using IViewActionDelegate.
- How to ask the user for a Java type using JavaUI.createTypeDialog().

Forward Pointers

- Other progress monitors? Avoiding separate dialog and showing the progressbar in-line in the dialog/workbenchwindo.

- Search engine scope

Editors? What's an example? View vs. editor.

View auxiliary to editors. Help navigate show properties etc.

Memories

Another of the Eclipse design rules is that you always split a plug-in into a core part and a user interface part. You'll see this in org.eclipse.jdt.ui vs. org.eclipse.jdt.core. The core plug-in provides all the functionality for representing and manipulating Java programs. The UI plug-in builds on that functionality to present programs and offer manipulations to a user. This is the Strata Rule:

Strata Rule: Separate the presentation logic from the model logic

The Strata Rule comes into play mostly strongly in two circumstances:

- When we want to support headless operation
- When we want multiple teams to work on related functionality

The split between jdt.core and jdt.ui helps folks who want to run the Java compiler programmatically. When Ant compiles Java code, it only needs to invoke the compiler, not the user interface.

Plug-in boundaries make good team boundaries. Sharing the manifest file is painful. It is difficult to merge two streams of changes to the XML. If two teams are going to work loosly coupled, then, they each need their own manifest. Since there is one manifest per plug-in, splitting the plug-in makes perfect sense.

Saved By The Bell

If the JUnitView is visible when we close Eclipse, the JUnitView will be visible when we re-open Eclipse. Unfortunately, though, the test class we selected is no longer visible. What can we do so the same test is visible? In this chapter we'll see how to:

- Save user interface state
- Restore user interface state

Ours is not the first view to have to solve this problem, of course. Every view needs to save and restore state to maintain the illusion of continuity for the user.

User Continuity Rule: Preserve user interface state across sessions

Every ViewPart has the opportunity to override to methods to save and restore state. The saving and restoring methods have a parameter, a simple database responding to the IMemento protocol. Mementos store key-value pairs of various types.

Sidebar: Serialization. Describe that Eclipse doesn't use object serialization to persist object state, too fragile etc. Persist state either in XML or simple properties. See DialogSettings.

For example, the ResourceNavigator saves a bunch of state, including the scroll position.

```
public void saveState(IMemento memento) {
        …
        //save vertical position
        ScrollBar bar = tree.getVerticalBar();
        int position = bar != null ? bar.getSelection() : 0;
```

```
            memento.putString(TAG_VERTICAL_POSITION, String.valueOf(position));
            …
    }
```

When restored, the ResourceNavigator reads the scroll position and moves the scroll bar:

```
    protected void restoreState(IMemento memento) {
            …
            //restore vertical position
            ScrollBar bar = tree.getVerticalBar();
            if (bar != null) {
                    try {
                            String posStr = memento.getString(TAG_VERTICAL_POSITION);
                            int position;
                            position = new Integer(posStr).intValue();
                            bar.setSelection(position);
                    } catch (NumberFormatException e) {
                    }
            }
            …
    }
```

Following this pattern, when we save the state, we should save the fully qualified name of the test, and when we restore the state we should turn the saved string back into an IType and set it in the label.

The relationship of these methods seems to be governed by the Symmetry Rule:

Symmetry Rule: Make concepts symmetrical

First, let’s make sure save and restore work as expected. We can implement saveState() to store the test’s name in the memento:

```
            public void saveState(IMemento state) {
                    System.out.println("saving");
                    state.putString("test", fTest.getText());
            }
```

When we close the runtime workbench, “saving” is printed on the console of the host workbench. Now we can do check restoreState():

```
            protected void restoreState(IMemento state) {
                    System.out.println("restoring: "+state.getString("test"));
            }
```

When we bring up the runtime workbench, though, nothing is printed. What is going on? It turns out that Eclipse violates the symmetry rule. SaveState() is called by the framework, but you have to create and call restoreState() yourself.

The memento used to save state is passed to the view when it is initialized, and we must store it:

```
            IMemento fState;
            public void init(IViewSite site, IMemento memento) throws PartInitException {
                    super.init(site, memento);
                    fState= memento;
            }
```

<<should explain the PartInitException and what happens when you fire it

We can't restore the state until the widgets are created:

```
public void createPartControl(Composite parent) {
	…
	restoreState(fState);
}
```

Now the console prints out "restoring: junit.samples.VectorTest" as expected.

This violation of the symmetry rule is probably a mistake. It causes us to add a field, fState, which is only useful for part of the life of the JUnitView. FState is only there to communicate between the init() method and createPartControl(), after which it is no longer interesting. A better API would be for the framework to remember the memento itself, and call restoreState() after calling createPartControl().

The rational behind this asymmetry is that you want to access the persisted state when creating your widgets. However, the framework doesn't know about your internal widget structure. Having a symmetrical restoreState call after createPartControl is called isn't useful since the widgets are already created and you would have to change them again which could result in flicker etc. A compromise solution would have been to pass in the memento in the createPartControl method. This avoids the client having to temporarily park the memento in an instance variable and retrieve it later when creating the widgets.

