Lab Exercises

Eclipse Platform Enablement
D/3ECA
IBM Corporation – RTP, NC

Exercise 1 Using the Plug-in Development Environment

At this point the terms and concepts behind Eclipse plug-in development have been introduced. However, sometimes the best way to learn is by doing. In this laboratory exercise you will implement a Workbench action extension to contribute a menu item to the window menu bar that displays the message box, “Hello, Eclipse world.” While admittedly the final result is anticlimactic, it is definitely worth doing for the experience. You’ll have a chance to see how the different parts of the Plug-in Development Environment (PDE) work, and will also verify that your environment is correctly set up for your future plug-in development projects.

At the end of this exercise you should be able to:

- Create an XML manifest for a plug-in using the Plug-in Manifest Editor
- Write the Java code to be executed for the extension
- Test and debug your plug-in in the run-time Workbench

In case you missed something, here’s an ultra mini-review.

- A plug-in is an extension of the Eclipse Platform. It is a set of related files that implement some function and a manifest file, called `plugin.xml`, which describes the content of the plug-in and its configuration.
- A plug-in can contribute to the Workbench by declaring an extension of an existing Workbench extension point. The manifest file describes this contribution. A plug-in can also declare new extension points that other plug-ins may use. That will be covered when we discuss *Creating New Extension Points: How Others Can Extend Your Plug-in*.

Exercise Setup

The PDE Target Platform configuration adjustment made in this step ensures that all external plug-ins are visible. This configuration simplifies the plug-in development and is oriented towards new plug-in developers. Access the PDE Target Platform preferences using **Window > Preferences**. Expand **Plug-In Development** and select **Target Platform**. Select **Not In Workspace** to make all plug-ins visible, as shown in Figure 3.1. Select OK to close the dialog.

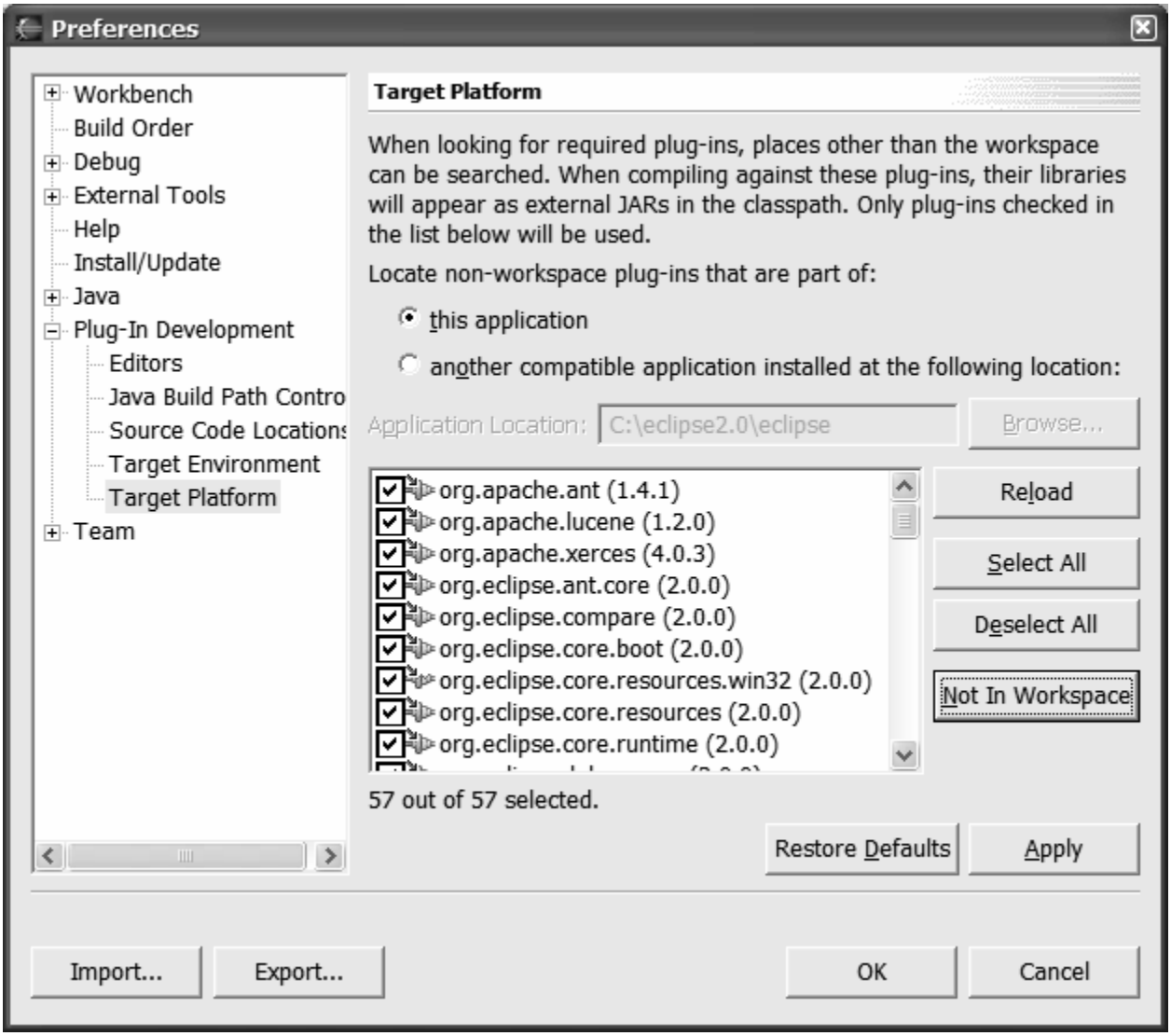


Figure 1-1
Making All External Plug-ins Visible

Note: The number of plug-ins that can be selected depends on your version (2.0 or 2.1) and build (Eclipse vs WebSphere Studio Workbench).

If you forget this step and continue creating a plug-in project using the PDE wizard, you will see the message shown in Figure 3.2. If you do see this message, it's no problem. Just select **OK** to accept its resolution and continue.

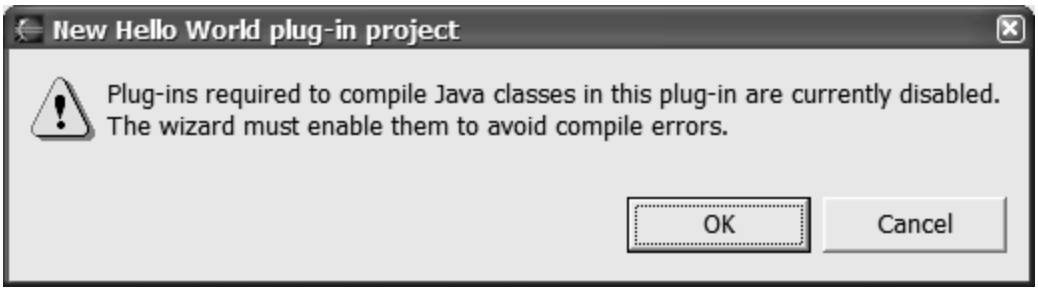


Figure 1-2
Wizard Warning Message

Exercise Instructions

You have a choice of how to complete this exercise.

- The first approach uses the PDE's plug-in project wizard to generate all the necessary code and the plug-in manifest details for an action extension.

  This first approach is presented in Section 1.
- The second approach will proceed step-by-step, showing all the dialogs and editors that you need to use when creating a plug-in with the PDE. This approach offers you the chance to use the PDE when the example is quite simple, so you can concentrate on using the tool, not the details of the coding at hand.

  If you prefer this second approach, turn to Section 2.

Reminder: Unlike some of the earlier exercises, this exercise has no template or solution associated with it. Once you have installed Eclipse and configured it as described above, you can start immediately.

Section 1: "Hello, World" in Five Minutes or Less

Eclipse should already be installed and open; you should have already completed the steps described in the "Exercise Setup" section.

1. Begin by creating a plug-in project using the New Plug-in Project wizard. Select **File > New > Project**. In the New Project dialog, select **Plug-in Development** and then **Plug-in Project** in the list of wizards, and then select **Next**. Name the project `com.ibm.lab.helloworld`. Accept the default workspace location and select **Next**. The PDE will create a plug-in id based on this name, so it must be unique in the system (by convention, the project name and the plug-in id are the same). Accept the default plug-in project structure and select **Next**. Select the **Hello, World** option, as shown in Figure 3.3, and then select **Next**.

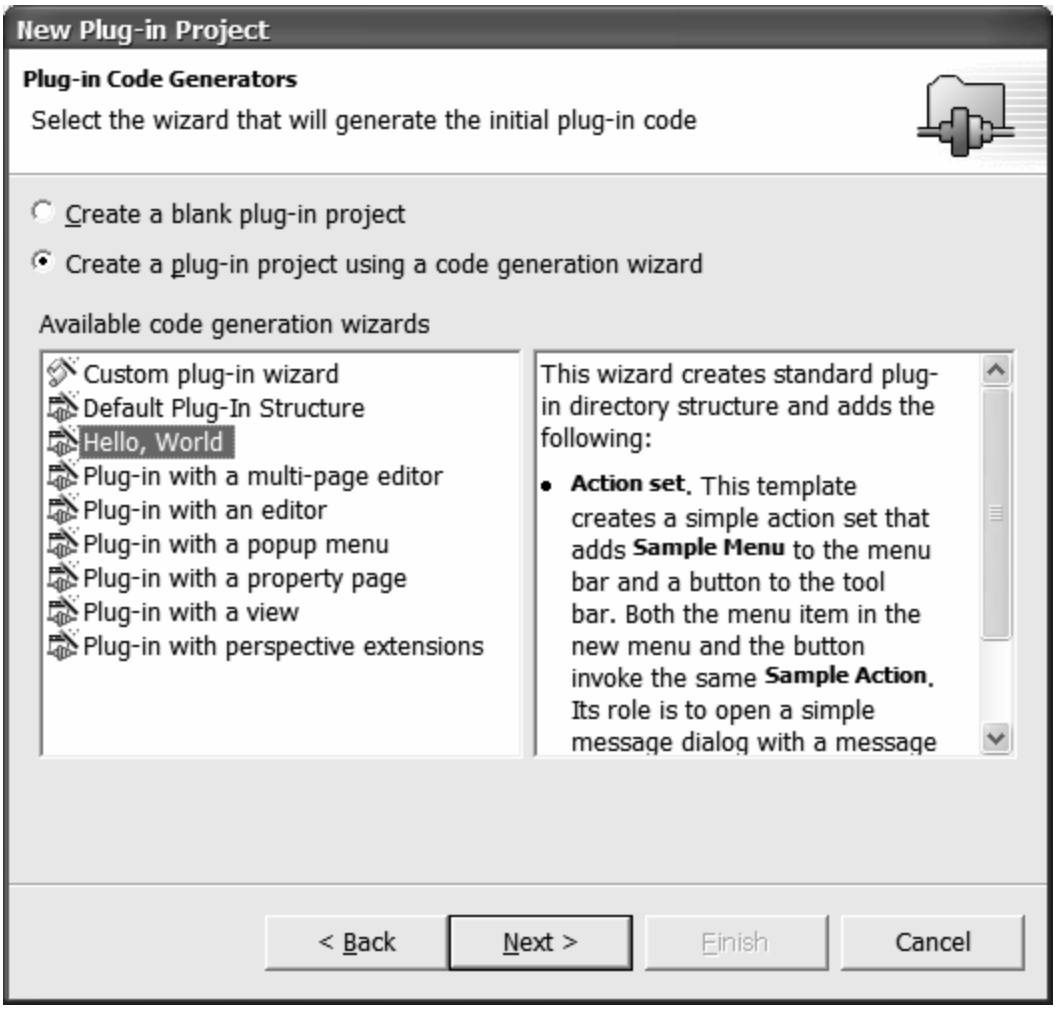


Figure 1-3
Hello, World Plug-in Code Generator

2. The proposed plug-in name and plug-in class name are based on the last word of the plug-in project, `com.ibm.lab.helloworld`. This example doesn't need any of the plug-in class convenience methods, so deselect all three options under **Plug-in code generation options**, as shown in Figure 3.4, and select **Next** (not **Finish**; you've got one more page to go).

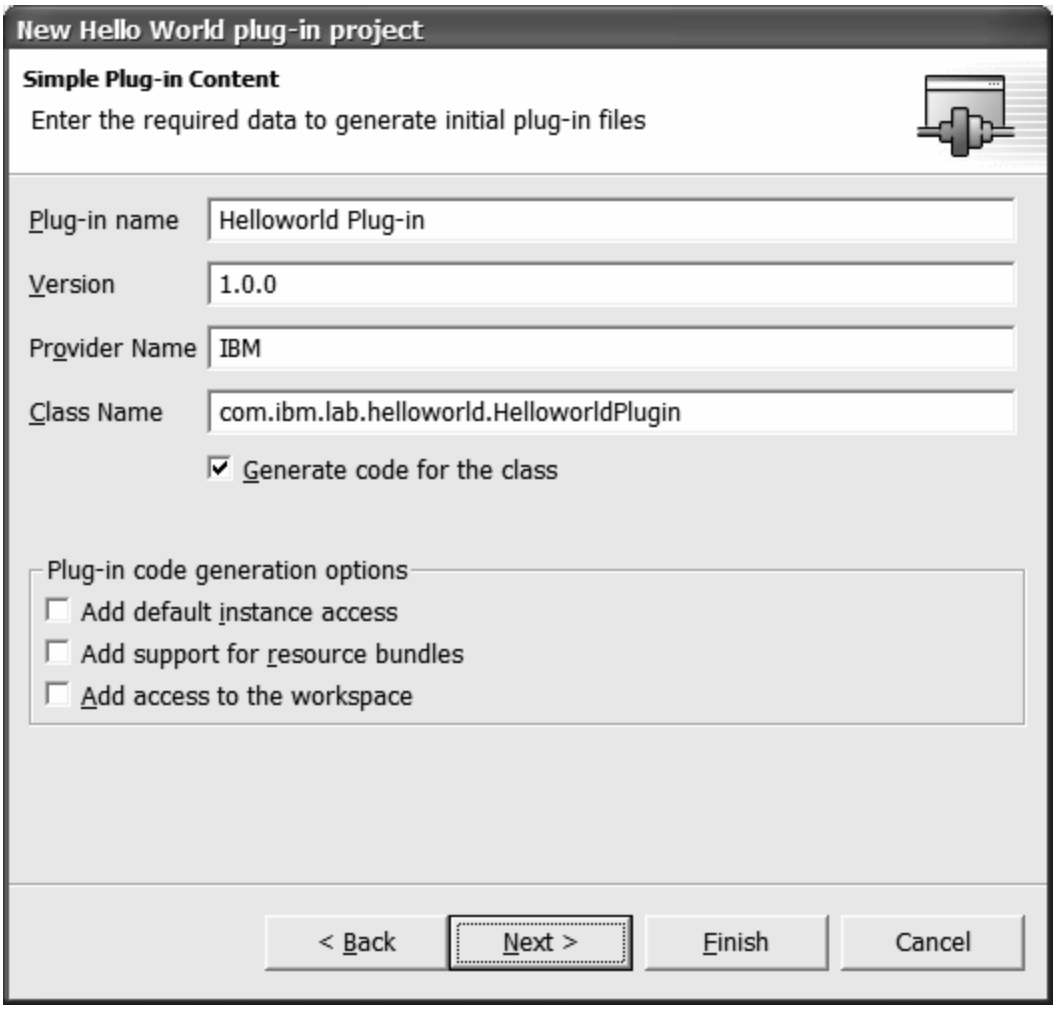


Figure 1-4
Hello, World's Simple Plug-in Content Wizard Page

3. The next page, shown in Figure 3.5, is where you can specify parameters that are unique to the "Hello, World" example, such as the message that will be displayed.

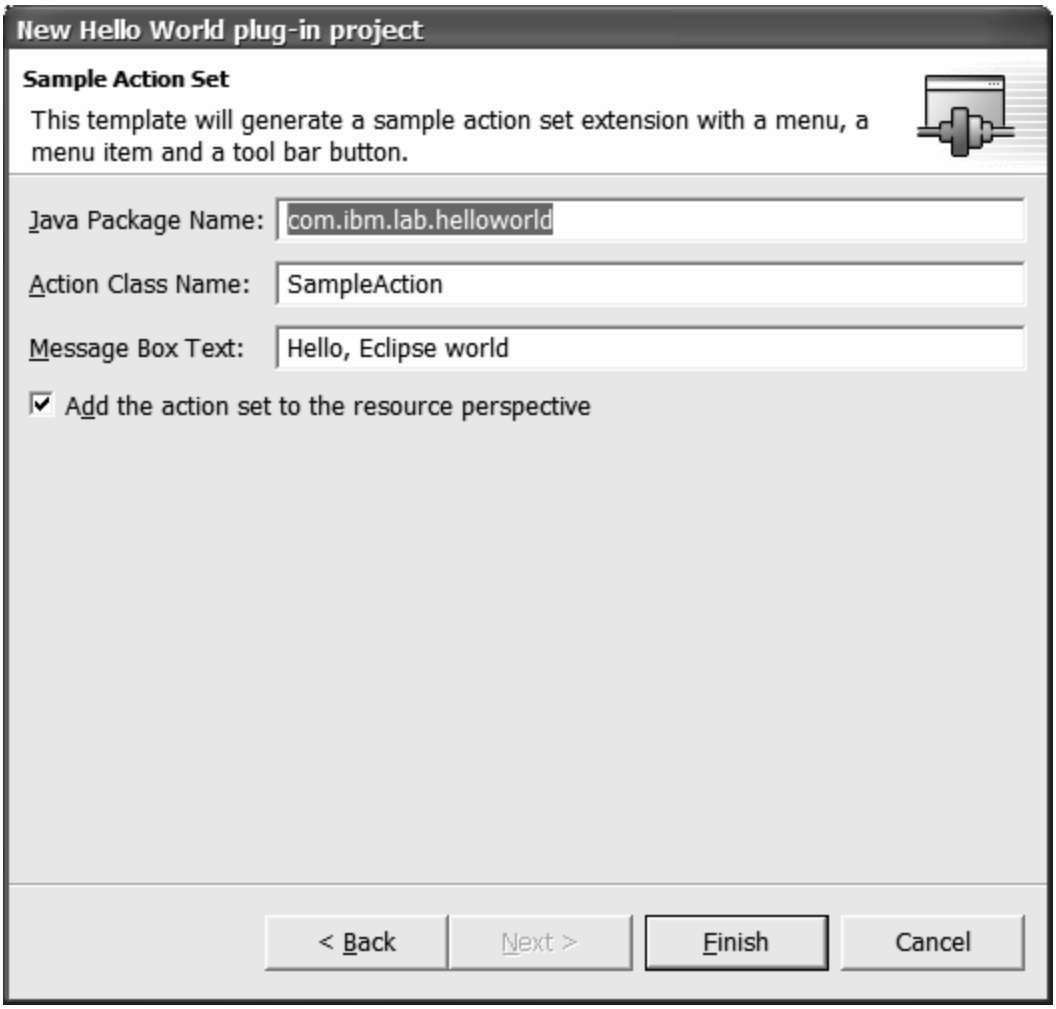


Figure 1-5
Hello, World Template Page

4. To simplify the resulting code, change the target package name for the action from `com.ibm.lab.helloworld.actions` to `com.ibm.lab.helloworld`, the same package as that of the plug-in class. While you might choose to have a separate package for grouping related classes in a real-world plug-in, in this case there will only be two classes (the plug-in class and the action), so let's put them together in the same package.
5. Select **Finish** and continue with Section 3, Testing with the Run-Time Workbench.