All that's left is mapping from the string "junit.samples.VectorTest" to the instance of IType. When mapping from a string to a type, Eclipse requires a bit more information than just the name. JavaCore.create(String) says that the parameter should be generated by IJavaElement.getHandleIdentifier(). When saving, then, we have to save this string and not the fully qualified name:

```
public void saveState(IMemento state) {
	state.putString("test", fTestClass.getHandleIdentifier());
}
```

When we restore, we set the view variable fTestClass and the text of the label based on what was stored in the state:

```
protected void restoreState(IMemento state) {
	try {
		IType test= (IType) JavaCore.create(state.getString("test"));
		fTestClass= test;
		fTest.setText(fTestClass.getFullyQualifiedName());
	} catch (Exception e) {
		e.printStackTrace();
	}
}
```

The error handling there is clearly wrong, but it's good enough for the end of the chapter. We'll see in the next chapter how to handle errors "The Eclipse Way". In the meantime, reviewing:

- We overrode saveState() to place our view's state in an IMemento

- We couldn’t just override a restoreState(), so we saved the memento when initializing the view
- We restored the state stored in the memento after we created the widgets for our view

Forward Pointers

- Persistent session state -> Plugin.getStateLocation
- Save state when no widget has been created

Life on the Breadcrumb Trail

We began to see the possibility of problems in the last chapter. How do we handle problems in a way consistent to the Eclipse rules? In this chapter we will add error handling to our JUnit user interface.

Before we dive into the details of implementing error logging, let’s take a moment to examine the purposes of error handling. We would like the customer’s experience of our contributions to be absolutely error free. As people who have written more than one line of code, though, we know that problems are inevitable. However, when bad things happen, they shouldn’t happen right in the user’s face. In Eclipse, this takes the form of of the Quiet Failure Rule.

> Quiet Failure Rule: The user should only be presented with information they can act on

On the other hand, as contributors we need to see lots of information—stack traces, invocation details, and so on—to diagnose problems. This suggests that we need to separate the end user’s experience of a fault from the programmer’s experience of the same fault.

What are the rules for ErrorDialog and logging?

Separating the way a user sees a fault and the way a programmer sees the same fault isn’t just an academic distinction. Imagine your contribution is enormously successful. You are going to have thousands or (eventually) millions of users. There is no way you can have an extended conversation each time a fault occurs. The economics would just kill you. You need a simple way to get more information from your customer.

Eclipse keeps a log of unusual events. When such an event occurs in our contribution, we need to add an entry to the log. We can search for users of Status for examples. Here’s one from AntCorePreferences:

```
protected void addToolsJar(List destination) {
        …
        } catch (MalformedURLException e) {
                // if the URL does not have a valid format, just log and ignore
                IStatus status = new Status(IStatus.ERROR,
AntCorePlugin.PI_ANTCORE, ERROR_MALFORMED_URL,
Policy.bind("exception.malformedURL"), e);
                AntCorePlugin.getPlugin().getLog().log(status);
        }
}
```

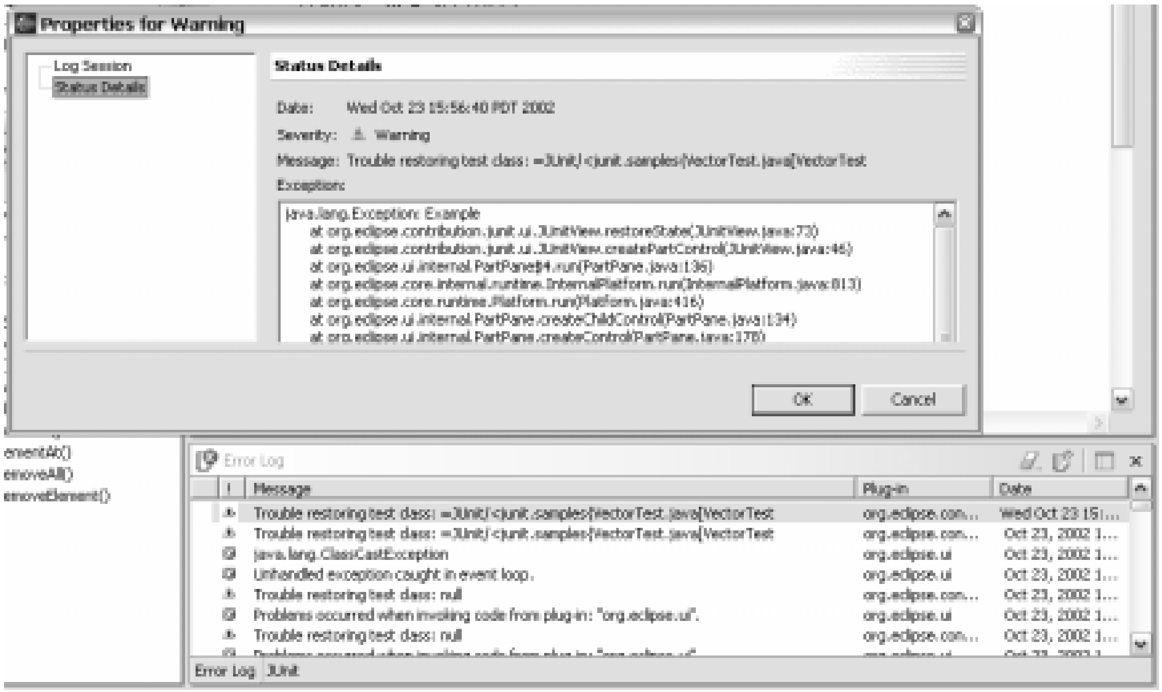
Each plug-in has access to the log. The entries in the log conform to IStatus, Status for example. A Status is built with a severity and the original exception (plus some other information.) Looking at our restoreState() code from the previous chapter, we can add a log entry if we have trouble restoring the test class:

```
protected void restoreState(IMemento state) {
        try {
                …
        } catch (Exception e) {
            String id= JUnitPlugin.getDefault().getDescriptor().getUniqueIdentifier();
            int code= 0;
            String message= "Trouble restoring test class: " + state.getString("test");
            IStatus status = new Status(IStatus.WARNING, id, code, message, e);
            JUnitPlugin.getDefault().getLog().log(status);
        }
    }
```