Section 2: "Hello, World" with Detailed Step-By-Step Instructions

This approach to creating your first plug-in focuses on how to use the PDE, and thus omits some of the smaller implementation details. Rest assured, the exercises to follow cover them. That being said, let's go!

Eclipse should already be installed and open; you should have already completed the steps described in the "Exercise Setup" section.

1. Begin by creating a plug-in project using the New Plug-in Project wizard. Select **File > New > Project**. In the New Project dialog, select **Plug-in Development** and **Plug-in Project** in the

list of wizards, and then select **Next**. Name the project `com.ibm.lab.helloworld`. Accept the default workspace location and select **Next**. The PDE will create a plug-in id based on this name, so it must be unique in the system (by convention, the project name and the plug-in id are the same). Accept the proposed plug-in project structure and select **Next**. Select the **Default Plug-In Structure** option, as shown in Figure 3.6, and then select **Next**.

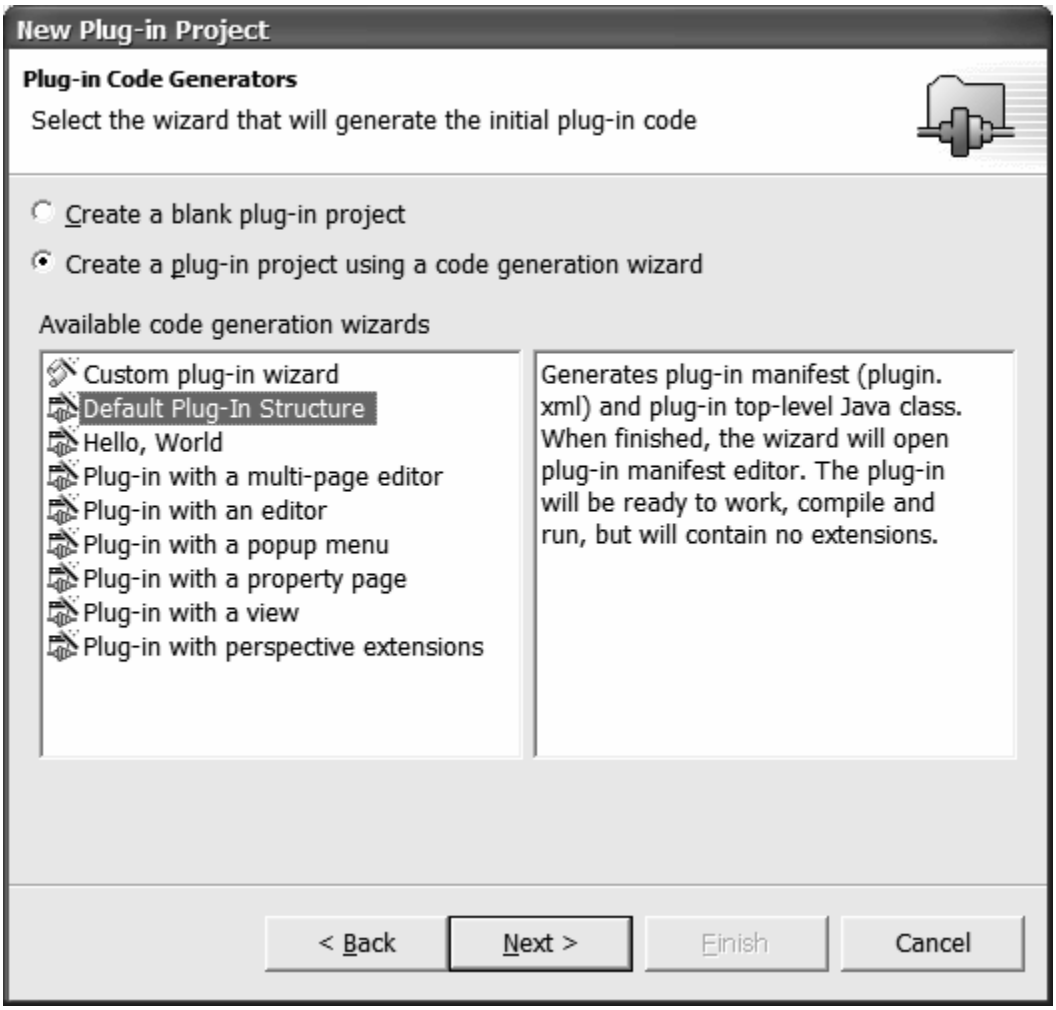


Figure 1-6
Default Plug-in Code Generator

2. The proposed plug-in name and plug-in class name are based on the last word of the plug-in project, `com.ibm.lab.helloworld`. This example doesn't need any of the plug-in class convenience methods, so deselect all three options under **Plug-in code generation options**, as shown in Figure 3.7, and select **Finish**.

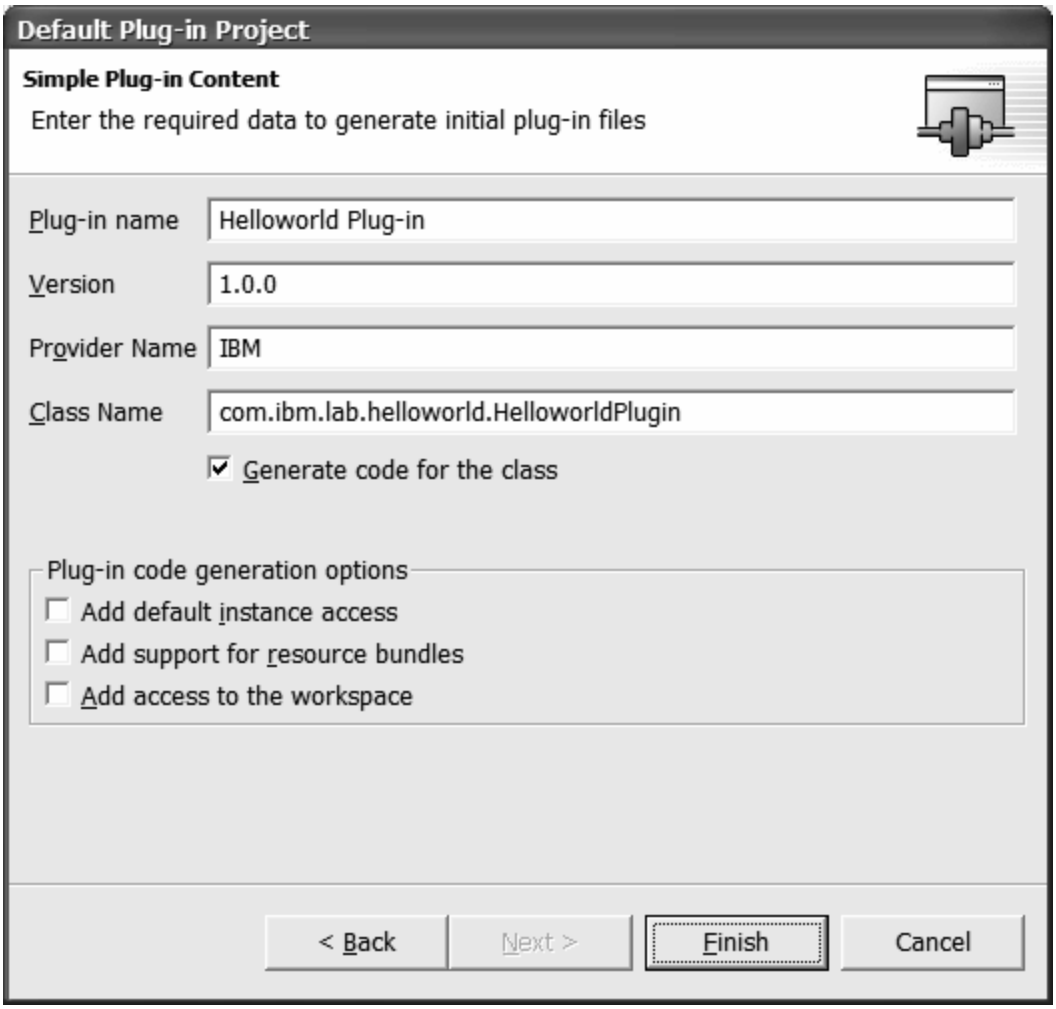


Figure 1-7
Simple Plug-in Content Wizard Page

Note: Although this example doesn't need a plug-in class, deselecting the **Generate code for the class** option without clearing the Class name field generates a value for the `class` attribute of the `<plugin>` tag. In this case, the plug-in definition would reference a plug-in class that was not generated, resulting in a run-time error, `Plug-in "com.ibm.lab.helloworld" activation failed while loading class "com.ibm.lab.helloworld.SampleAction"` in the Console. For the sake of simplicity, let the wizard generate the plug-in class.

The `plugin.xml` file generated should be automatically opened for editing using the Plug-in Manifest Editor in the PDE perspective after you select **Finish**, as shown Figure 3.8.

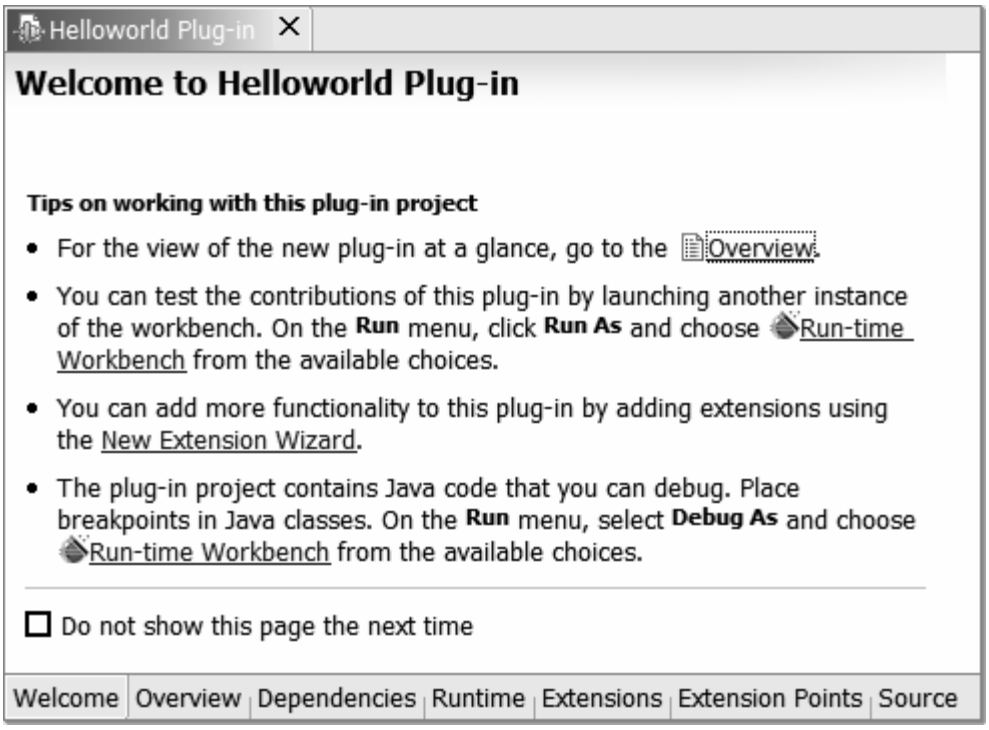


Figure 1-8
Welcome to `Helloworld` *Plug-in*

You won't see the **Welcome** page if you previously opened this project and selected **Do not show this page the next time**. In this case, you would see the **Overview** page instead.

3. Select the **Source** page. Verify that the generated `plugin.xml` content is as follows.

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
   id="com.ibm.lab.helloworld"
   name="Helloworld Plug-in"
   version="1.0.0"
   provider-name="IBM"
   class="com.ibm.lab.helloworld.HelloworldPlugin">

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

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

</plugin>
```

Note that the `<requires>` section states that the basic user interface and core services must be present for this plug-in to load successfully.

4. Select the **Extensions** page and select **Add…** to start the New Extension wizard (see Figure 3.9). Select **Generic Wizards** and **Schema-based Extension.** This wizard will lead you through the creation of an extension based on the extension point's schema definition, that is, based on its expected XML child tags and attributes. Select **Next**.

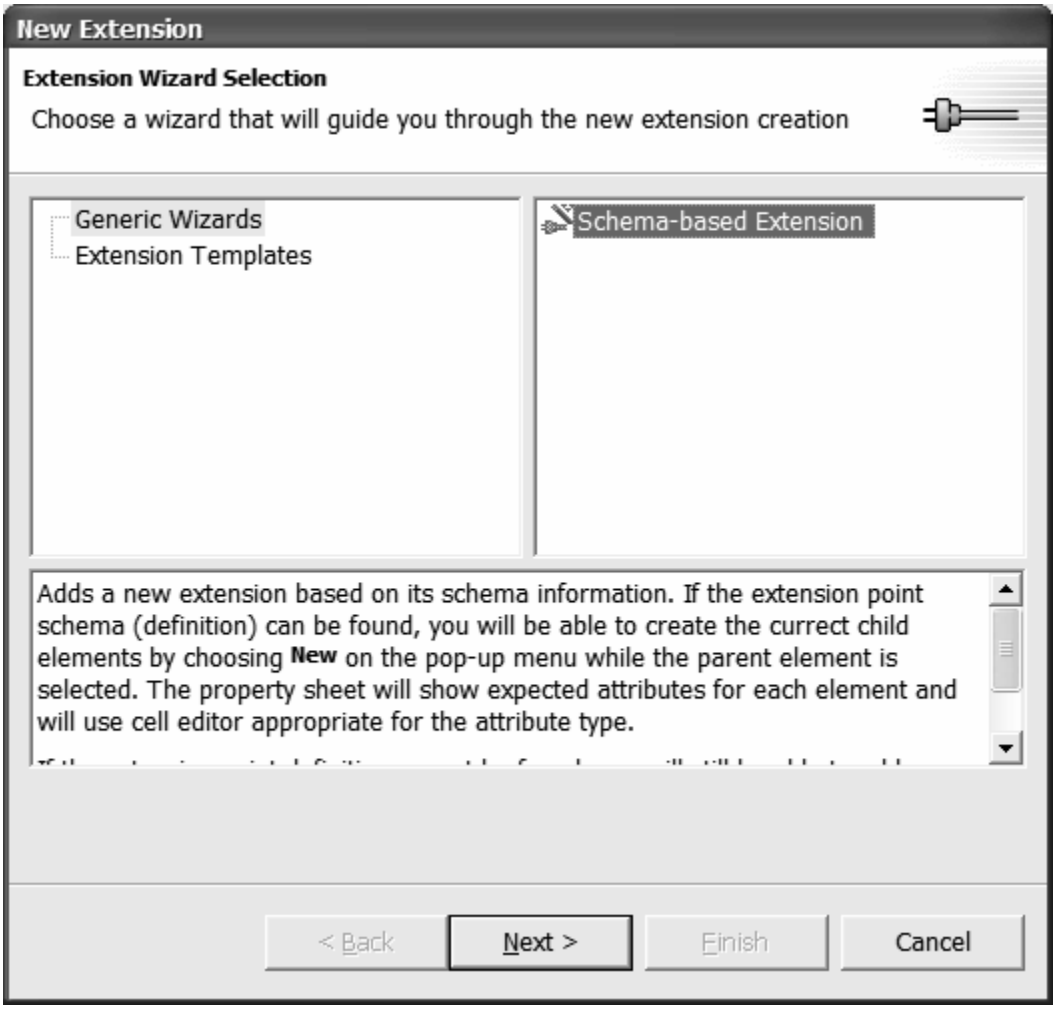


Figure 1-9
New Extension Wizard

5. Select the `org.eclipse.ui.actionSets` extension point (see Figure 3.10).

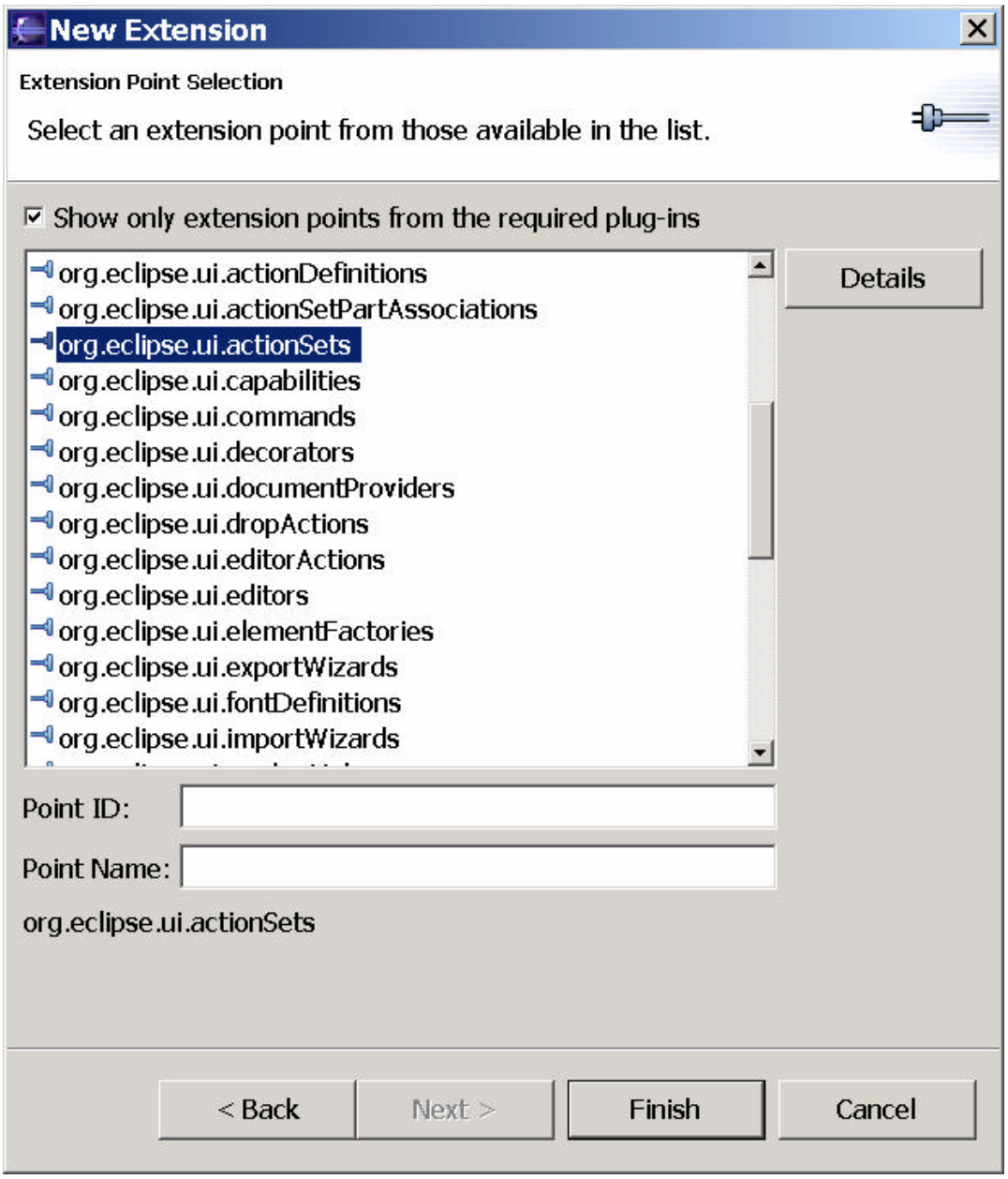


Figure 1-10
Extension Point Selection

This extension point is used to add menus, menu items, and toolbar buttons to the common areas in the Workbench window. These contributions are collectively known as an **action set** and appear within the Workbench window menu or toolbar. Select **Finish** to create the new extension.

6. There are very few extension points that do not require one or more child tags to complete the definition of the extension. In this particular case, the `<actionSet>` child tag must be added. Right-click on `org.eclipse.ui.actionSets` in the **All Extensions** list and select **New > actionSet** (see Figure 3.11).

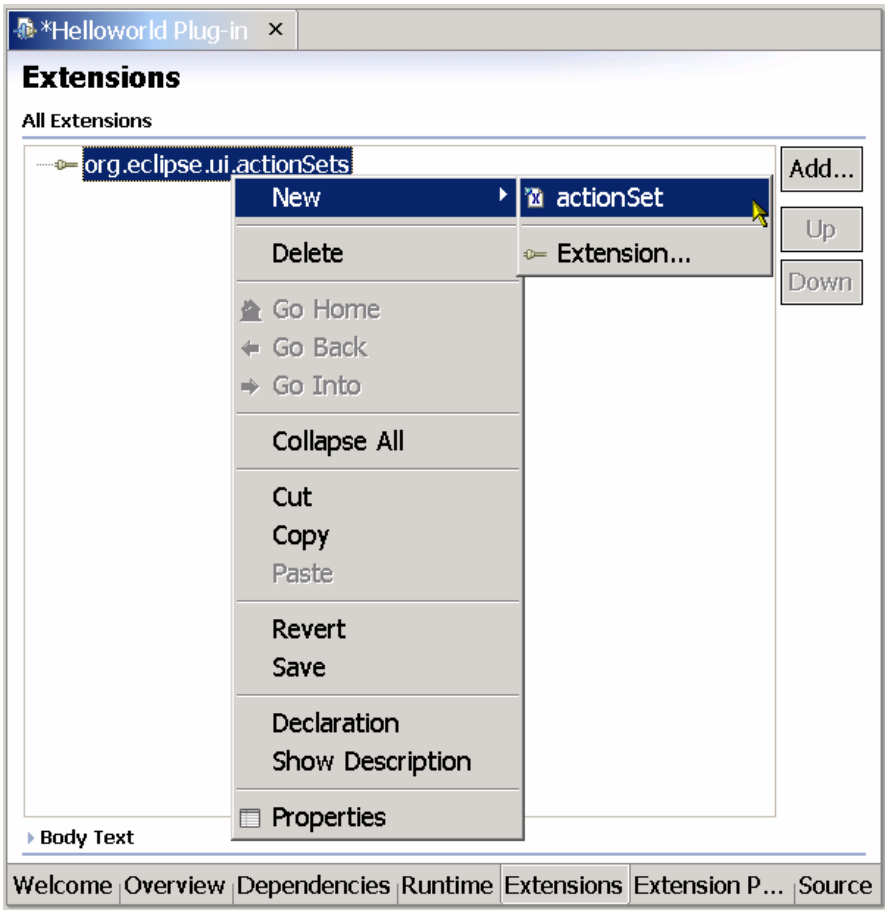


Figure 1-11
Adding an Action Set

7. Open the Properties view on the `actionSet` tag and set the `id`, `label`, and `visible` property values as shown in Figure 3.12.

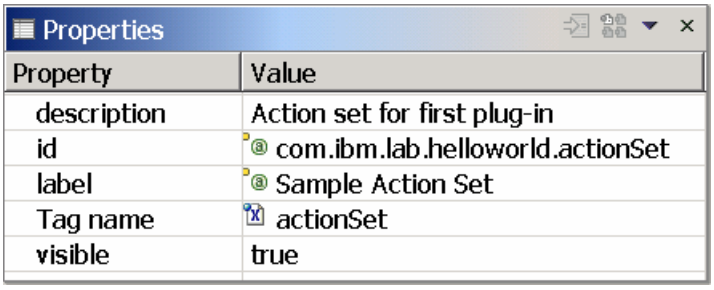


Figure 1-12
`actionSet` Properties View

Remember to press **Enter** after changing a property value; otherwise, the value may revert to its previous value when the Properties view loses the focus.

The `id` is a required unique identifier that can be used as a reference to this action set. The `visible` attribute indicates whether the action set should be initially visible in all

perspectives. The XML code will create an action set called “Sample Action Set.” If you turn to the **Source** page, you will see the XML that you’ve created.

```
<extension
  point="org.eclipse.ui.actionSets">
    <actionSet
      label="Sample Action Set"
      visible="true"
      id="com.ibm.lab.helloworld.actionSet">
    </actionSet>
</extension>
```

Note that the `visible=true` attribute is only honored when the user opens a new perspective that has not been customized. Selecting **Window > Reset Perspective** will show all action sets having the `visible` attribute set to `true`.

8. Now create a top-level menu. Right-click on the action set element and select **New > menu** (see Figure 3.13).

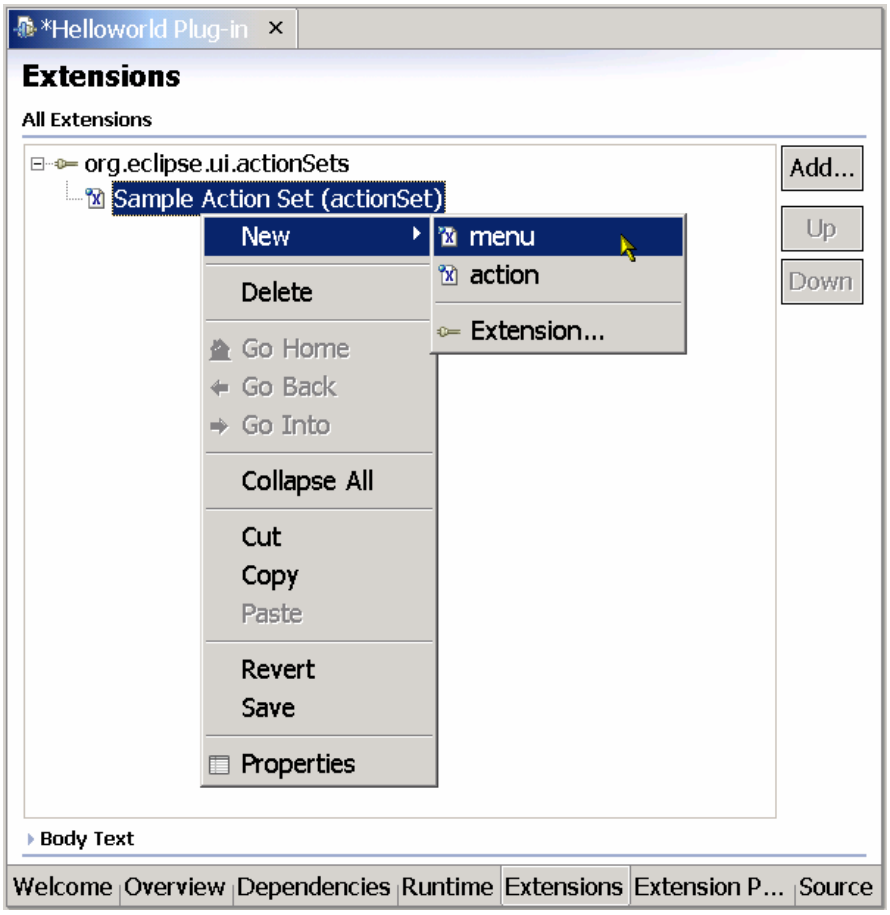


Figure 1-13
Creating a New Menu

Open the Properties view on `menu` and set the `label` and `id`. The ampersand (`&`) in the value for the `label` indicates the next character, M, is a menu accelerator.

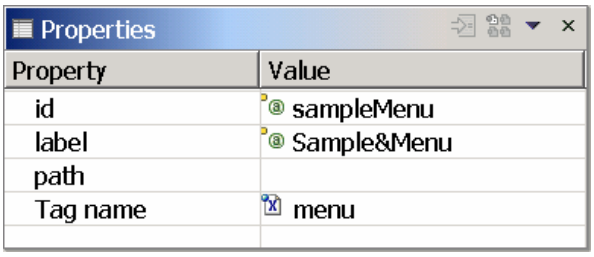


Figure 1-14
Updating the New Menu's Attributes

When you turn to the **Source** page, this value is shown as `&`, since the plug-in manifest is specified in XML, and that is the proper representation of an ampersand in XML. If you had entered it directly in the **Source** page as `&`, the manifest editor would detect an XML parsing error. If forced into a plugin.xml file, Eclipse would detect the error during the XML parsing that ocurrs at startup.

9. Now add a menu separator by right-clicking on the top-level menu and selecting **New > separator** (see Figure 3.15).

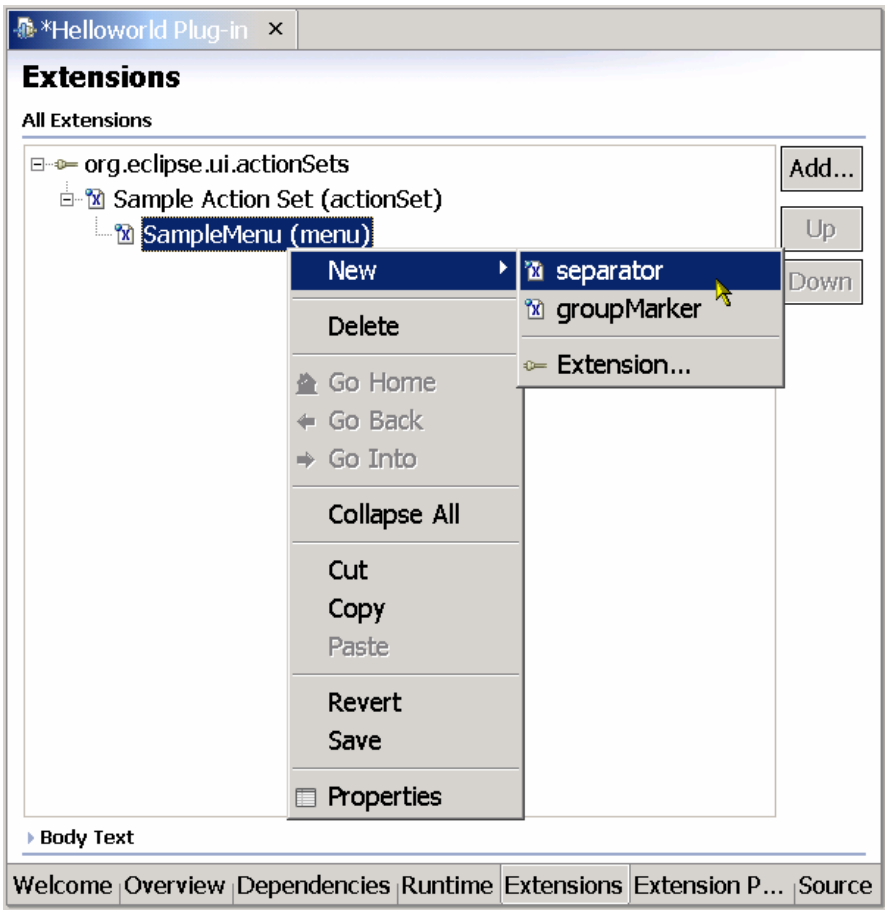


Figure 1-15
Adding a Menu Separator

Actions are generally inserted into a menu relative to another item. Follow this convention by adding a separator that provides a placeholder for your action and other plug-in

developers that might want to contribute menu items. Contributors would do so by specifying the separator's id in their action's `menubarPath` or `toolbarPath` attribute.

10. Change the separator's `name` attribute to `sampleGroup` in its Properties view. Remember to press **Enter** while the Properties view entry has the focus to register the change.
11. Now add a new action. You can do this by selecting `actionSet` in the **All Extensions** list as shown in Figure 3.16, and then select **New > action**.

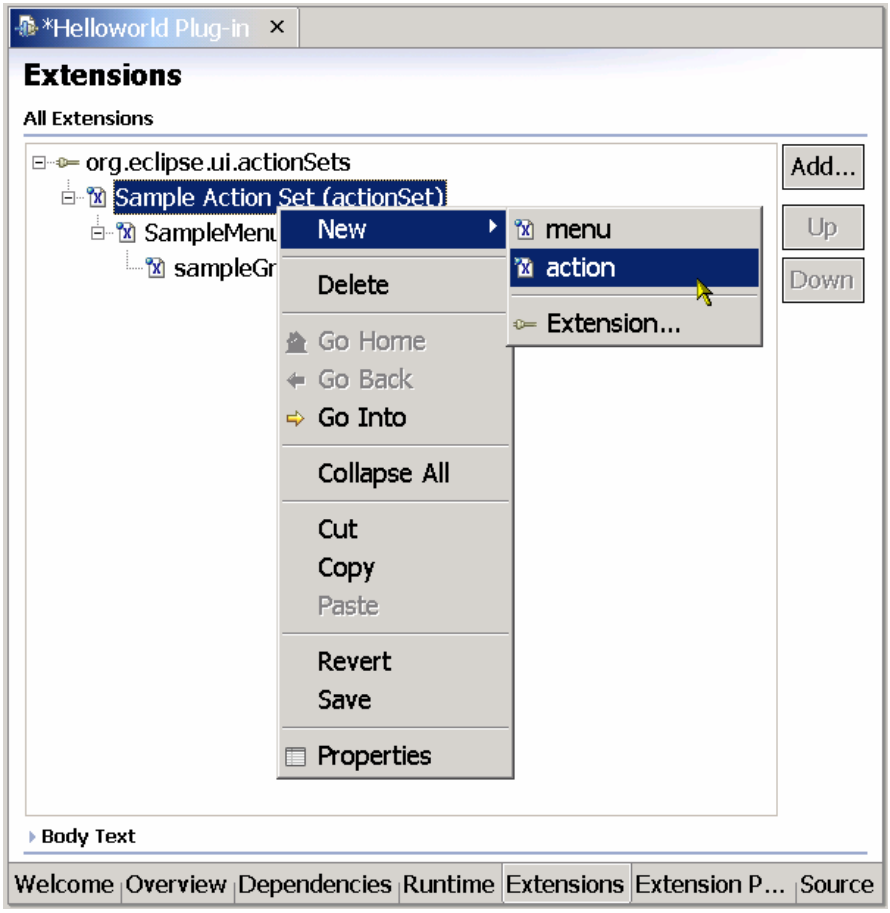


Figure 1-16
Adding a New Action

In the Properties view, set the action's `tooltip` property value to `Hello, Eclipse world`, the `id` to `com.ibm.lab.helloworld.SampleAction`, and the label to `&Sample Action`. Then set `menubarPath` to `sampleMenu/sampleGroup` and `toolbarPath` to `sampleGroup`. These attributes are slash-delimited paths that are used to specify the location of the action in the menu. Also note that while it is more typical to add an action to either the window menu or the main toolbar, you can add them to both at the same time in a single `<action>` tag by specifying both the `menubarPath` and `toolbarPath` attributes.

The `action` tag declares an action that will be available from the window menu and the main toolbar. When the user clicks on the action, the class referenced in the `class` attribute will be instantiated and its `run` method called.

12. Next you'll code a simple action class that will display a message dialog. To create the action class, select the `class` attribute in the action's Property view, then its "more" button (the button with the ellipsis next to the `class` attribute) to display the Java Attribute Editor, as shown in Figure 3.17.

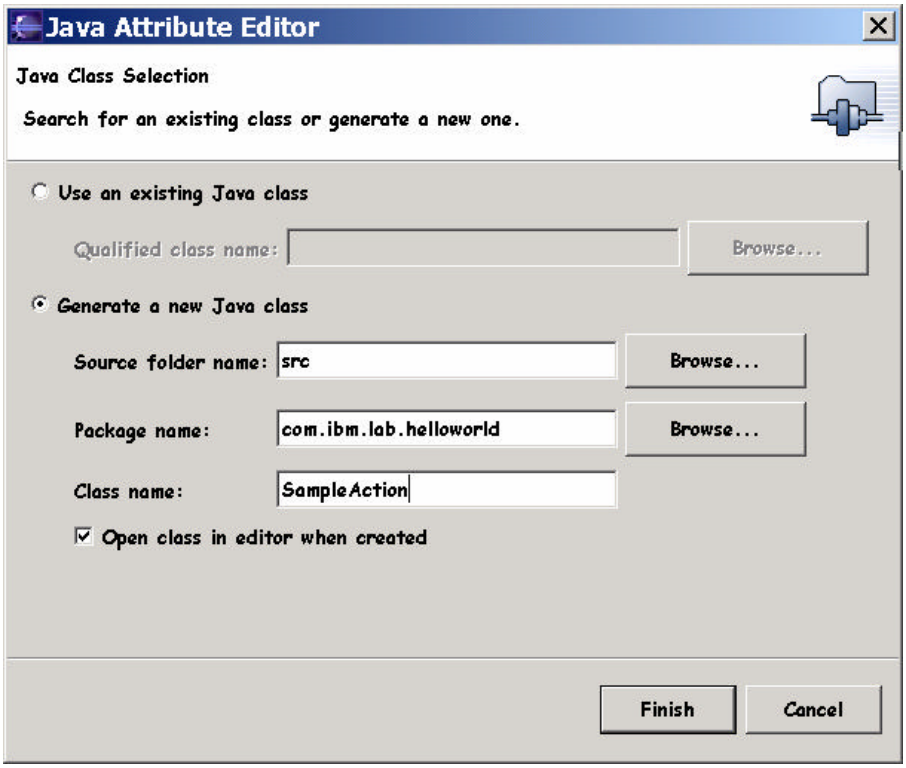


Figure 1-17
Creating a Sample Action

Select **Generate a new Java class**, accept the source folder name, and then enter the `com.ibm.lab.helloworld` package name (or use the **Browse** button) and the class name `SampleAction`.

After these last two steps the following XML will be added to the manifest file, along with XML for the other entries you just created.