With this code in place, we can induce an error to see what happens:

```
    protected void restoreState(IMemento state) {
        try {
            …
            throw new Exception("Example");
        } catch (Exception e) {
            …
        }
    }
```

And then, by opening the Error Log (Window->Show View…->Other…->PDE Runtime->ErrorLog), we see a log entry:



Status Details

Date: Wed Oct 23 15:56:40 PDT 2002

Severity: Warning

Message: Trouble restoring test class: =JUnit/<junit.samples{VectorTest.java[VectorTest

Exception:

```
java.lang.Exception: Example
    at org.eclipse.contribution.junit.ui.JUnitView.restoreState(JUnitView.java:73)
    at org.eclipse.contribution.junit.ui.JUnitView.createPartControl(JUnitView.java:46)
    at org.eclipse.ui.internal.PartPane$4.run(PartPane.java:136)
    at org.eclipse.core.internal.runtime.InternalPlatform.run(InternalPlatform.java:813)
    at org.eclipse.core.runtime.Platform.run(Platform.java:416)
    at org.eclipse.ui.internal.PartPane.createChildControl(PartPane.java:134)
    at org.eclipse.ui.internal.PartPane.createControl(PartPane.java:178)
```

!	Message	Plug-in	Date
⚠	Trouble restoring test class: =JUnit/<junit.samples{VectorTest.java[VectorTest	org.eclipse.con…	Wed Oct 23 15:…
⚠	Trouble restoring test class: =JUnit/<junit.samples{VectorTest.java[VectorTest	org.eclipse.con…	Oct 23, 2002 1…
⊠	java.lang.ClassCastException	org.eclipse.ui	Oct 23, 2002 1…
⊠	Unhandled exception caught in event loop.	org.eclipse.ui	Oct 23, 2002 1…
⚠	Trouble restoring test class: null	org.eclipse.con…	Oct 23, 2002 1…
⊠	Problems occurred when invoking code from plug-in: "org.eclipse.ui".	org.eclipse.ui	Oct 23, 2002 1…
⚠	Trouble restoring test class: null	org.eclipse.con…	Oct 23, 2002 1…

Another nice feature to add to our JUnit UI would be to have the tests run automatically whenever a source file changes. First, though, we'll review:

- We found the log attached to our plug-in class
- We added a Status to the log describing what went wrong

Forward Pointers

Status/multistatus

ErrorDialog

SafeRunnable

Chacun à son goût

French for “to each his own”

Now that we finished logging problems, we can move on to the next bit of functionality. Wouldn’t it be nice to get immediate feedback whenever a test failed, just as we get immediate feedback whenever a compile fails? Every time the code changes, the tests should run and display their results.

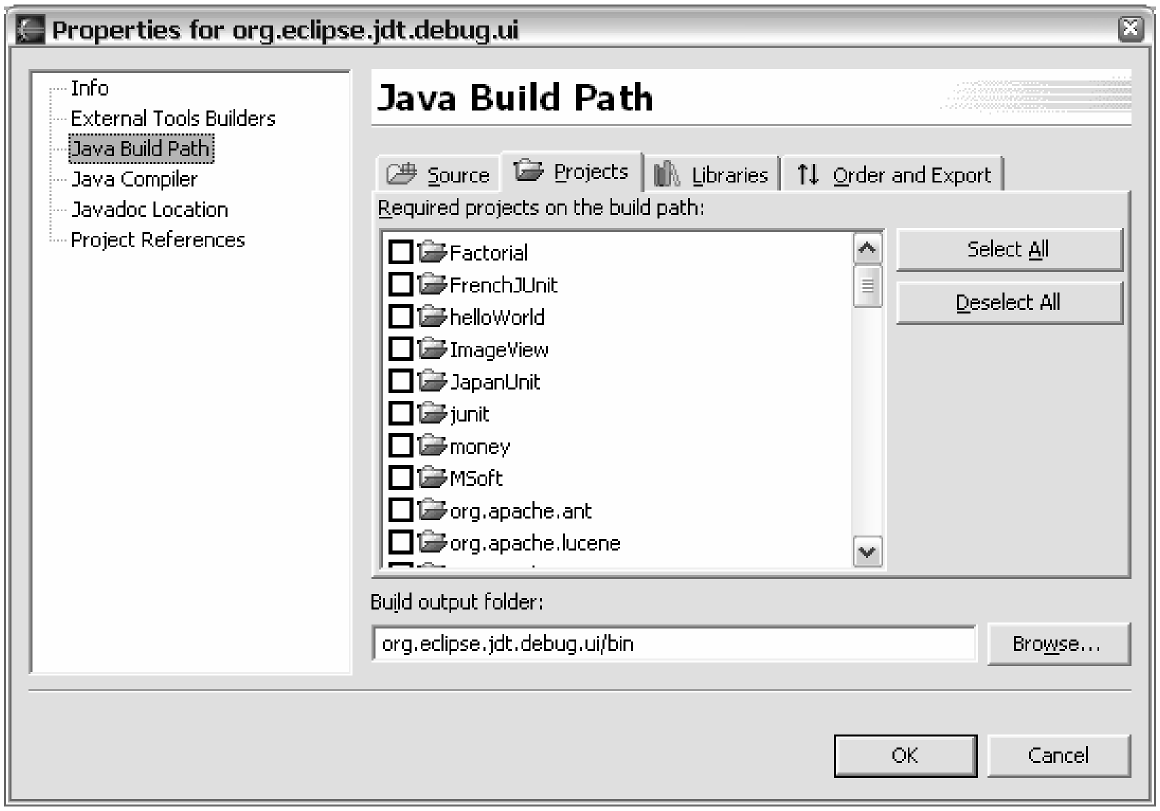
Is there a rule about configuration to express here?