```
<action
  label="&amp;Sample Action"
  class="com.ibm.lab.helloworld.SampleAction"
  tooltip="Hello, Eclipse world"
  menubarPath="sampleMenu/sampleGroup"
  toolbarPath="sampleGroup"
  id="com.ibm.lab.helloworld.SampleAction">
```

Then the class generator opens an editor on the source it created and adds a set of reminders to the Tasks view to implement your action (see Figure 3.18).

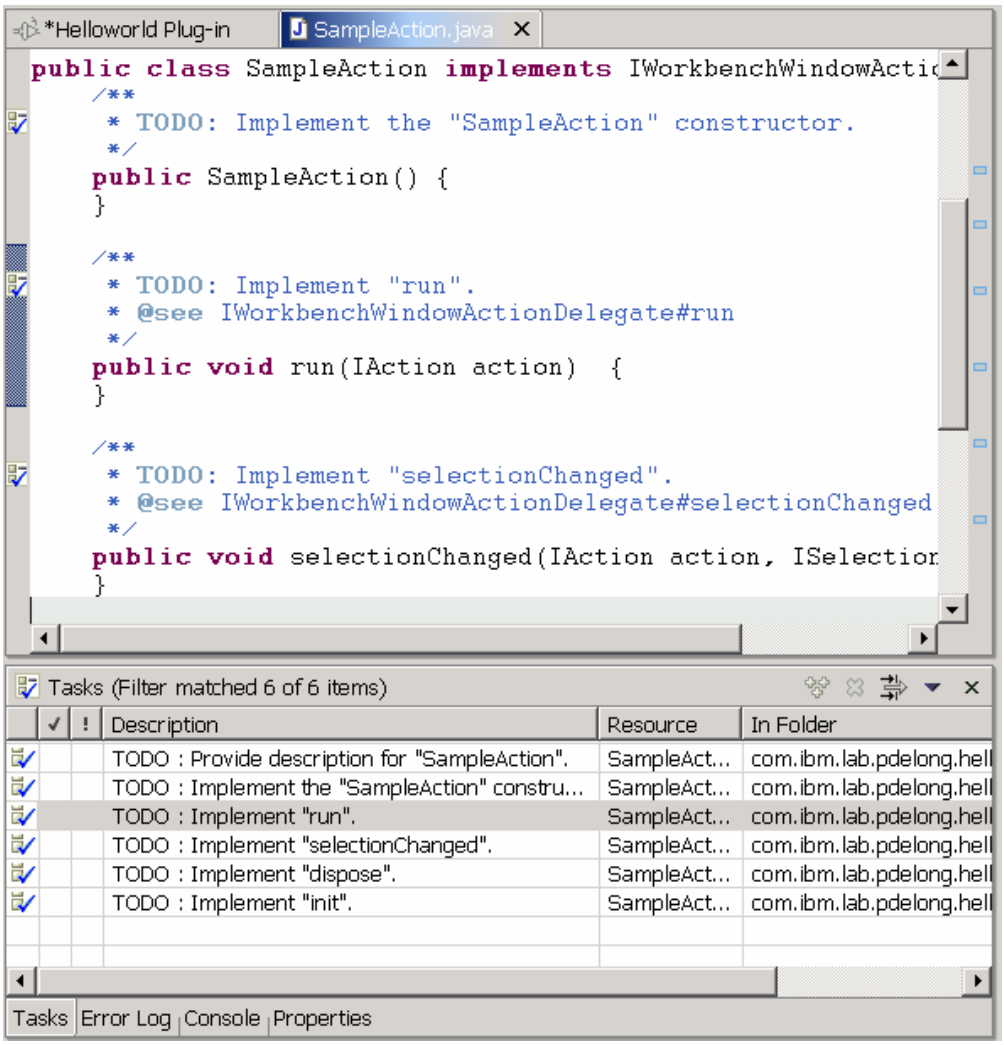


Figure 1-18
`SampleAction` Generated Code and Reminder

13.Verify the plugin.xml in the Source page. It should look like this.

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
   id="com.ibm.lab.helloworld"
   name="Helloworld Plug-in"
   version="1.0.0"
   provider-name="IBM"
   class="com.ibm.lab.helloworld.HelloworldPlugin">

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

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



```
   <extension
         point="org.eclipse.ui.actionSets">
      <actionSet
            label="Sample Action Set"
            visible="true"
            id="com.ibm.lab.helloworld.actionSet">
         <menu
               label="Sample &amp;amp;Menu"
               id="sampleMenu">
            <separator
                  name="sampleGroup">
            </separator>
         </menu>
         <action
               label="&amp;amp;Sample Action"
               class="com.ibm.lab.helloworld.SampleAction"
               tooltip="Hello, Eclipse world"
               menubarPath="sampleMenu/sampleGroup"
               toolbarPath="sampleGroup"
               id="com.ibm.lab.helloworld.SampleAction">
         </action>
      </actionSet>
   </extension>

</plugin>
```

14. The plug-in manifest is complete. However, to make this result the same as generated by the PDE's "Hello, World" example, you can add the following XML just above the closing `</plugin>` tag.

```
<extension
    point="org.eclipse.ui.perspectiveExtensions">
  <perspectiveExtension
      targetID="org.eclipse.ui.resourcePerspective">
    <actionSet
      id="com.ibm.lab.helloworld.actionSet">
    </actionSet>
  </perspectiveExtension>
</extension>
```

This isn't strictly required, but it's a good idea. It adds the new action set to an existing perspective, so users doesn't have to add it themselves with the **Window > Customize Perspective…** menu choice. The action set id above must match the action set id you specified earlier.

Switch to the **Source** page and enter the new extension. When you turn back to the **Extensions** page, notice that the list is updated to include the modifications you made in the **Source** page, specifically, the addition of the `org.eclipse.ui.perspectiveExtensions` extension. The Plug-in Manifest Editor keeps page modifications synchronized, wherever they are entered. This comes in handy when you want to make a minor change. That is, you can modify an attribute directly in

the **Source** page instead of selecting the associated extension in the **Extension** page list and then modifying the attribute with the Properties view.

15. Save the `plugin.xml` file.

You can now finish the implementation of the action logic in the `SampleAction` class.

16. Before an action's `run` method is invoked, the Workbench first calls its `init` method, providing the current Workbench window. Some actions need to know the context in which they are invoked, so they start by saving a reference to the Workbench window. This requires that we add an instance variable declaration and logic to save the window reference.

    Add this instance variable to the `SampleAction` class:

    ```
    private IWorkbenchWindow window;
    ```

    Add this logic to the `init` method:

    ```
    this.window = window;
    ```

17. The `run` method implements the action function. This action is simle, it will display your "Hello, Eclipse world" message. Add the code below to the action's `run` method.

```
public void run(IAction action) {
  MessageDialog.openInformation(
    window.getShell(),
    "Helloworld Plug-in",
    "Hello, Eclipse world");
}
```

Notice that the editor indicates that a Quick Fix is available to correct the "undefined" `MessageDialog` class (see Figure 3.19).

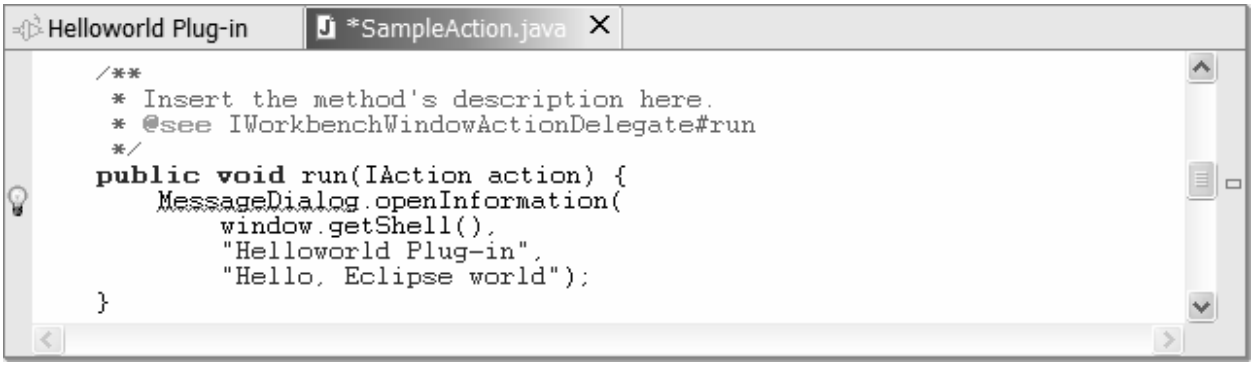


Figure 1-19
Quick Fix Indicator

Clicking the light bulb will propose several possible solutions. Choose to import the missing class reference, and then save your modifications to `SampleAction.java`.

You have just completed coding your first "Hello, World" plug-in, equivant to the one that you can create with the PDE's Plug-in Code Generator. Continue with the next section to test it.

Section 3: Testing with the Run-Time Workbench

You should have already completed Section 1, "'Hello, World' in Five Minutes or Less," or Section 2, "'Hello, World' with Detailed Step-By-Step Instructions." Whether you got here by taking the shortcut or the long way, you're ready to test!

1. Test your `com.ibm.lab.helloworld` plug-in by selecting the **Run > Run As > Run-time Workbench** menu choice (if you don't see this menu choice, verify that you're in the Plug-in Development perspective and you have a plug-in project, folder, or file selected).

   After a few seconds, a second instance of the Workbench will open. If the Resource perspective is active, the **Sample Menu** pull-down should already be shown. However, in other perspectives, your action set must be explicitly added to the user interface. In that case, select the **Window > Customize Perspective…** menu choice. You should see your **Sample Actions** in the list under **Other**. Select your action set to add it, as shown in Figure 3.20.

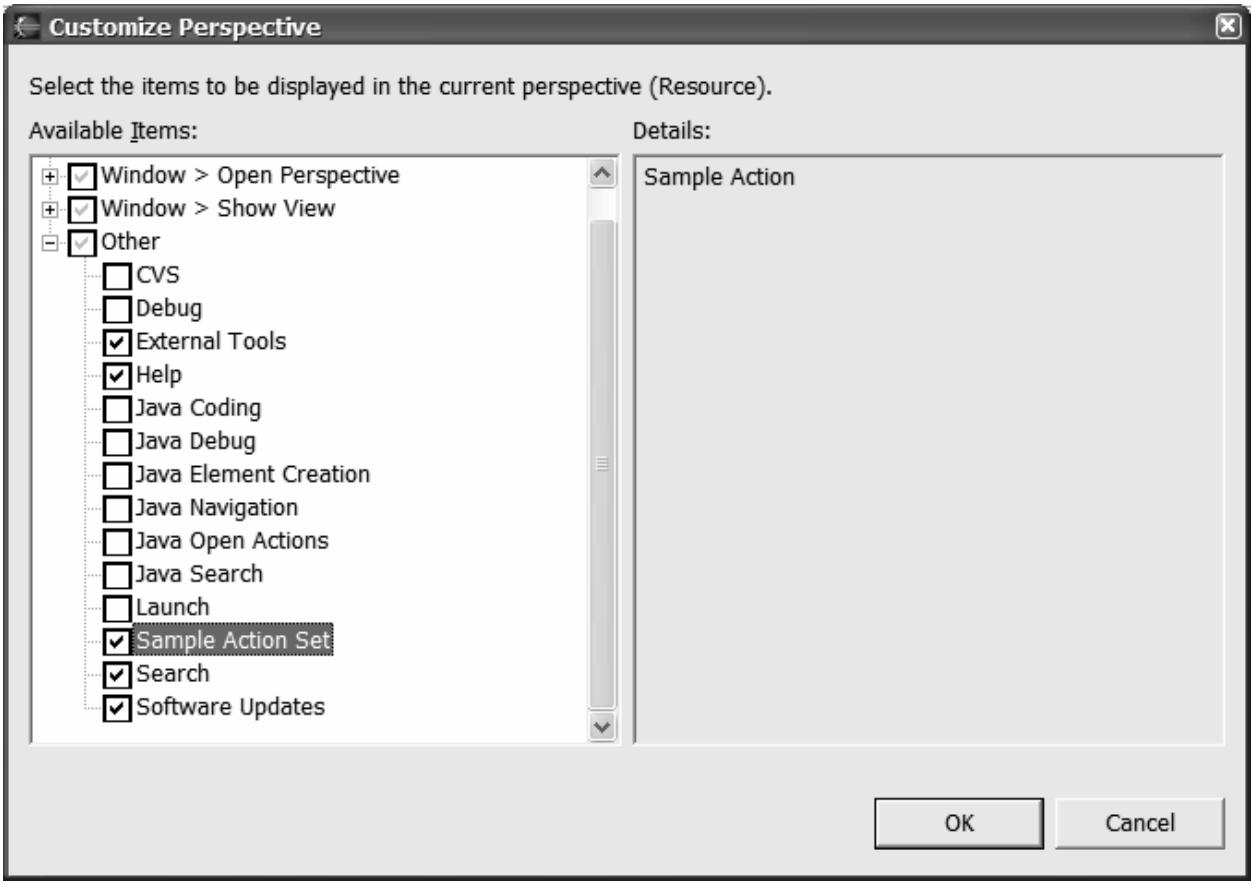


Figure 1-20
Adding an Action Set to the Current Perspective

You should see your addition in the Workbench menu bar. Note that the `visible=true` attribute of the `<actionSet>` tag is only honored when the user opens a new perspective that has not been customized. Selecting **Window > Reset Perspective** will show all action sets having the visible attribute set to `true`.

2. Select **Sample Menu > Sample Action** to display the message box (see Figure 3.21).

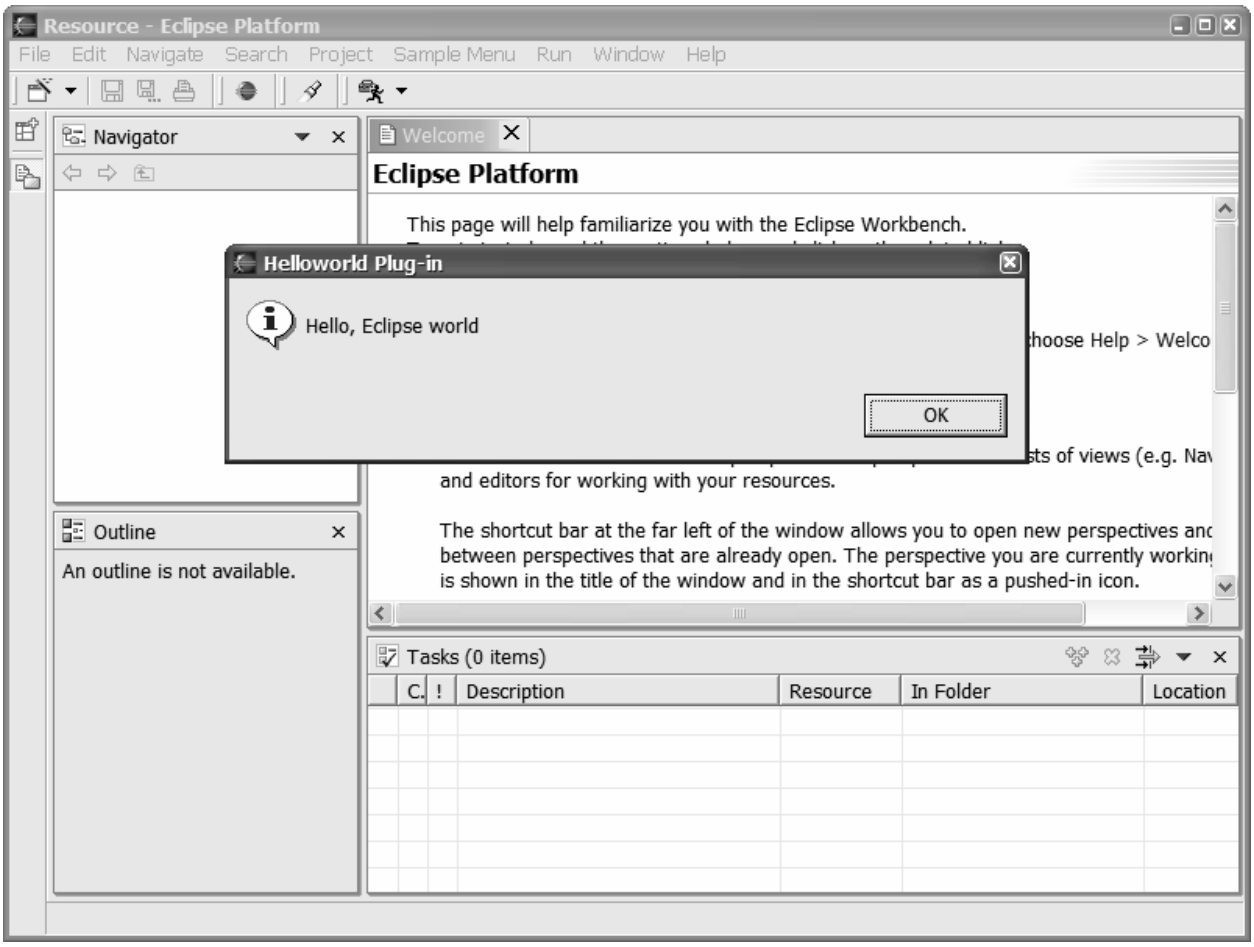


Figure 1-21
Hello, Eclipse World Message Box

Note: If you chose the 5-minute approach, your UI will have a small round Eclipse image for the contributed tool bar action (shown in Figure 3.21). If you chose the detailed approach, you would not have specified an image so the Workbench will use a red square (default for missing images) to represent your action.

3. Before you close the run-time instance of the Workbench, close the **Welcome** page. This will avoid the spurious version 2.0-only message "`An error has occurred while restoring the workbench; See error log for more details.`" Be sure to close the second instance of the Workbench.

Congratulations, you have just created and tested your first plug-in for Eclipse! Now you are ready to try your hand at debugging a plug-in in the next section. If your plug-in didn't work as expected, see the section "Correcting Common Problems" later in this exercise for help.

Section 4: Debugging with the Run-Time Workbench

This section explores the tools the PDE has to help debug your plug-ins. You already have coded and tested your "Hello, Eclipse world" example, so how about intentionally introducing some bugs to see how they manifest themselves? The short debug session that follows is an example of how to find plug-in specific errors. Begin by verifying that you've closed the run-time Workbench from the prior section, and then return to the Plug-in Development perspective of your Eclipse development environment. Next, open your plug-in's manifest file.

1. Turn to the **Source** page and introduce an error in the `class` attribute of the `<action>` tag.

```
<action
  label="&amp;amp;Sample Action"
  class="com.ibm.lab.hello.SampleAction"  <!-- error, was "helloworld" -->
  tooltip="Hello, Eclipse world"
  menubarPath="sampleMenu/sampleGroup"
  toolbarPath="sampleGroup"
  id="com.ibm.lab.helloworld.SampleAction">
</action>
```

When you launch the run-time Workbench and select **Sample Action**, a dialog is displayed that indicates that the chosen operation is not currently available, and a message is displayed in the Console of your development Workbench, as shown in Figure 3.22.

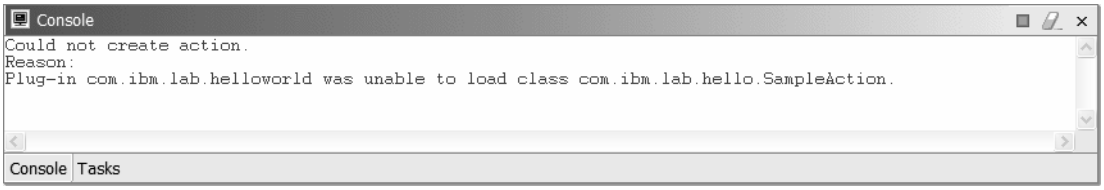


Figure 1-22
Console Error Message

That is, messages in the run-time instance to `System.out` and `System.err` are redirected to the Console in the development Workbench. Before closing the run-time instance, open the Plug-in Registry view (**Window > Show View > Other... > PDE Runtime > Plug-in Registry**) and scroll down to your plug-in, as shown in Figure 3.23.

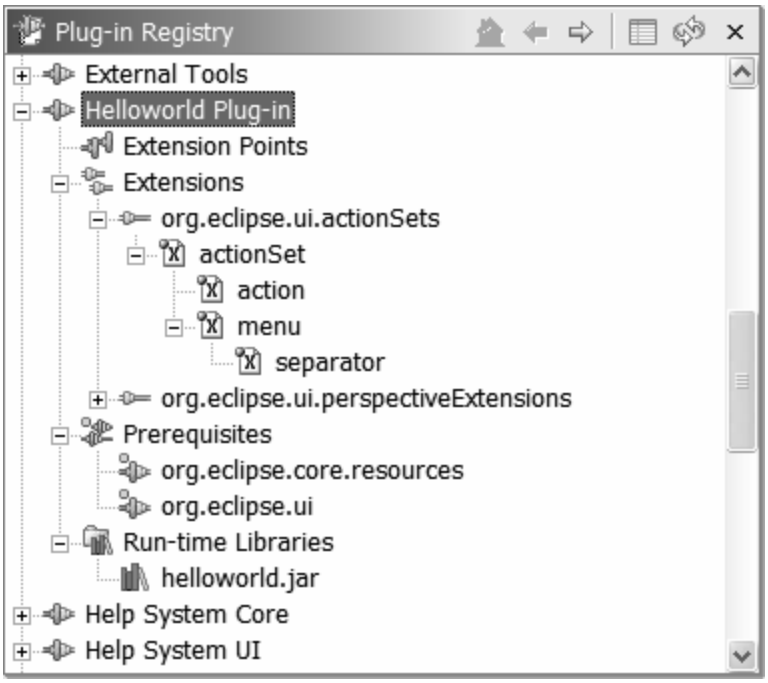


Figure 1-23
Plug-in Registry

From here you can see precisely what was parsed from your `plugin.xml` file, similar to the Outline view of the plug-in manifest, but available at runtime. Close the run-time Workbench.

2. Correct the error from the prior step (e.g., by selecting `plugin.xml`, and then **Replace With > Previous From Local History**). Now let's introduce a more serious error. Comment out the code below from `SampleAction`.

```
 public void init(IWorkbenchWindow window) {
//      this.window = window;
 }
```

This change will provoke a null pointer exception in the `run` method. Save the change and relaunch the run-time Workbench, this time using the **Run > Debug As > Run-time Workbench** menu choice. Notice that the perspective automatically changes to the Debug perspective.

3. Again select the **Sample Action** menu choice. Nothing appears to happen. No message from the run-time instance, so look in the Console of the *development* Workbench. As expected, the message `Unhandled exception caught in event loop. Reason: java.lang.NullPointerException` is displayed. To get more details, go back to the run-time instance of the Workbench and open the plug-in Error Log (**Window > Show View > Other... PDE Runtime > Error Log**). Indeed, there are two new entries. Double-click the `java.lang.NullPointerException` message and select **Status Details**, as shown in Figure 3.24.

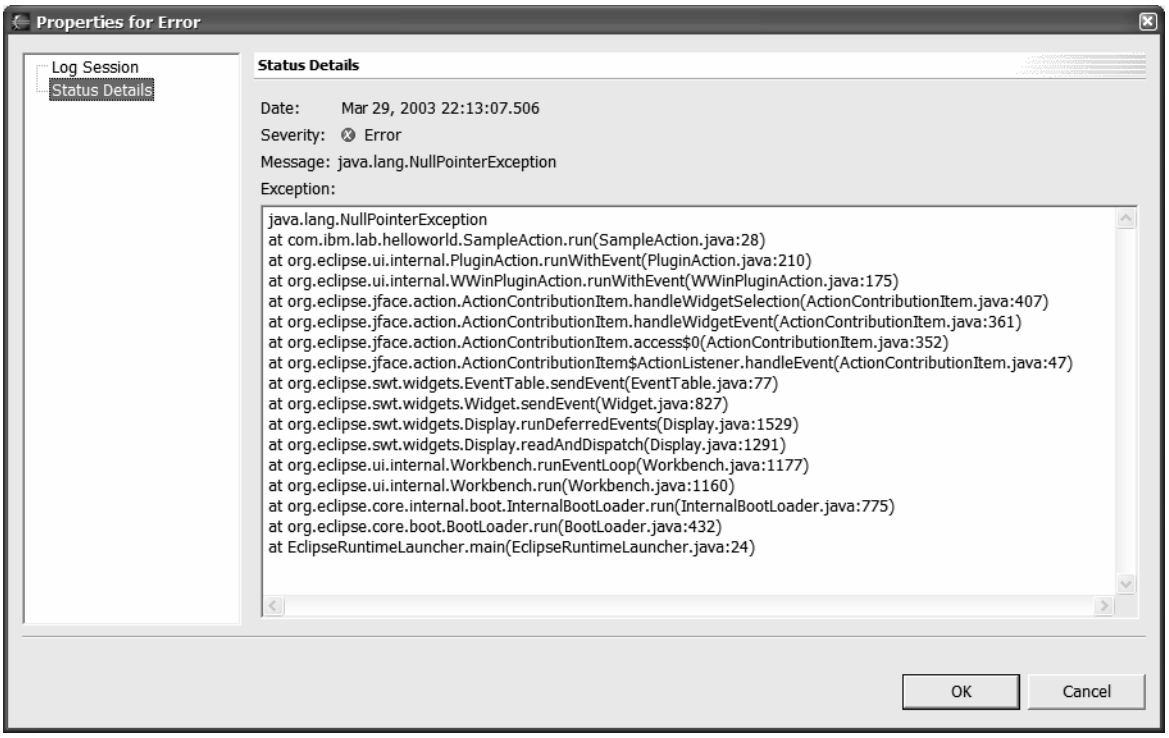


Figure 1-24
Error Status Details

At this point in a real debug session you might consider setting an exception breakpoint for `NullPointerException` from the **J!** button on the **Breakpoints** page to further diagnose the problem. Don't set such an exception breakpoint before launching the Workbench; it will stop in a lot of places that have nothing to do with your problem. Instead, set and then disable the exception you want to debug before starting the Workbench, then enable it when you're ready to reproduce the problem.

4. Close the run-time instance before continuing.

This short debug session gives you a flavor of debugging plug-ins. As you create your own plug-ins, you'll find more difficult problems that this one. When that happens, refer to the section "Correcting Common Problems" later in this exercise.

Section 5: Exploring (and Sometimes Correcting) the Eclipse Platform Code

One of the benefits of an open source project is the fact that the source is yours to study, and if necessary, correct. Let's see how the PDE helps you to learn and modify Eclipse code.

You have already been introduced to the notion of "external" versus "workspace" plug-ins; the Target Platform preference page, shown back in Figure 3.1, allows you to add external plug-ins to the list of those available in the test environment and your plug-in's build path. But what if you want to modify the code found in an external plug-in to help you debug or to correct a bug in the Eclipse code? The PDE includes options in the Plug-in view that makes it easy, as shown in Figure 3.25.

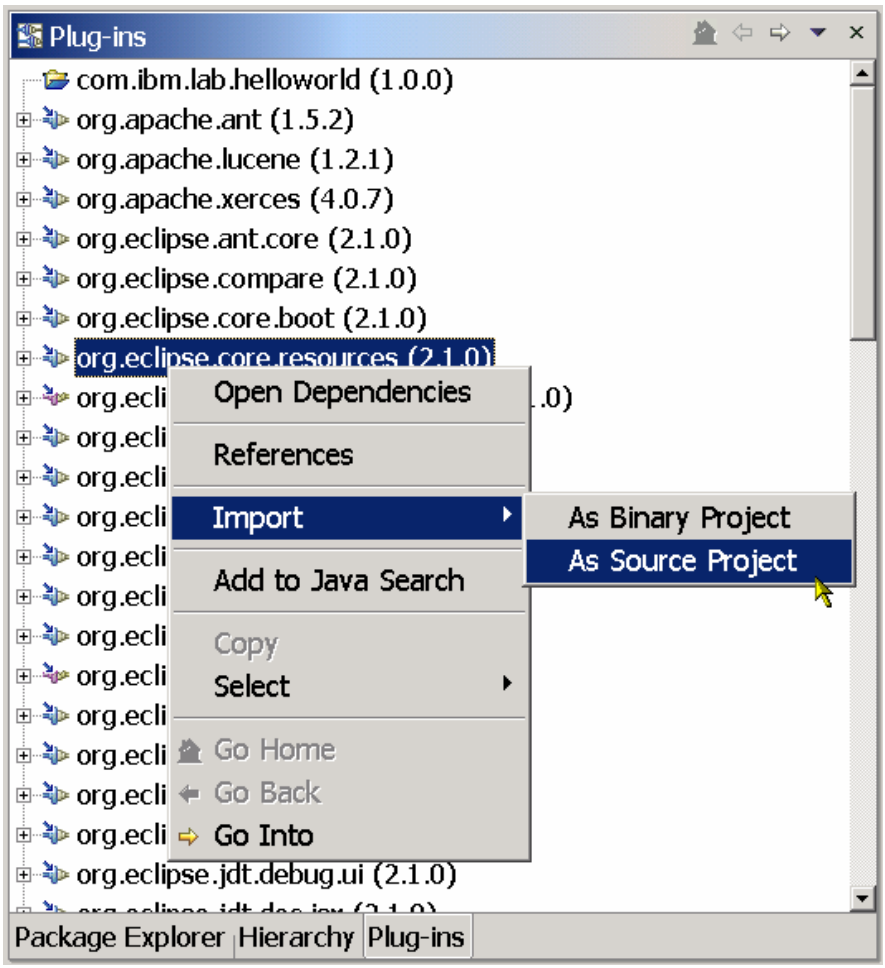


Figure 1-25
Import External Plug-in As Source Project

To get a better idea of how this works, let's import one of the Eclipse plug-ins and add some debug code.

1. If you haven't closed the run-time instance of Eclipse, do so now. Then turn to the Plug-ins view as shown in Figure 3.25, select the `org.eclipse.core.resources` plug-in, and then select **Import > As Source Project**. This will copy the plug-in from the `plugins` directory to your workspace, including its source, and recompile it. After recompiling, you'll notice quite a few errors in the Tasks view, such as "The project was not built since it is involved in a cycle or has classpath problems" and "Missing required Java project." This is because the original Eclipse plug-ins were built in a single workspace, while yours has some plug-ins in your workspace and others in the `plugins` subdirectory. Turn to the Package Explorer view, select the project, select **Update Classpath…** from its pop-up menu, and then select **Finish** in the dialog. This will recalculate the project's build (class) path based on your configuration and recompile, correcting the above errors.
2. Repeat the procedures in step 1 for the `org.eclipse.core.runtime` plug-in, but this time select **Import > As Binary Project** instead. After updating the classpath, look at both projects in the Package Explorer view. Notice that the source importation results in Java source files available in the `src-resources` folder on the left, and Java class files in the `runtime.jar` file on the right, as shown in Figure 3.26.

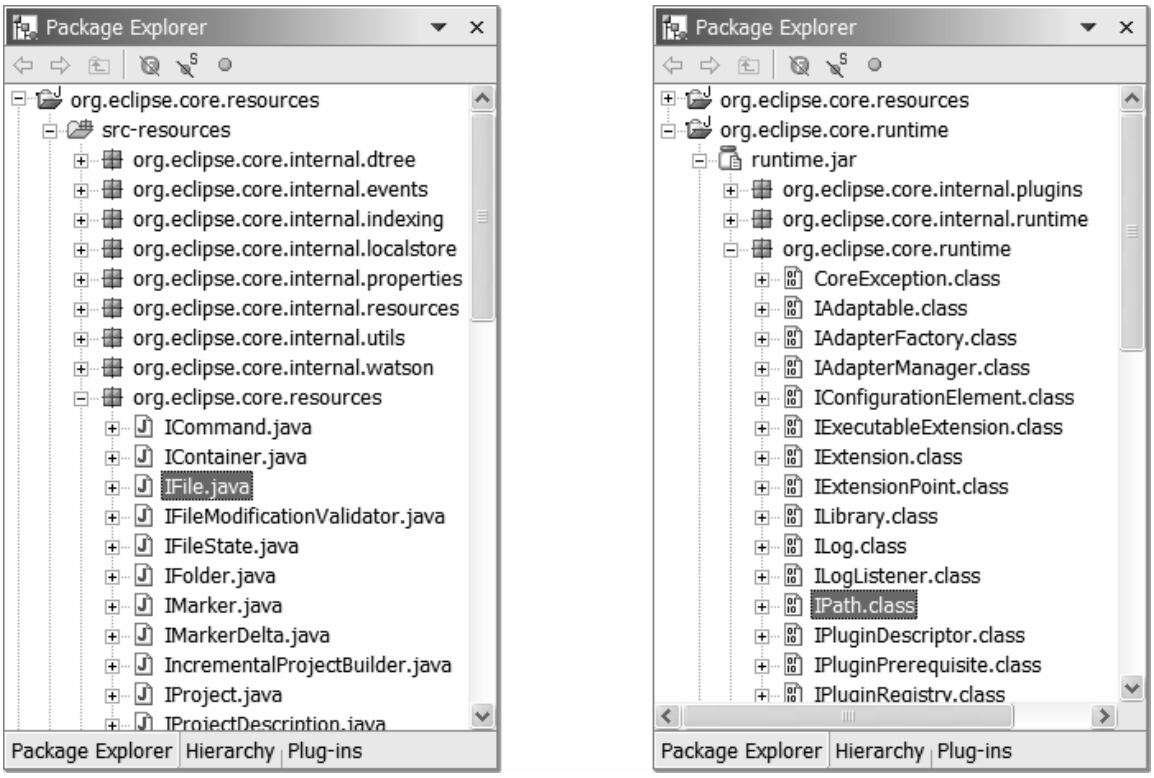


Figure 1-26
Import External Plug-in Results

Both projects are now in your workspace, and therefore are included in all workspace searches, but the `org.eclipse.core.resources` plug-in can be modified as well. If you turn back to the Plug-ins view, note that the project icon has changed from an external plug-in () to a folder, indicating that it is now in your workspace.

3. To see an example of how you might use this to help your debugging, let's assume that you want to know more about what resources (projects, files, and folders) are created and when. Begin by opening the class the represents them, `Resource`, in the package `org.eclipse.core.internal.resources` by selecting **Navigate > Open Type…** or pressing **Ctrl+Shift+T** (more than one matching package is shown in the Open Type dialog; choose the one located at `/org.eclipse.core.resources/src-resources`). Add a debug `System.out.println` statement in the constructor, as shown below.

```
protected Resource(IPath path, Workspace workspace) {
  this.path = path.removeTrailingSeparator();
  this.workspace = workspace;