We don’t want auto-testing for all projects. We would like to switch it off and on for individual projects (this will let us explore property pages, without which our chapter would have no topic.) In this chapter we will:

- Create a project property page with a checkbox for auto testing.
- Create stubs for the methods which get and set the property on the project.

In the next chapter we’ll worry about how to actually find and run the tests.

First, we need to get the property page to appear when we bring up the properties of a Java project. We can copy the declaration from one of the other Java-related property pages:



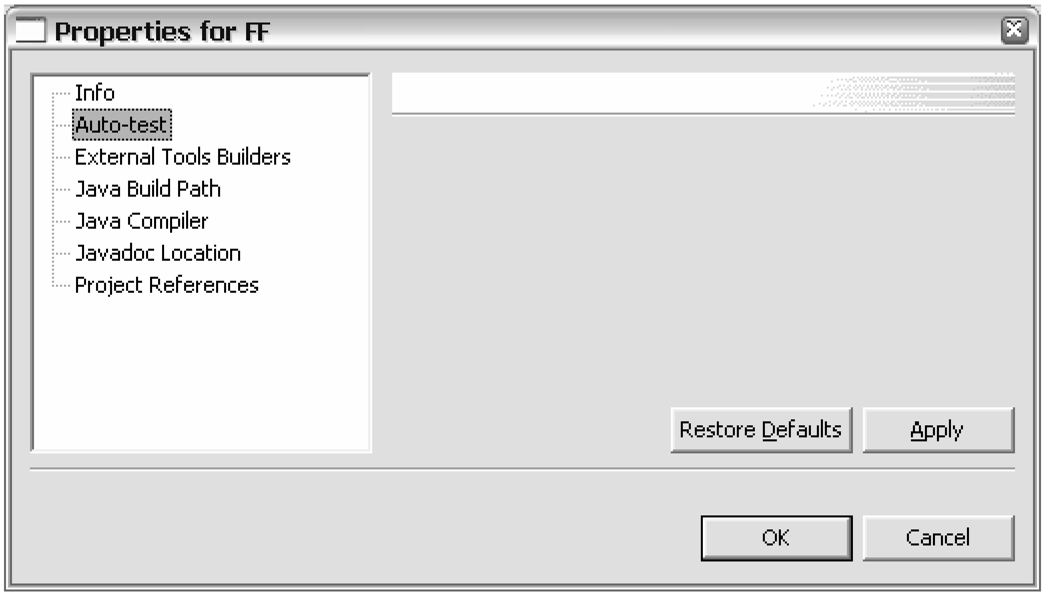
To find the implementation of this property page, we look in org.eclipse.jdt.ui. In the manifest, we find the extension that declares this page:

```
<extension point="org.eclipse.ui.propertyPages">
    <page
        name="%buildPathPageName"
        id="org.eclipse.jdt.ui.propertyPages.BuildPathsPropertyPage"
        objectClass="org.eclipse.core.resources.IProject"
        class="org.eclipse.jdt.internal.ui.preferences.BuildPathsPropertyPage">
        <filter name="nature" value="org.eclipse.jdt.core.javanature"/>
    </page>
</extension>
```

Copying this into our manifest, we get:

```
<extension point="org.eclipse.ui.propertyPages">
    <page
        name="Auto-test"
        id="org.eclipse.contribution.junit.autotestproperty"
        objectClass="org.eclipse.core.resources.IProject"
        class="org.eclipse.contribution.junit.ui.AutoTestPropertyPage">
        <filter name="nature" value="org.eclipse.jdt.core.javanature"/>
    </page>
</extension>
```

Running it, we see the name of the property page, but when we select it, we get an error because the class hasn't been implemented yet.



To implement the property page, we can look at our example. BuildPathsPropertyPage extends PropertyPage, so we probably should, too:

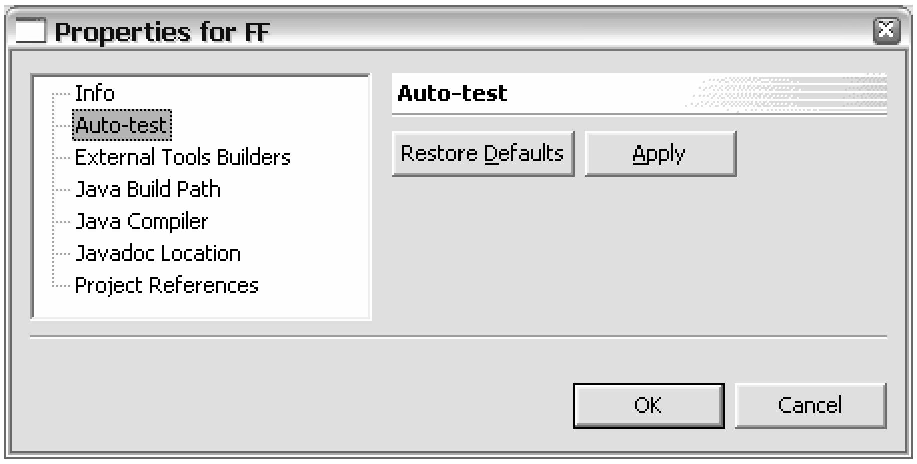
```
public class AutoTestPropertyPage extends PropertyPage {

    public AutoTestPropertyPage() {
        super();
    }

    protected Control createContents(Composite parent) {
        return null;
    }

}
```

Now when we select the auto-test property page, we get a blank page:



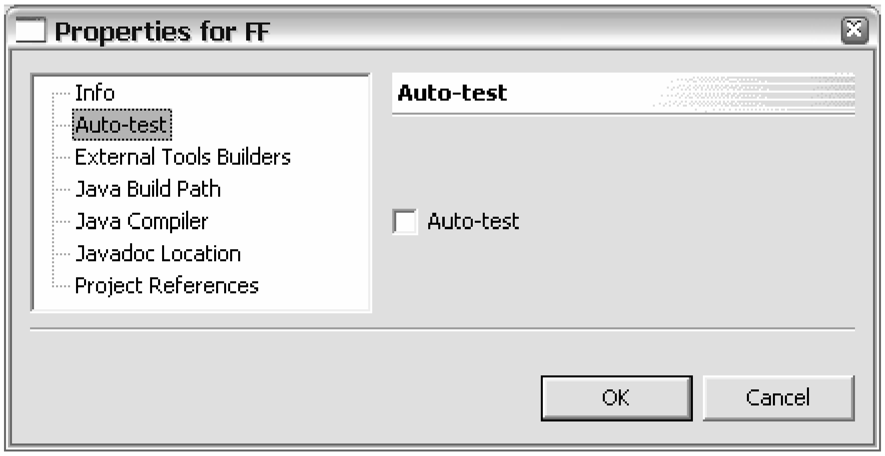
We don't want those ugly "Restore Defaults" and "Apply" buttons, so we add a line:

```
    protected Control createContents(Composite parent) {
        noDefaultAndApplyButton();
        return null;
    }
```

Just as we did with the JUnit view, we need to add controls to the page. In our case, we want to add a checkbox with the label "Auto-test". Looking at all the other implementors of createContents() (there are more than 100 of them), it looks like we only need to add a Button and return it as the value of the method:

```
    Button fAutoTest;
    protected Control createContents(Composite parent) {
        noDefaultAndApplyButton();
        Button fAutoTest= new Button(parent, SWT.CHECK);
        fAutoTest.setText("Auto-test");
        return fAutoTest;
    }
```

The result looks ugly, but we'll take up layout briefly in the next chapter.



When we create the check box, we should set its selection to true if auto-testing is already active. For now we will just define a stub:

```
protected Control createContents(Composite parent) {
	noDefaultAndApplyButton();
	fAutoTest= new Button(parent, SWT.CHECK);
	fAutoTest.setText("Auto-test");
	fAutoTest.setSelection(getIsAutoTesting());
	return fAutoTest;
}

private boolean getIsAutoTesting() {
	return false;
}
```

When the “OK” button is pressed, we need to set the property to the value of the checkbox. There are several similar methods we could override, but performOK() is the bottleneck through which all the messages flow.

```
public boolean performOk() {
	setIsAutoTesting(fAutoTest.getSelection());
	return true;
}

private void setIsAutoTesting(boolean on) {
}
```

Notice that we’re careful always to return true from performOk(), even if there is a problem. If we don’t, the user won’t be able to proceed if there is a problem.

In the next chapter we’ll figure out what to do to turn auto-testing on and off. For the moment, we can review properties:

- We declared a properties page in the manifest. By copying cleverly, we were even able to add it only to Java projects (those with the Java nature.)
- Added a checkbox to the page.
- Made changes when the “OK” button was pressed by overriding performOk().

A Time to Build

In the previous chapter, we left ourselves two stubs to fill in.

- How do we detect if auto-testing is turned on for a project?
- How do we turn auto-testing off and on?
- How do we declare a builder to actually run the tests?

The most obvious answer to these questions lies in persistent properties. Each resource has associated with it a number of properties which are saved and restored when Eclipse shuts down and restarts.

```
IResource
public String getPersistentProperty(QualifiedName key) throws CoreException;
public void setPersistentProperty(QualifiedName key, String value) throws
CoreException;
```

The problem with persistent properties from our perspective is that they are stored per workspace, they are not shared with the team. Once we have turned auto-testing on for a project, we would like it to stay on, no matter who is changing the project.

Each project has associated with it a set of builders. A Builder is an object which is informed any time a resource in the project changes. A complicated builder like the one that runs the Java compiler carefully examines the details of the changes to minimize the amount of recompilation. We don't need to be nearly that clever. We will just run all the tests on any change.