   // Debug code.
  System.out.println
    ("Created resource " + path + " in workspace " + workspace);
}
```

This will generate output to the Console whenever a new resource is created.

4. Launch the run-time Workbench. You should see the warning shown in Figure 3.27, since the two plug-ins that you imported are in the workspace and are also specified in the **Plug-in Development > Target Platform** preference page as external plug-ins.

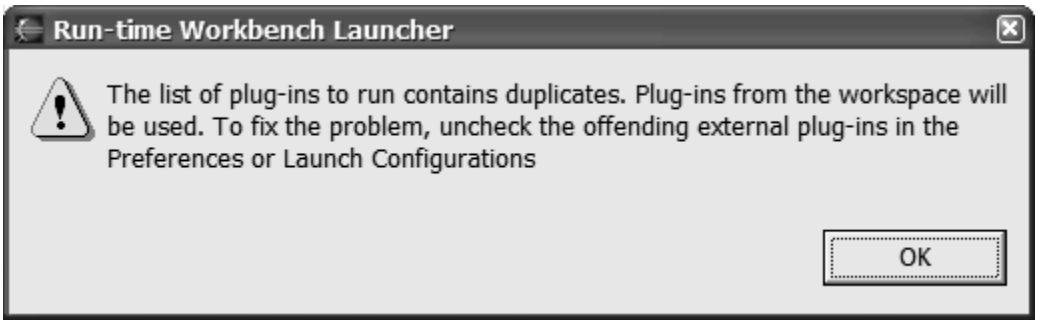


Figure 1-27
Duplicate Plug-ins Warning

If you want to avoid this warning, go back to this preference page and select **Not in Workspace**. That will automatically deselect the two plug-ins that you imported. Verify that your debug code shows its output in the Console by creating a new project, folder, and file.

This short example demonstrates how you can add debug code, and how you could also apply your own fixes to Eclipse Platform code, should the need present itself.

Section 6: Correcting Common Problems

Table 3.1 lists some of the more common errors you might encounter when writing your first few plug-ins, their symptoms, and possible resolutions. If you don't find the problem you're seeing in this table, follow the "Hints and Tips" link from the `readme.html` on the CD-ROM included with this book, which has more suggestions. If you are still stuck, consider posting a message to the eclipse.org newsgroups.

Table 1.1 Common Errors and Possible Resolutions

Symptom	Source	Possible Resolution
"Plug-ins required to compile Java classes in this plug-in are currently disabled. The wizard must enable them to avoid compile errors."	Warning message dialog when creating new project	Accept the suggested action. This is equivalent to selecting **Not in Workspace** in **Preferences > Plug-In Development > Target Platform**.
"The project was not built since it is involved in a cycle or has classpath problems," or one of several errors similar to "Missing required Java project: org.eclipse.xxx."	Tasks view	Verify manifest is not missing required plug-ins in the `<import>` statements of the `<runtime>` tag. Verify that required plug-ins are either imported into workspace or available from the **Preferences > Plug-In Development > Target Platform** list. Then select **Update Classpath...** for the affected project. For Eclipse version 2.0 only: This suggested correction might not work if the project was created before setting the external plug-ins on the **Target Platform** page. An easy, albeit dramatic, workaround is to start with a fresh workspace, remembering to set the external plug-ins first.
"This compilation unit indirectly references the missing type java.lang.Object (typically some required class file is referencing a type outside the classpath)" or "This plug-in contains unresolved and/or cyclical references to other plug-ins."	Tasks view and **Overview** page of Plug-in Manifest Editor	Project build path is incorrect. The PDE will automatically update the build path of plug-ins when the `<requires>` tag is modified, unless the **Preferences > Plug-In Development > Java Build Path Control** options are deselected. Verify and try **Update Classpath...** for the affected project. If this doesn't correct the errors, manually add the required plug-ins from the project's **Properties > Java Build Path > Library > Add Variable** dialog. Select `ECLIPSE_HOME` and add the required plug-in JAR files by selecting the **Extend...** button.
"The list of plug-ins to run contains duplicates. Plug-ins from the workspace will be used. To fix the problem, uncheck the offending external plug-ins in the Preferences or Launch Configurations."	Warning message dialog when attempting to start run-time Workbench	There are one or more plug-ins that are present in the workspace and in the **Preferences > Plug-In Development > Target Platform** list or the **Plug-in and Fragments** page of the launch configuration (**Run...** or **Debug...**). This often occurs after: ▪ Importing a plug-in into the workspace with **Import > External Plug-ins and Fragment** or one of the Plug-ins view's **Import** menu choices ▪ Copying a plug-in that exists in the workspace to the `plugins` directory ▪ Or installing a feature that references plug-ins that exist in the workspace Either close the project of the duplicate plug-in in the workspace so it will be ignored, or deselect the appropriate external plug-in in the list, as the message suggests.

"Exception launching the Eclipse Platform" followed by a long exception trace.	Error Log or Console	Check if the run-time Workbench is already running. If this is the case, you will see this message at the bottom of the exception trace, "The platform metadata area is already in use by another platform instance, or there was a failure in deleting the old lock file. If no other platform instances are running, delete the lock file and try starting the platform again." This can occur after the Workbench abnormally terminates. Verify that the run-time Workbench is closed, or if it is already terminated, manually delete the `.lock` file.
Contributed action is not present.	Run-time Workbench	Verify the `id` attributes of your actions–there may be a duplicate. The actions are stored in a keyed table, so duplicate entries are lost.
Contributed pull-down menu is disabled or contributed action is not present.	Console	Verify the `menubarPath` attribute. Look in the Console for the message "Invalid Menu Extension (Path is invalid)." Check the spelling, especially if the word "separator" is in the menu path. Many programmers misspell this word. Consider using "groupXXX."
"The chosen operation is not currently available." is displayed after selecting a contributed action.	Run-time Workbench	There are several possibilities. ▪ Check the Console. If it contains the error message, "Could not create action. Reason: Plug-in xxx was unable to load class yyy," where "xxx" is your plug-in and "yyy" is your action class, verify that your specification of the action's `class` attribute is correct, the code compiled correctly, and there are no build path errors associated with your plug-in. ▪ Check the enablement specification of your action; it may be inconsistent with the enablement logic of your action's `selectionChanged` method. For example, your action's XML specifies the action is available only if the selection is an `IFile` with extension `.java`, but the action's `selectionChanged` method checks for an `IFile` with the extension `.class`. The static enablement logic of your action and its dynamic logic must be consistent.
"An error has occurred while restoring the workbench; See error log for more details." is displayed and "Unable to restore editor - createElement returned null for input element factory: org.eclipse.ui.internal.dialogs.WelcomeEditorInputFactory" is shown in Console.	Error message dialog when launching run-time Workbench	A harmless message in the version 2.0 Workbench that you can safely ignore (it has since been corrected). Closing the **Welcome** page of the run-time Workbench will avoid the message.

Don't be intimidated by the number of entries in this table. It includes problems that are not very likely for your "Hello, World" exercise, but they may prove helpful should you decide to experiment beyond the instructions in this exercise.

Exercise Activity Review

In this exercise you used the PDE to create, test, and debug a plug-in. You now have the basics of writing, testing, and debugging a plug-in. Return to Part II where you left off— it will lead you further in your study of the Eclipse extension points and frameworks that you can employ to enhance the Workbench's capabilities.

Exercise Setup: Import Exercise Templates and Solutions

Introduction

The lab materials are provided in the `Lab_Import_Master.zip` file. This zip file contains templates and solutions to the exercises. This procedure setting up your workspace for plug-in development and the import of project templates into the workspace.

The process involves:

1. Initializing the Plug-in Development Environment (required once per workspace)
2. Importing plug-in projects from the file system to a workbench workspace
3. Re-calculating the project classpath

Exercise Instructions

Part 1: Initialize the Plug-in Development

1. Open a Plug-in Development perspective, using **Window > Open Perspective > Other…**

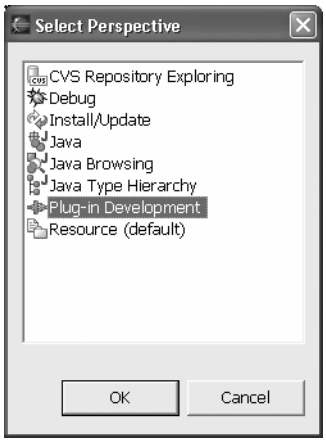


Figure S-1
Open PDE Perspective

2. Open the **Plug-In Development > Target Platform** preference page. Select **Not In Workspace** to make all plug-ins visible.

   This PDE Target Platform configuration adjustment ensures that all external plug-ins are visible. This configuration simplifies plug-in development and is oriented towards new workbench users.

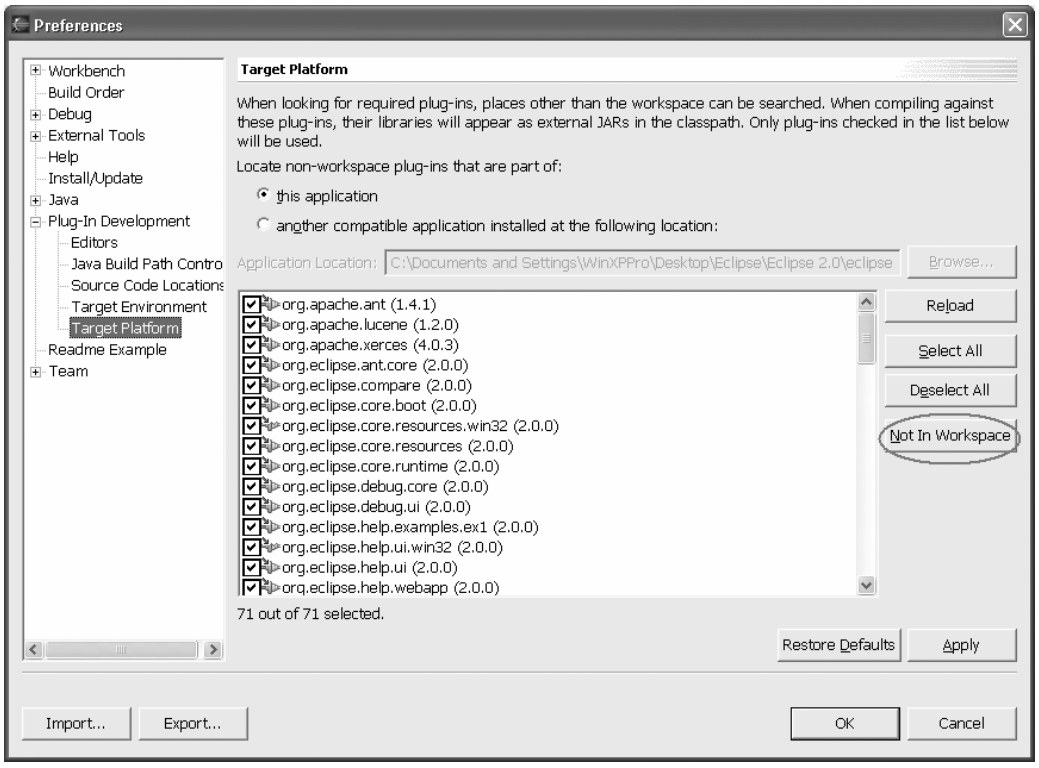


Figure S-2
Configure Target Run-times plug-ins

Part 2: Importing Plug-in Projects

The plug-in import wizard is used to create a plug-in project from plug-in sources in the file system. You should already have extracted the contents of the `Lab_Import_Master.zip` file to directory location on your file system. For example, if you unzipped the file to C:\ you will have these directories:

```
C:\Lab_Import_Master
   \JavaProjects
   \PluginLabTemplatesCore
   \PluginLabTemplatesOptional
   \SolutionsToPluginLabs
```

To do the labs used during the course you need to import the projects found in the `\PluginLabTemplatesCore` directory. You can import the optional lab templates if desired, and the solutions.

1. Start the plug-in import wizard with workbench menu **File>Import**. Select **External Plug-ins and Fragments** and press **Next.**

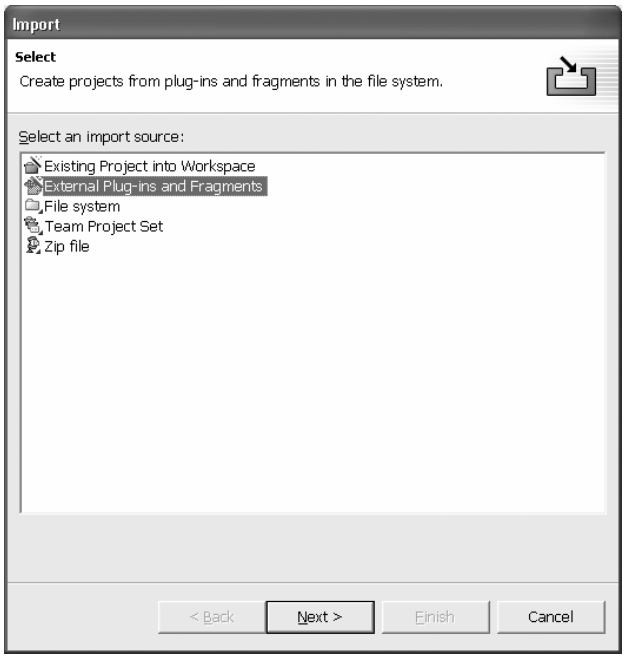


Figure S-3
Import Dialog

2. On the **Import External Plug-ins and Fragments** page of the plug-in import wizard:
   - Deselect **Choose from plug-ins in the run-time workbench**.
   - Select **Extract source archives and create source folders in projects**.
   - In the location field, browse to the directory location where the plug-in zip was extracted.

   Once done, select **Next**.

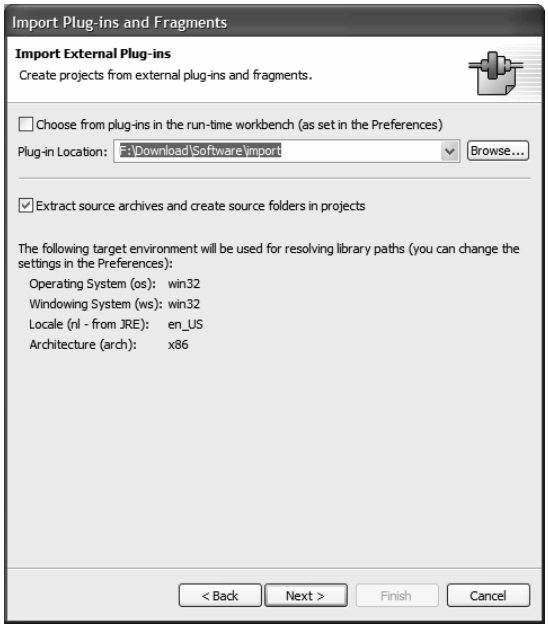


Figure S-4
Plug-ins and Fragments

3. On the **Selection** page of the plug-in import wizard, **Select All** and press **Finish**.

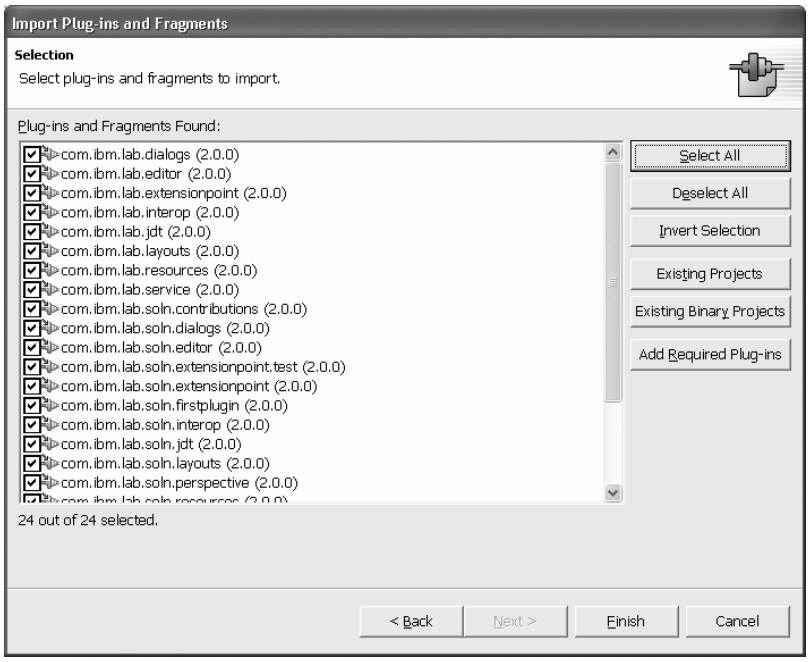


Figure S-5
Plug-in selection

The import procedure completes with an automatic build. The build may result in errors such as those listed in the Tasks view shown in Figure S-6.

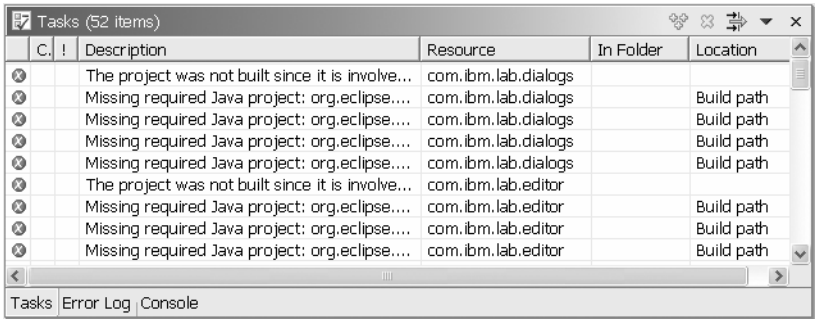


Figure S-6
Task List

Note: When using Eclipse 2.1 the list of errors may not be the same as those shown.

The next part of the import process will resolve these errors. These steps should be performed even if you do not have the same errors as those listed in Figure S-6.

Part 3: Update Project Classpath

1. The classpath must be updated for each project. Select any plug-in project and then choose the **Update Classpath…** pop-up menu.

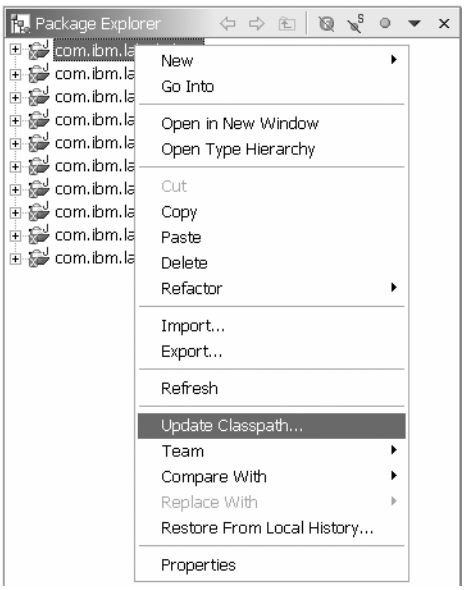


Figure S-7
Project context menu

2. On the Update Java class path dialog, choose **Select All** and then **Finish**.

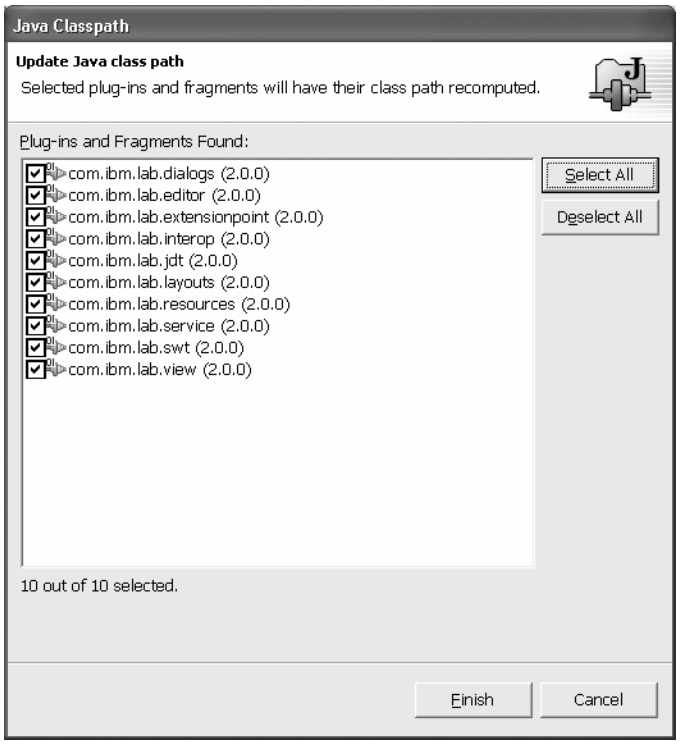


Figure S-8
Update Classpath Dialog

Note: After updating the project classpath on all of the projects, some errors may remain on certain packages. These errors will be worked out as the lesson are completed

Exercise 2: SWT Programming

Introduction

This exercise will show you how to use some of the SWT controls.

Skill Development Goals

At the end of this lab, you should have a general understanding of how to create a user interface with SWT and understand their event handling.

Exercise Setup

Setup tasks must be complete before you begin the exercise.

A PDE project has been set-up for you named `com.ibm.lab.swt`. We will use a view as a container for our SWT widgets. You may not be familiar with programming views yet, so a plug-in containing an empty view and view class has been defined for you.

Load the project `com.ibm.lab.swt` into your workspace.

Note: You can use code snippets from the SWTView.jpage file included in the exercise template to save you from having to retype them.

Exercise Instructions

Part 1: Use buttons to invoke the FileDialog and the MessageDialog

1. Add the `SelectionListener` interface to the class as defined below

```
public class SWTView extends ViewPart implements SelectionListener{
// … code …
}
```

To clear up any compile problems use the editor context menu called **Source > Organize Imports** or **Add Import** (Ctrl+Shift+M). You will need to use this feature frequently in the lab.

Select the class in the outline view and from the context menu, select **Source > Override/Implement Methods...** to generate the required methods of the interface. See Figure 2-1. You should add all the methods that the `ISelectionListener` interface defines, although for the moment we are only interested in the `widgetSelected` method.

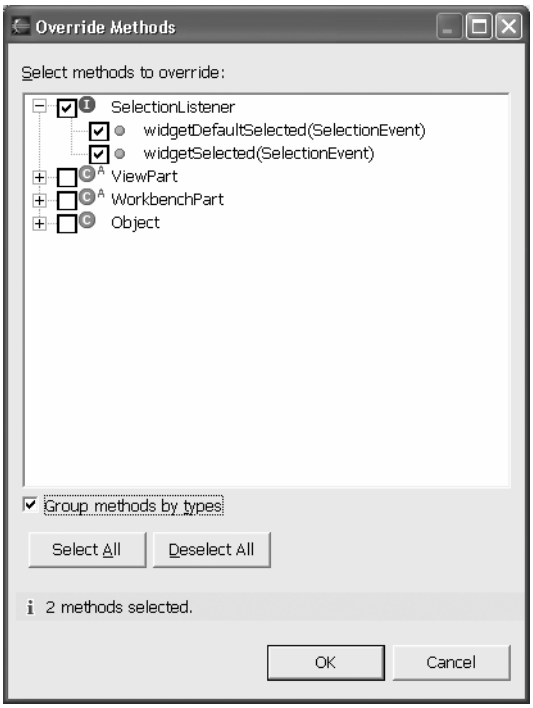


Figure 2-1
Method override selection dialog [swt_01.tif]

Note: The added method stubs are added but not compiled. You must save the file SWTView.java to compile them.

2. Add the following field to the class:

```
private Shell workbenchShell;
```

3. The `createPartControl` method will accept the creation of widgets. Add the following statement:

```
workbenchShell =
  PlatformUI.getWorkbench().getActiveWorkbenchWindow().getShell();
```

We will need access the workbench shell as a parameter in the `FileDialog` class later in the exercise. The static method `getWorkbench` in the `PlatformUI` class provides a link to the shell. The `createPartControl` method passes a `Composite` object (representing our view container). This `Composite` object is the parent to all the widgets that we will be creating.

Create a method with the following signature:

```
void open(Composite parent) {}
```

The `Composite` object comes from `createPartControl` and is our reference to the view container.

Add the following statements to the `open` method in order to provide a layout manager for our controls. This defines a single column grid layout.