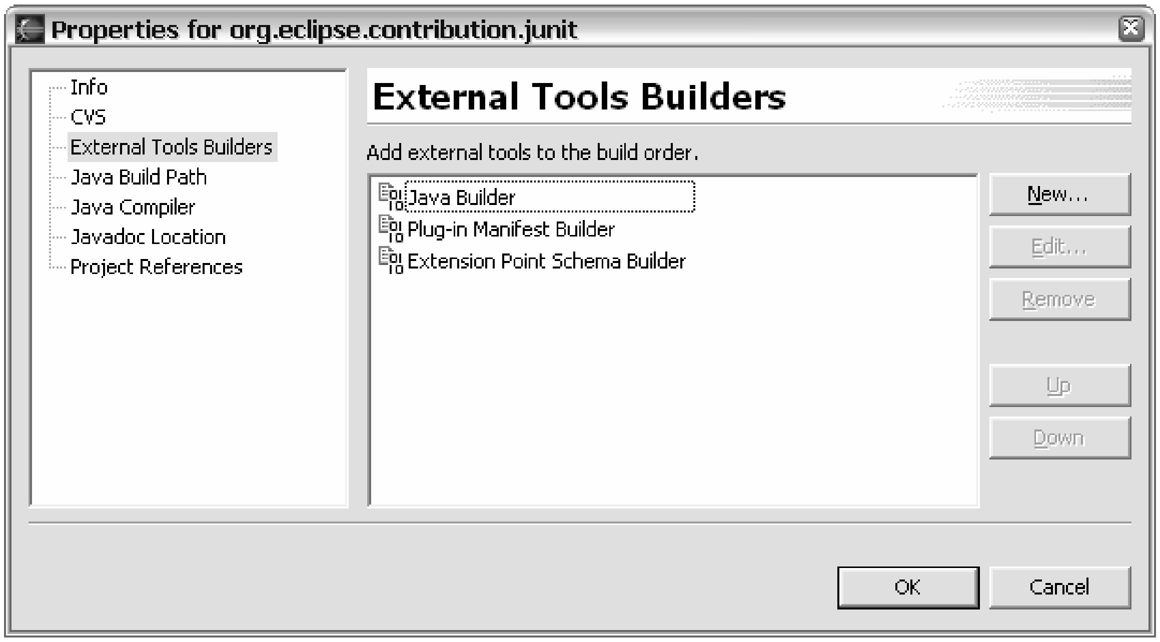
Re-examination in Design

There is actually an intellectually interesting problem hidden behind running the tests. The behavior we really want is Martin Fowler's mythical Automatic Bug Detector ™. If we write code that doesn't do what we mean, we would like the system to tell us. Since at the moment automated tests are the only way for us to objectively state what we mean, this implies that we run the tests and notify the user on error. How do we know which tests to run? A perfect answer to that question is the interesting problem.

What we start with are a bunch of tests and a bunch of changes to the code (we haven't talked about resource deltas yet, but we will soon.) In a perfect world, we would only run those tests whose answer could possibly have changed given the changes we have made. Since we don't know how to do this analysis, our heuristic solution is to run all the tests in the project. It will take too long, and there will be errors uncaught (for example, in dependent projects), but it's a lot better than no feedback at all. In practice, it approaches the ABD.

Design is often like this. You find a problem. A general solution seems expensive. By examining the nature of the problem you find a special case solution that is adequate and cheaper. But the art in design is never forgetting you have made that compromise, to being willing always to revisit your inevitable shortcuts. If the "adequate" solution isn't adequate, or it turns out not to be "cheap", you go back to the general problem instead of throwing good effort after bad.

Where are the builders stored with respect to the project? Here are the builders for a plug-in project, as seen in Eclipse:

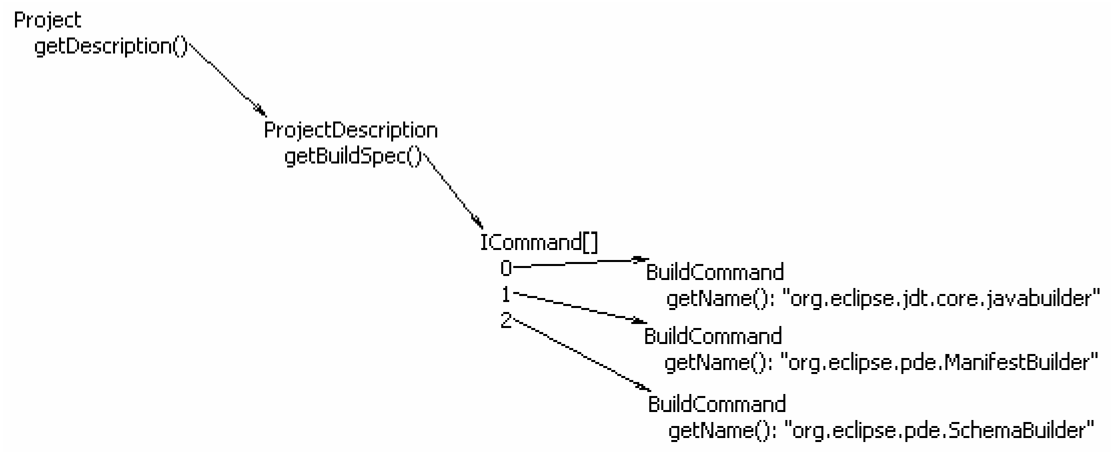


External Tools Builders

Add external tools to the build order.

- Java Builder
- Plug-in Manifest Builder
- Extension Point Schema Builder

The same builders are represented in Eclipse as an array of builder id's contained in the Project's ProjectDescription.


Project
getDescription()
ProjectDescription
getBuildSpec()
ICommand[]
0
1
2
BuildCommand
getName(): "org.eclipse.jdt.core.javabuilder"
BuildCommand
getName(): "org.eclipse.pde.ManifestBuilder"
BuildCommand
getName(): "org.eclipse.pde.SchemaBuilder"

This is a whole lot of indirection:

- From Project to ProjectDescription,
- From ProjectDescription to ICommand,
- From builder id in ICommand to a concrete builder class,
- From builder class to actual builder instance.

Whether all this indirection is really needed is out of scope for this discussion (we've talked about it a lot, and noticed that beer helps the discussion but prevents us remembering our conclusions.) All we really care about for the moment is that if auto-testing is turned on, a new builder should be installed in this array.

We can sense whether auto-testing is on by searching the array:

```
    private static final String BUILDER_ID= "org.eclipse.contribution.junit.ui.autotest";

    private ICommand getAutoTestingBuilder() throws CoreException {
        IProjectDescription description= getProject().getDescription();
        ICommand command= description.newCommand();
        command.setBuilderName(BUILDER_ID);
        return command;
    }

    private boolean getIsAutoTesting() throws CoreException {
        IProjectDescription description = getProject().getDescription();
        List commands= Arrays.asList(description.getBuildSpec());
        return ! commands.contains(getAutoTestingBuilder());
    }
```

To implement the setting of auto-testing, first we need a switch so if we are turning auto-testing on, we add the builder, and if we're turning auto-testing off, we remove it:

```
        private void setIsAutoTesting(boolean on) {
            try {
                if (on)
                    addBuilder();
                else
                    removeBuilder();
            } catch (CoreException e) {
            }
        }
```

It would be nice if the API for adding a builder was as simple as IProject.addBuilder(IBuilder) or some such, but it isn't. We have to manipulate the array ourselves. By converting it to a List, we can take advantage of the List protocol to simplify our code:

```
    private void addBuilder() throws CoreException {
        IProjectDescription description= getProject().getDescription();
        List commands= Arrays.asList(description.getBuildSpec());
        if (! getIsAutoTesting())
            commands.add(commands.size(), getAutoTestingBuilder());
        desc.setBuildSpec((ICommand[]) commands.toArray(new ICommand[0]));
        getProject().setDescription(description, null);
    }
```

Removing the builder works analogously—if the builder is present, we remove it:

```
    private void removeBuilder() throws CoreException {
        IProjectDescription description = getProject().getDescription();
        List commands= Arrays.asList(description.getBuildSpec());
        if (getIsAutoTesting())
```

```
                commands.remove(getAutoTestingBuilder());
            desc.setBuildSpec((ICommand[]) commands.toArray(new ICommand[0]));
            getProject().setDescription(description, null);
    }
```

We get an error in the error log every time we click “OK” with auto-test on, because we don’t have a builder class yet. First, we need to declare the class in the manifest.

```
  <extension point="org.eclipse.core.resources.builders"
      id="autotest"
      name="Auto Test">
    <builder>
      <run class="org.eclipse.contribution.junit.ui.AutoTestBuilder">
      </run>
    </builder>
  </extension>
```

Not that the id of the extension is not fully qualified. At runtime, the plug-in id will be prepended to the id in the manifest. This bites us from time to time, as the policy is not consistent. You have to look carefully in the documentation to see which ids are absolute and which are plug-in relative.

The class itself needs to subclass IncrementalProjectBuilder. We’ll stub the implementation just to see when our builder gets called:

```
    public class AutoTestBuilder extends IncrementalProjectBuilder {
            public AutoTestBuilder() {
            }

            protected IProject [] build(int kind, Map args, IProgressMonitor monitor)
    throws CoreException {
                System.out.println("building");
                return null;
            }
    }
```

All that’s left is actually running the tests when the AutoTestBuilder is told to build(). For now we can review:

- We read the list of installed builders for a project to see if auto-test was turned on. If the auto-test builder was present, we told the user auto-testing was on.
- We installed or removed the auto-test builder when the user turned auto-testing on or off.
- We created a stub subclass of IncrementalProjectBuilder where we could later actually run the tests for the project.

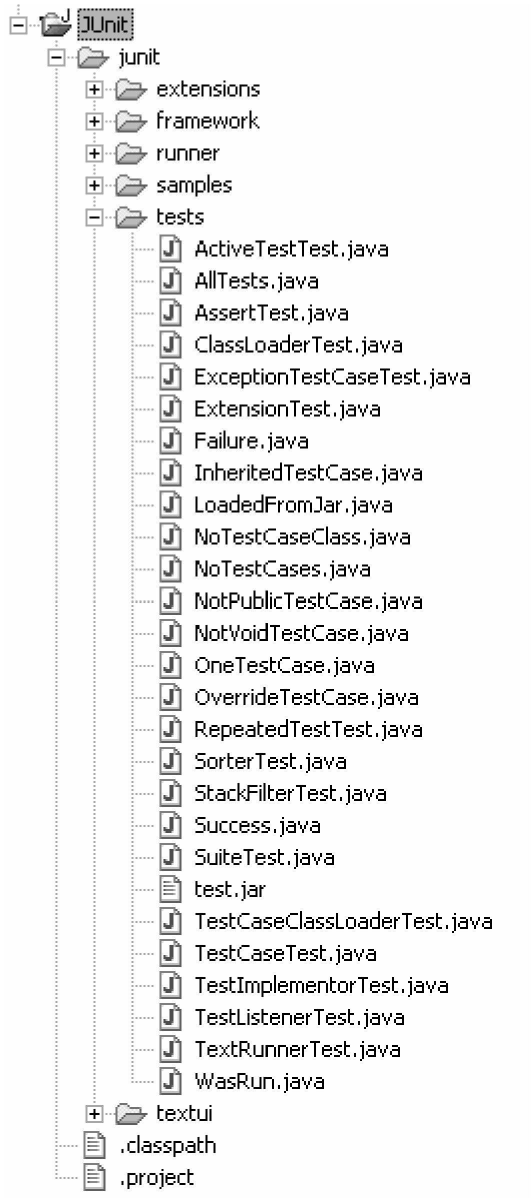
Bringing in the Tests

What's the best way to do this?

Where were we. Ah, yes, searching for tests. We want a function that given a resource, returns a JUnit TestSuite containing all of its test cases. For purposes of this chapter we are going to use a simple criterion for deciding whether something "is-a" test—any class that subclasses junit.framework.TestCase.

If we want to run all the tests automatically, we need to be able to find them. The goal of this chapter is to traverse the project, adding to a TestSuite whenever we find a subclass of TestCase.

Here is a typical project:



```
JUnit
  junit
    extensions
    framework
    runner
    samples
    tests
      ActiveTestTest.java
      AllTests.java
      AssertTest.java
      ClassLoaderTest.java
      ExceptionTestCaseTest.java
      ExtensionTest.java
      Failure.java
      InheritedTestCase.java
      LoadedFromJar.java
      NoTestCaseClass.java
      NoTestCases.java
      NotPublicTestCase.java
      NotVoidTestCase.java
      OneTestCase.java
      OverrideTestCase.java
      RepeatedTestTest.java
      SorterTest.java
      StackFilterTest.java
      Success.java
      SuiteTest.java
      test.jar
      TestCaseClassLoaderTest.java
      TestCaseTest.java
      TestImplementorTest.java
      TestListenerTest.java
      TextRunnerTest.java
      WasRun.java
    textui
  .classpath
  .project
```

Here are some of the objects representing this project in Eclipse:

Ch-ch-ch-ch-changes

Resource listener to change color back to background color if any Java source changes. Would use a JavaElementListener since we are only interested in changes to Java Source, see the org.eclipse.jdt.junit.plugin the DirtyListener in TestRunnerViewPart.

Looking Good

Managing images, happy unhappy face in the label.

AbstractUIPlugin->ImageRegistry.

Convention for images: icons folder in the plugin directory

Eclipse corner article on managing images

Reputation Building

JUnit builder, run tests after every compile. FP to incremental build functionality.

Get at project description and register an additional builder. See Antlr plugin

(see the community page on eclipse corner

Eclipse corner article on builders

Listening Louder

Write some unit tests for plug-ins. (Maybe just a sidebar?)

Needs the option org.eclipse.pde.junit plugin

Sidebar: Templates

The Big Picture

Test outline/editor

Prime Time

Productization (plug-in -> contribution)

Over There

Internationalization.

Focus plugin externalization (plugin.properties)

Sidebar (JDT support for string externalization)

Fragments (3rd iteration?)

use 2.0 nl fragments as an example

I Need Somebody

Help

- Help contents
  - Extension point for defining contents
  - Make junit documentation available
  - See the Help Documentation example
- Context help
  - HelpContextID
  - Help plugin for jdt.ui: org.eclipse.jdt.doc.user_2.1.0

Posterity

Documenting extension points

And Now…

Feature

Distributing with an update site

Circle Three

Core vs UI

Cook’s tour of:

- SWT
    - Shell
    - Thread rule
- JFace
    - Use JFace to implement a View
- Workbench
    - Workbench adapter
- Workspace (core)
    - IStartup
    - Resources
        - Handles
        - Resource Delta
        - Workspace runnables
        - Projects/natures
        - Folders
        - Files
    - Markers
    - Builders

JDT

- AST
- JavaModel

Debug

- Launch Configuration

Compare

Team

Committer

Building a perspective

Place holder for view

Building an entire app

Branding

Evolution

Keeping the API stable

Splitting a plugin

How Eclipse structures the computing experience

Long running operations

?Cancelling test run – added to result view

not a good example for an operation -> it doesn't run in the same process but is a separate process

Appendix: Rules

Contribution Rule: Everything is a contribution
Lazy Loading Rule: Contributions are only loaded when they are needed
Sharing Rule: Add, don't replace
Conformance Rule: Contributions must conform to expected interfaces
Monkey See/Monkey Do Rule: Always start by copying the structure of a similar plug-in
Relevance Rule: Contribute only when you can successfully operate
Safe Platform Rule: As the provider of an extension point, you must protect yourself against misbehavior on the part of extenders.
Invitation Rule: Whenever possible, let others contribute to your contributions
Explicit Extension Rule: Declare explicitly where a platform can be extended
Diversity Rule: Extension points accept multiple extensions
Explicit API Rule: Separate the API from internals
Stability Rule: Once you invite someone to contribute, don't change the rules
Revelation Rule: Reveal the API a little at a time
Integration Rule: Integrate your contribution with existing contributions.
Strata Rule: Separate the presentation logic from the model logic
User Continuity Rule: Preserve user interface state across sessions
Symmetry Rule: Make concepts symmetrical