```
parent.setLayout(new org.eclipse.swt.layout.GridLayout());
parent.setLayoutData(new GridData(GridData.GRAB_VERTICAL));
```

4. Invoke this method at the end of the `createPartControl` method:

```
open(parent);
```

5. Create a method `void createButtons(Composite parent)`. In this method, create a group to hold the pushbuttons. Make sure that you have set the layout information for your group, otherwise your buttons will not be visible. Add two push buttons to the group. The first button should have the text, “Show Messagebox”. The second button should have the text, “Show File Dialog”.

```
Group group = new Group(parent, SWT.NONE);
group.setLayout(new GridLayout());
group.setText("Buttons");
group.setLayoutData
  (new GridData(GridData.GRAB_HORIZONTAL | GridData.HORIZONTAL_ALIGN_FILL));
Button b1 = new Button(group, SWT.PUSH);
b1.setText("Show Messagebox");
b1.setFocus();
Button b2 = new Button(group, SWT.PUSH);
b2.setText("Show File Dialog");
```

6. In the `open` method, call `createButtons`. Add these statements to the end of the `open` method.

```
new Label(parent, SWT.NONE); //vertical spacer
createButtons(parent);
```

7. Since we want the `SWTView` class to be notified when the button has been pressed, we need to add this class as a `SelectionListener` for the button. Use the `addSelectionListener` method and pass in `this` as the parameter. Add the following line of code (bold) to the `createButtons` method after `b1.setFocus`.

```
Button b1 = new Button(group, SWT.PUSH);
b1.setText("Show Messagebox");
b1.setFocus();
b1.addSelectionListener(this);
```

8. This listener interface requires you to implement two methods:

   - `void widgetSelected(SelectionEvent e)`
   - `void widgetDefaultSelected(SelectionEvent e)`

   These methods were generated earlier. You just have to fill in the body of the method. When the button is selected, a message dialog will be displayed. We must add the following code to the `widgetSelected` method. The other method is ignored since we don't care about a default selection on a button. The first parameter we are passing is a `null` instance of a `Shell` object, which is unnecessary in this instance.

```
MessageDialog.openInformation(
  null,
  "SWTView",
  "Hello World! ");
```

9. When button `b2` is selected, we will display a `FileDialog` with the style set to `SWT.OPEN`. Print the filename selected to the console. Just for variety, use an anonymous inner class to implement the `widgetSelected` method. Modify the `createButtons` method by adding the code below after `b2.setText("Show File Dialog")`. Note that `FileDialog` does require a `Shell` object as the first parameter.

```
b2.addSelectionListener(new SelectionAdapter() {
  public void widgetSelected(SelectionEvent e) {
    FileDialog dialog = new FileDialog(workbenchShell, SWT.OPEN);
    dialog.setFilterExtensions(new String[] { "*.jar;*.zip" });
    String selectedFile = dialog.open();
    System.out.println("The file you chose is " + selectedFile);
 }
});
```

10. Test your application by starting up your test workbench from this PDE project. Open view **Lab: SWT > Lab: SWT Lab** (use **Window > Show View > Other…**). You can double click on the view title bar so that the view is the only visible frame (double clicking again returns to multi-frame mode). It should look like Figure 2-2.

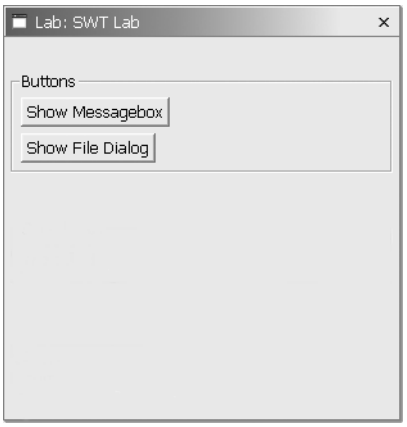


Figure 2-2
SWT Lab Panel [swt_02.tif]

Press the **Show Messagebox** button to see a message pop-up. Press the **Show File Dialog** button and select a file. Look at the **Console** view in the development workbench instance to see the file selected.

Part 2: Add StyledText

1. `StyledText` is class that supports many attributes of what is commonly known as “rich text”. For instance, you can set color and font of a `StyledText` field.

   Add a private field in the `SWTView` class of type `StyledText`.

```
private StyledText myText;
```

2. Create a method called `createStyledText(Composite parent)` and put the text control in its own group box. Set the initial text to “abcdefg”.

```
public void createStyledText(Composite parent) {
  new Label(parent, SWT.NONE); //vertical spacer
  Group group2 = new Group(parent, SWT.NULL);
  group2.setLayout(new GridLayout());
  group2.setText("Styled Text");
  group2.setLayoutData(
    new GridData(GridData.GRAB_HORIZONTAL | GridData.HORIZONTAL_ALIGN_FILL));
  new Label(parent, SWT.NONE);
  myText = new StyledText(group2, SWT.SINGLE|SWT.BORDER);
  myText.setText("abcdefg");
}
```

3. In the `open` method, add the call to `createStyledText`. Insert it at the end of the method.

```
new Label(parent, SWT.NONE); //vertical spacer
createStyledText(parent);
```

4. Add the code to change the text when the button, `b1`, is pressed. Modify the `widgetSelected(SelectionEvent)` method to set the text to “Hello World” after the message dialog is displayed. The line of code to add at the end of the method is:

```
myText.setText("Hello World");
```

5. To learn about another listener interface, add a `ModifyListener` to the end of the `createStyledText` method. The method you need to implement for this interface is `void modifyText(ModifyEvent e)`. To see when this event is being generated, print a message “Got new text” to the console when you receive the event. Set the size and redraw the text when you get this event.

```
myText.addModifyListener(new ModifyListener() {
  public void modifyText(ModifyEvent e) {
    myText.setSize(90,25);
    System.out.println("Got new text");
  }
});
```

6. Let's observe another listener. Also add a `ControlListener` to the code in `createStyledText`. Implement the `void controlResized(ControlEvent e)` method. Print out a message stating that the control has been resized.

```
myText.addControlListener(new ControlListener() {
  public void controlResized(ControlEvent e) {
    System.out.println("Control Resized");
  }
  public void controlMoved(ControlEvent e) {}
});
```

7. Before we finish let's assign a font to this `StyledText` widget to illustrate widget disposal. A `Font` object must be disposed since it does not directly belong to the widget hierarchy of the view (the parent `Composite` object in this case).

   a. Add the font to the class:

```
private Font font;
```

Update the `createStyledText` method to assign a bold, Courier font to the myText field.

```
font = new Font(myText.getDisplay(), new FontData("Courier",14,SWT.BOLD));
myText.setFont (font);
```

Lastly, we want to dispose of the font. We will override the dispose method of the view so that when it gets disposed, the font does, too. Select the class in the outline view and select **Override Methods** in its context menu. In the dialog, select the class `WorkbenchPart` then `dispose`. The method will be added to your code. Add this line of code to the dispose method.

```
font.dispose();
```

Set a breakpoint on this line. The class `FontRegistry` is something you might want to investigate if you are using many fonts. It handles disposal for you.

8. Test your application using the debugger. You should see a view like Figure 2-3. Click on the buttons and look at the Console view in the development Workbench instance to see the results of your code. Close the view. Did you stop at your breakpoint? Reopen the view and then close your test instance of the workbench. Did you stop at your breakpoint again?

Figure 2-3
SWT lab panel [swt_03.tif]

Part 3: Add a List

1. Go to the class definition and add a static array of strings. The following code should be added to the class definition.

```
static String[] ListData1 =
  {"Sherry", "Scott", "Dan", "Jim", "Pat", "John", "The Longest String"};
```

2. Create the `createList` method.

```
public void createList(Composite parent) {}
```

3. In the `createList` method, create a `List` and use the styles to specify that you want to be able to scroll and have multiple selections.

```
List myList =
  new List (parent, SWT.V_SCROLL | SWT.H_SCROLL | SWT.MULTI | SWT.BORDER);
myList.setLayoutData(new GridData());
```

4. Use the static array, `ListData1`, to initialize the contents of the list. Add the following code to the end of the `createList` method.

```
myList.setItems (ListData1);
```

5. Use `SelectionListener` to be notified when an element in the list is selected. Add this to the end of `createList`.

```
myList.addSelectionListener(
  new SelectionListener() {
    public void widgetSelected(SelectionEvent e) {
      System.out.println("List item selected");
    }
    public void widgetDefaultSelected(SelectionEvent e) {
       System.out.println("List item default selected");
    }
  }
);
```

6. Update the `open` method to invoke the `createList` method.

```
new Label(parent, SWT.NONE); //vertical spacer
createList(parent);
```

Test your application. Notice that you only receive one event on a multiple select. You should see a window like Figure 2-4:

Figure 2-4
SWT lab panel [swt_04.tif]

Part 4: Thread Synchronization (optional)

You will run the solution `com.ibm.lab.soln.swt`. Make sure it is available and open in your workspace. The solution has one more function that demonstrates synchronization between the Workbench running in the SWT UI thread and another thread.

1. Select the project and start a test instance of Eclipse using menu **Run > Run As > Run-time Workbench**.

2. From the test instance of Eclipse, open the **Soln: SWT Lab** view. Open it using **Window > Show View > Other… > Soln: SWT > Soln: SWT Lab**. Expand the view and press the **Fill Progress Bar** button. The `Display.asyncExec` method updates the progress bar from the user thread. The `Display.syncExec` method displays the dialog half way through the process. The code is in `SWTView.createProgressBar` method. Information is written to the Console view of the Eclipse host instance during this operation.

Exercise Activity Review

What you did in this exercise:

- Used SWT controls to create a simple user interface
- Experimented with SWT event handling

Exercise 3 Defining a New Project Wizard

This exercise takes you through the process of defining a simple wizard to understand the basics, and then converting the wizard so it creates a project with customized content.

Introduction

Wizards are useful user interface components in the Eclipse workbench. Your tool may just choose to define wizards that are associated to one of the wizard extension points defined by the platform or also use wizards in other places in your tool.

Exercise Concepts

The exercise begins with a plug-in project as generated by the PDE, with the addition of one action extension, which will be used to show how you can open a wizard directly.

You will begin by building a simple wizard, test it, and then use the action provided to open it directly. test this existing code, then build a JFace component that includes a viewer, content provider, and label provider. This implementation will expose a portion of the viewer API so that the JFace component can accept an input, add selection change listeners, use a predefined table in the viewer, and allow alternate content provider and label provider implementations to be identified.

Skill Development Goals

This exercise looks at the definition and use of Wizards so that you can understand:

- How wizards and wizards pages are defined
- How you can open a wizard directly
- Where wizards exist when defined as part of a platform extension point
- How you can build on the existing wizard and wizard page framework in Eclipse

Exercise Setup

Before you can begin this exercise, you must have access to the `com.ibm.lab.newWizard` template plug-in project. This project should already have been imported into your workspace. If not, import the plug-in from the exercise templates location on your workstation.

Primary Exercise Instructions

The goal is to implement a wizard that creates a new project. At this point the project will not have any special attributes (such as a project nature), but it will be created such that it does have a predefined folder which contains two files. One file created with dynamic content and one created based on the content of a file in the plug-in's runtime/install directory. The process of getting there will include the definition of a simple wizard and opening the wizard directly in a menu action.

Part 1: Implement A Simple Wizard

The first task is to define and create a simple wizard. First, you add the extension definition, and then you create a wizard class by letting it be generated by the PDE.

Step 1. Define New Wizard Extension

Edit the plugin.xml and add the view extension, this includes a category for the view (how it will be found in the Show View dialog) and the view itself.

1. Edit the `plugin.xml` file in the `com.ibm.lab.newWizard` project you imported earlier. Select the Extensions tab. You will now specify the information needed for the View extension.
2. Define the wizard extension.

   Select the **Add…** button. Select **Generic Wizards > Schema-based Extensions**. Press **Next**. Scroll down the list, select the extension point for new wizards, **org.eclipse.ui.newWizards**, and Press **Finish**.
3. Specify the category for this new wizard.

   Select the **org.eclipse.ui.newWizards** entry in the extensions list and press Finish. Choose the **New > Category** context menu. Using the Properties view, modify the id for the category to be **com.ibm.lab.newWizard.category** and the name for the category to be **Edu: Wizards**. The parent category will be blank.
4. Define the new wizard

   Select the **org.eclipse.ui.newWizards** entry in the extensions list and choose the **New > Wizard** context menu.

   Using the Properties View, specify the following:

   - category: `com.ibm.lab.newWizard.category`
   - icon: `icons\CustomNature.gif`
   - id: `com.ibm.lab.newWizard.wizard`
   - name: `Edu: Custom Project`
   - project: `true`

   Ignore the class property for now; you will specify a value later.
5. Save the `plugin.xml` file. The XML for the new wizard should look like this in the Source page:

```
<extension
      point="org.eclipse.ui.newWizards">
   <category
         name="Edu: Wizards"
         id="com.ibm.lab.newWizard.category">
   </category>
   <wizard
         name="Edu: Custom Project"
         icon="icons\CustomNature.gif"
         category="com.ibm.lab.newWizard.category"
         class="com.ibm.lab.newWizard.NewWizard1"
         project="true"
         id="com.ibm.lab.newWizard.wizard">
   </wizard>
</extension>
```

Step 2. Generate Wizard

The PDE can generate classes for many extension types. You will now use the PDE to generate a class for your wizard.

1. Return to the extensions page of the `plugin.xml` file to generate the wizard class using the PDE and select the Edu: Custom Project (wizard) entry in the list.
2. In the Properties View, generate the class by selecting the continuation entry (...) in the class field.

   In the **Java Attribute Editor**, specify that you want to generate a new Java class. The class name is `CustomProjectWizard` and you want to let the wizard (see, you can use them anywhere) open an editor on the class after it is created. Leave the source folder and package name at their default settings.
3. Select **Finish** to generate the class.

   When complete, the **Java Attribute Editor** (which is also a wizard) will open an editor open on the `CustomProjectWizard` class. The required `init()`and `performFinish()` methods were generated by the **Java Attribute Editor** wizard.
4. Save the `plugin.xml` file.

Step 3. Define a Wizard Page and Add it to the Wizard

The wizard is a controller for the wizard pages it will display. You need at least one wizard page before you have a user interface for the wizard.

1. Use the new class wizard to create a new class in the existing package. When creating the class you should:
   - Name it `CustomProjectFilePage`
   - Extend `org.eclipse.jface.wizard.WizardPage`
   - Select “Constructors from the superclass”
   - Select “Inherited abstract methods”

   Select **Finish** to generate the class.

   When complete the wizard will open an editor open on the `CustomProjectFilePage` class. The wizard generated the required `createControl()` method and two constructors.

2. Define the user interface for the wizard page by adding these two fields to the class and making these changes to the `createControl()` method:

```
public Button loggingFile;
public Button readmeFile;
…
public void createControl(Composite parent) {
  // Build page ui
  Composite pageui = new Composite(parent, SWT.NONE);
  FillLayout fillLayout = new FillLayout();
  fillLayout.type = SWT.VERTICAL;
  pageui.setLayout(fillLayout);
  setControl(pageui);

  // populate pageui with required controls
  Composite buttonui = new Composite(pageui, SWT.NONE);
  RowLayout rowLayout = new RowLayout();
  rowLayout.type = SWT.VERTICAL;
  buttonui.setLayout(rowLayout);

  loggingFile = new Button(buttonui, SWT.CHECK);
  loggingFile.setText("Create change log file");
  loggingFile.setSelection(true);

  readmeFile = new Button(buttonui, SWT.CHECK);
  readmeFile.setText("Create readme file");
  readmeFile.setSelection(true);
}
```

Note: Copy the method above from the `JPages\Part1_Wizard.jpage` file.

Use the **Source > Organize Imports** option (**Ctrl+Shift+O**) to add the required import statement(s), be sure to choose the `org.eclipse.swt.widgets.Button` import.

3. Add the wizard page to the wizard by adding a field and the `addPages()` method to the `CustomProjectWizard` class. This field and method logic should be added:

```
private CustomProjectFilePage filePage;
…
public void addPages() {
  filePage =
    new CustomProjectFilePage("filePage", "Generated Project Files", null);
  filePage.setDescription("Select the files to be added to the project.");
  addPage(filePage);
}
```

Note: Copy the logic above from the `JPages\Part1_Wizard.jpage` file.

4. Customize the `performFinish()` method in the `CustomProjectWizard` class so that it returns `true` and allows the wizard to close after the Finish button is pressed. The method logic should also indicate what options were chosen on the wizard page.

   Make the `performFinish()` method look like this:


```
public boolean performFinish() {
  // Identify choices made on the filePage
  boolean log = filePage.loggingFile.getSelection();
  boolean readme = filePage.readmeFile.getSelection();

  // Show choices from filePage
  MessageDialog.openInformation(
     null,
     "File Selections Made",
     "Logging file is " + log + "\n Readme file is " + readme);
   return true;
}
```

Note: Copy the method above from the `JPages\Part1_Wizard.jpage` file.

Use the **Source > Organize Imports** option (**Ctrl+Shift+O**) to add the required import statement(s) and then save the Java source.

5. Launch the runtime workbench to test the wizard user interface.

   Open the New dialog: **File > New > Project…**and then select **Edu: Wizards > Edu: Custom Project**. Then click Next to enter the wizard.

   You should see a wizard page that looks like this:

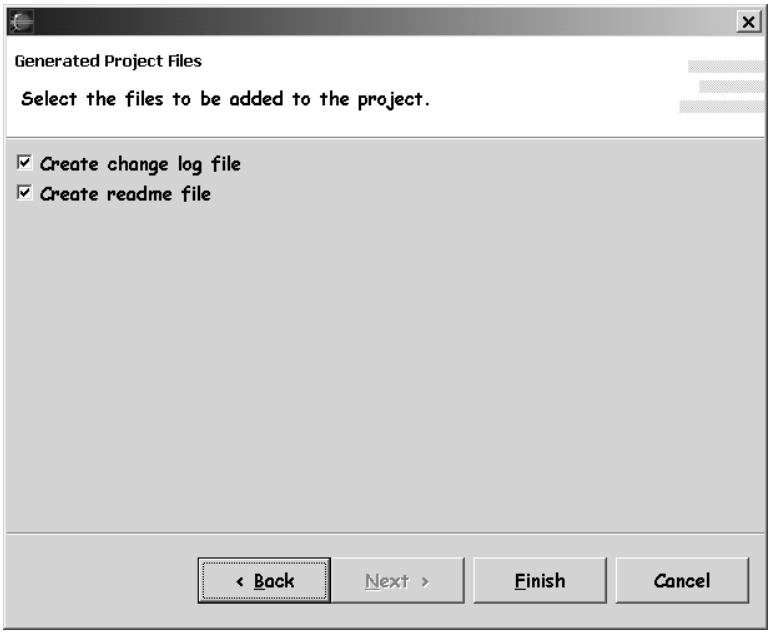


Figure 3-1
CustomProjectFilePage Displayed in CustomProject

You now have a working wizard. If you select one or both toggles and then click on Finish you will be told what was selected and the wizard will close.

Close the runtime workbench after testing is complete.

Step 4. Open the Wizard from an Action

You defined a wizard by extension, but if your wizard class just extended `Wizard` and implemented the `IWizard` interface, you could use your wizard anywhere. An action contribution extension and the associated class have been defined for you in the template project. You can customize this action to open the wizard you just defined.

1. Modify the `run()` method in the `OpenWizardAction` class so that the action will open the wizard you just defined. Modify the `run()` logic to look like this:

```
public void run(IAction action) {
  Shell shell =
    PlatformUI.getWorkbench().getActiveWorkbenchWindow().getShell();

  // Create the wizard
  CustomProjectWizard wizard = new CustomProjectWizard();

  // Create the dialog to wrap the wizard
  WizardDialog dialog = new WizardDialog(shell, wizard);

  dialog.open();
}
```

   Note: Copy the method above from the `JPages\Part1_Wizard.jpage` file.

   Use the **Source > Organize Imports** option (**Ctrl+Shift+O**) to add the required import statement(s) and then save the Java source.

2. Launch the runtime workbench to test the wizard user interface.

   Open the wizard using the **Edu: Actions > Open Wizard** menu option. If this option is not visible, reset the perspective (**Window > Reset Perspective**).

   You should see a wizard page that looks just like the one shown earlier.

You have now defined a working wizard, and seen how the workbench will find an open a wizard defined as an extension. You have also learned how you can open your own wizards from anywhere you might need one in your tool.

Part 2: Revise the Wizard so that it Adds a New Project to the Workspace

The current wizard just shows one wizard page and reports selection status. The goal is to create a wizard that will create a new project and customize the project with files in a predefined folder.

You will begin by modifying the existing wizard so that it incorporates the existing new project wizard function provided by the platform and customizing it to include the wizard page already defined with control logic that actually adds files to the project once it has been created.

Step 1. Add the Reusable New Project Wizard Page to your Wizard

The platform provides reusable wizard pages. You will add the reusable wizard page that supports the creation of a basic project to your wizard. This same page is seen when you use the existing function found in the user interface (**New > Project… > Simple > Project**).

1. Start by adjusting the `addPages()` method in the `CustomProjectWizard` class. It must add the `projectPage` before the `filePage`, and to ensure that the `filePage` is visited, its page complete state must be set to false.

   Add the field shown and modify the `addPages()` method so that it looks like this:

```
private WizardNewProjectCreationPage projectPage;
…
public void addPages() {

  projectPage = new WizardNewProjectCreationPage("projectPage");
  projectPage.setTitle("Create a New Project");
  projectPage.setDescription(
    "Enter name and optional custom location for a project");
  addPage(projectPage);

  filePage =
    new CustomProjectFilePage("filePage", "Generated Project Files", null);
  filePage.setDescription("Select the files to be added to the project.");
  filePage.setPageComplete(false);
  addPage(filePage);
}
```

   Note: Copy the logic above from the `JPages\Part2_Wizard.jpage` file.

2. Once the `filePage` has been visited, the page complete state needs set to true so the wizard can be finished. Override the inherited `setVisible()` method by adding this method to the `CustomProjectFilePage` class:

```
public void setVisible(boolean visible) {
 super.setVisible(visible);
  setPageComplete(true);
}
```

   Note: Copy the method above from the `JPages\Part2_Wizard.jpage` file.

3. Launch the runtime workbench to test the wizard user interface.

   Open the New dialog: **File > New > Project…**and then select the Edu: Wizards in the left column and then Edu: Custom Project in the right. Then click Next to enter the wizard.

   The new first page of the wizard looks like this:

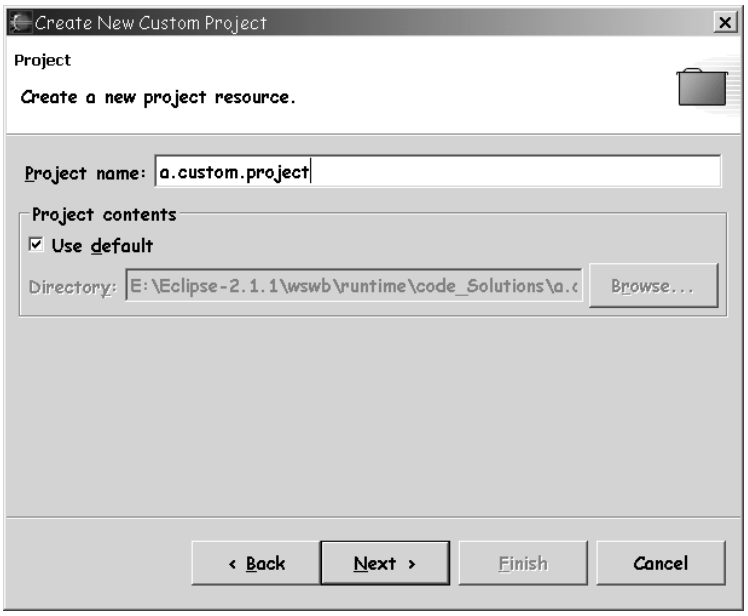


Figure 3-2
Custom New Project Wizard

You now have a working multi-page wizard. The Finish button will not be enabled until you visit the last page (`filePage`). When you do press Finish, a message dialog will be shown, but that is all. The logic to create and then customize the project has not been integrated.

Close the runtime workbench after testing is complete.

Step 2. Implement Create New Project Wizard Finish Logic
The wizard first needs to create a project using the values defined on the `projectPage`.

1. In the `CustomProjectWizard` class, select these statements in the `performFinish()` method:

```
// Show choices from filePage
MessageDialog.openInformation(
   null,
   "File Selections Made",
   "Logging file is " + log + "\n Readme file is " + readme);
```

Once the statements have been selected use the **Refactor > Extract method…** context menu option to create a new method. Name the new method `customizeProject()`.

The extracted method should look like this:

```
private void customizeProject(boolean log, boolean readme) {
  // Show choices from filePage
  MessageDialog.openInformation(
    null,
    "File Selections Made",
    "Logging file is " + log + "\n Readme file is " + readme);
}
```

2. Adjust the `performFinish()` method so that it first obtains the project and project location reference from the projectPage and then invokes a new method, `createProject()`, with these values passed as parameters to actually create the project. The `createProject()` method does not exist yet, so ignore the associated compile error. The `customizeProject()` method (created in the previous task) is invoked if the project exists.

   The modified `performFinish()` method should look like this:

```
public boolean performFinish() {

  // Get project from projectPage
  IProject newProject = projectPage.getProjectHandle();

  // Get project location as required
  IPath projectLoc = null;
  if (!projectPage.useDefaults())
     projectLoc = projectPage.getLocationPath();

  // Create project using custom location if provided
  createProject(newProject, projectLoc);


  // Identify choices made on the filePage
   boolean log = filePage.loggingFile.getSelection();
   boolean readme = filePage.readmeFile.getSelection();

  // If project exists, customize it
  if (newProject.exists())
     customizeProject(log, readme);

  //    If project exists, return true, if not return false
  if (newProject.exists())
     return true;
  else
     return false;

}
```

   Note: Copy the method above from the `JPages\Part2_Wizard.jpage` file.

   Use the **Source > Organize Imports** option (**Ctrl+Shift+O**) to add the required import statement(s).

3. Use the quick fix support (**Ctrl+1** while the cursor is on the `createProject()` error marker) to create a method named `createProject()`. Once done, you need to customize the method so it creates a project.

   This requires that you first create a project description. The description defines the project location using either the default file system location (project folder is in the workspace directory) or using the location value that was defined on the `projectPage`. The project is then created using the description and then opened.

   The completed `createProject()` method should look like this:

```
private void createProject(IProject newProject, IPath projectLoc) {

  // Create project description
  IProjectDescription projectDesc =
   NewWizardPlugin.getWorkspace().newProjectDescription(newProject.getName());

  projectDesc.setLocation(projectLoc);

  // Create project
  try {
    newProject.create(projectDesc, null);
    newProject.open(null);

  } catch (CoreException e) {
    // Auto-generated catch block
    e.printStackTrace();
  }
}
```

   Note: Copy the method above from the `JPages\Part2_Wizard.jpage` file.

   Use the **Source > Organize Imports** option (**Ctrl+Shift+O**) to add the required import statement(s) and then save the Java source.

4. Launch the runtime workbench to test the wizard logic.

   Open the New dialog: **File > New > Project…**and then select the Edu: Wizards in the left column and then Edu: Custom Project in the right. Then click Next to enter the first wizard page, the `projectPage` (the field name in the `addPages()` method).

   Enter a project name, optionally a location other than the default, click Next to enter the `filePage`, and then Finish. The logic added to the method will add a new project to the workspace.

   Close the runtime workbench after testing is complete.

Step 3. Implement Create Folder and Files Wizard Finish Logic
The wizard now creates a project, but you still need to have it react to the choices made on the `filePage` and create the required folder and requested files. The folder will always be created while file creation will depend on the selections made on the `filePage`.

1. Customize the signature and logic for the existing `customizeProject()` method so that it looks like this:

```
private void customizeProject(IProject project, boolean log, boolean readme) {

  // Create Folder
  IFolder readmeFolder = ProjectCustomizer.createFolder(project, "readme");

  // Create Logging File (if required)
  if (log)
    ProjectCustomizer.createChangeLogFile(project, "Change_log.txt");

  // Create Readme File from template (if required)
  if (readme) {
    IPath template = new Path("readme_template/readme_file_template.readme");
    String fileName = project.getName() + ".readme";
    ProjectCustomizer.copyTemplate(readmeFolder, fileName, template);
  }

}
```

   Note: Copy the method above from the `JPages\Part3_Wizard.jpage` file.

   Use the **Source > Organize Imports** option (**Ctrl+Shift+O**) to add the required import statement(s). When prompted, select the `org.eclpise.core.runtime.Path` option. You will correct the compile error in the next task.

   The `ProjectCustomizer` class was provided in the template project. Review the methods in this class to see how the project reference and passed parameters are used to create the required resources. A folder, two files, and a bookmark marker are created if you select both options.

   Note: One of the files that are created comes from a file in the plug-in's source directory. Can you tell which one?

2. Adjust the invocation of the `customizeProject()` method in the `performFinish()` method to add the `newProject` parameter which was added to the method signature.
3. Launch the runtime workbench to test the modified wizard finish logic.

   Enter a project name, and then go to the end and select Finish. The project should be created and customized based on the selected options on the `filePage`. The new project should look something like this in the Navigator view:

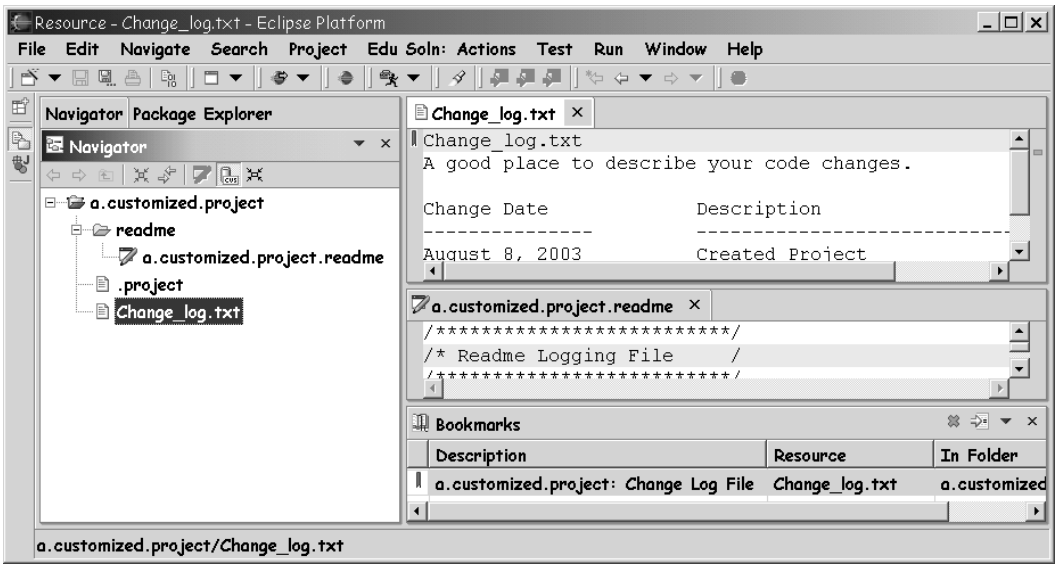


Figure 3-3
Structure of Customized Project

Close the runtime workbench after testing is complete.

You are done with the basics. If you have time, or want to return here later, you can do the optional portion of this exercise.

Optional Exercise Instructions

The optional portion of this exercise allows you to learn more about how to manage resource events (so you have a well performing plug-in), provide task feedback using the wizard monitor, and may the flow of the wizard dynamic.

Optional Part 1: Adjust the Project Customization Process to Limit Resource Events

The wizard you defined creates a project and customizes it as required. It works, and this is not bad, but it can be better.

How? Put your performance hat on. Remember the discussion about resource change events? Each resource creation or modification action in the `createProject()` and `customizeProject()` methods triggers a resource change event. Let's start by looking at this in a bit more detail.

Step 1. Trace the Resource Change Events Triggered by the Wizard
A resource change listener has been provided as part of the project template. By enabling this listener, it will report changes that have occurred as messages in the console log.

1. Change the `NewWizardPlugin` class so that the provided resource change listener is added when the plug-in starts. Modify the `startup()` method; uncomment the `addRCL()` method invocation:

```
public void startup() throws CoreException {
  // Let super do the normal startup work
  super.startup();
  // The resource change listener will be added if/when this method is invoked
  addRCL();
}
```

2. Test the new code by starting the runtime workbench and creating a new custom project.

   The resource change listener that was provided will print information about each resource event that occurs to the Console view. The following would be listed for the creation of the customized project:

```
RCE: 1
RCE: 1 -> Event triggered...
RCE: 1 -> Resource has been changed.
RCE: 1 ->   Resource / has changed.
RCE: 1 ->   Resource /custom.project was added.
RCE: 1 ->   Resource /custom.project/.project was added.
RCE: 2
RCE: 2 -> Event triggered...
RCE: 2 -> Resource has been changed.
RCE: 2 ->   Resource / has changed.
RCE: 2 ->   Resource /custom.project has changed.
RCE: 2 ->  ResourceDelta Kind: Opened
RCE: 3
RCE: 3 -> Event triggered...
RCE: 3 -> Resource has been changed.
RCE: 3 ->   Resource / has changed.
RCE: 3 ->   Resource /custom.project has changed.
RCE: 3 ->   Resource /custom.project/readme was added.
RCE: 4
RCE: 4 -> Event triggered...
RCE: 4 -> Resource has been changed.
RCE: 4 ->   Resource / has changed.
RCE: 4 ->   Resource /custom.project has changed.
RCE: 4 ->   Resource /custom.project/Change_log.txt was added.
RCE: 5
RCE: 5 -> Event triggered...
RCE: 5 -> Resource has been changed.
RCE: 5 ->   Resource / has changed.
RCE: 5 ->   Resource /custom.project has changed.
RCE: 5 ->   Resource /custom.project/Change_log.txt has changed.
RCE: 5 ->   ResourceDelta Kind: Marker Change
RCE: 5 ->      Marker delta kind: Added
RCE: 5 ->      Marker type: org.eclipse.core.resources.bookmark
```



```
RCE: 6
RCE: 6 -> Event triggered...
RCE: 6 -> Resource has been changed.
RCE: 6 ->   Resource / has changed.
RCE: 6 ->   Resource /custom.project has changed.
RCE: 6 ->   Resource /custom.project/Change_log.txt has changed.
RCE: 6 ->   ResourceDelta Kind: Marker Change
RCE: 6 ->      Marker delta kind: Changed
RCE: 6 ->      Marker type: org.eclipse.core.resources.bookmark
RCE: 6 ->      Marker content: [1]
RCE: 7
RCE: 7 -> Event triggered...
RCE: 7 -> Resource has been changed.
RCE: 7 ->   Resource / has changed.
RCE: 7 ->   Resource /custom.project has changed.
RCE: 7 ->   Resource /custom.project/Change_log.txt has changed.
RCE: 7 ->   ResourceDelta Kind: Marker Change
RCE: 7 ->      Marker delta kind: Changed
RCE: 7 ->      Marker type: org.eclipse.core.resources.bookmark
RCE: 7 ->      Marker content: [1, 10]
RCE: 8
RCE: 8 -> Event triggered...
RCE: 8 -> Resource has been changed.
RCE: 8 ->   Resource / has changed.
RCE: 8 ->   Resource /custom.project has changed.
RCE: 8 ->   Resource /custom.project/Change_log.txt has changed.
RCE: 8 ->   ResourceDelta Kind: Marker Change
RCE: 8 ->      Marker delta kind: Changed
RCE: 8 ->      Marker type: org.eclipse.core.resources.bookmark
RCE: 8 ->      Marker content: [11, 10, 1]
RCE: 9
RCE: 9 -> Event triggered...
RCE: 9 -> Resource has been changed.
RCE: 9 ->   Resource / has changed.
RCE: 9 ->   Resource /custom.project has changed.
RCE: 9 ->   Resource /custom.project/Change_log.txt has changed.
RCE: 9 ->   ResourceDelta Kind: Marker Change
RCE: 9 ->      Marker delta kind: Changed
RCE: 9 ->      Marker type: org.eclipse.core.resources.bookmark
RCE: 9 ->      Marker content: [custom.project: Change Log File, 11, 1, 10]
RCE: 10
RCE: 10 -> Event triggered...
RCE: 10 -> Resource has been changed.
RCE: 10 ->   Resource / has changed.
RCE: 10 ->   Resource /custom.project has changed.
RCE: 10 ->   Resource /custom.project/readme has changed.
RCE: 10 ->   Resource /custom.project/readme/custom.project.readme was added.
```

The first two events (`RCE:1-2`), are the creation and opening of the project. All the rest are the creation of the folder (`RCE:3`), Change_log.txt file(`RCE:4`), Change_log.txt file marker(`RCE:5`), marker attributes(`RCE:6-9`), and the .readme file(`RCE:10`). The interesting thing is that the `custom.project` project resource and associated `.project` file were created as part of one event. The rest of the project customization should also take place as part of that same event if you want a better performing tool. There is no reason to fire ten events when one will do.

Step 2. Wrap Wizard Resource Creation in a Runnable

There are several options for how to wrap a set of workspace operations to limit the number of resource change events. The workspace can process an object that implements the `IWorkspaceRunnable` interface using the `workspace.run()` method.

When performing workspace updates from a user interface you may also want to consider the use of the `WorkspaceModifyOperation` class. This class allows you to construct a runnable that has access to a progress monitor. A progress monitor can be obtained from the wizard container.

1. Create a `WorkspaceModifyOperation` structure that can be used to wrap the resource creation and modification logic in a runnable. The goal is an inner class with an `execute()` method that performs project creation and customization. You could just copy this from the JPage file, but for fun why not let the Eclipse JDT do most of the work.

   Make these edits to the `performFinish()` method:

   - Add a few empty lines before the invocation of the `createProject()` method.
   - Enter this text, `Workspacem`, and then press Ctrl+Space to trigger code completion. Choose the option `WorkspaceModifyOperation`
   - Enter `op = new` on the same line and then press Ctrl+Space and again choose the `WorkspaceModifyOperation` option.
   - Enter a left parenthesis (the JDT will generate the match) and then press Ctrl+Space again. The JDT will open a dialog to help you complete the `WorkspaceModifyOperation` inner class definition.
   - The default selection (`execute()` method) is fine so just select OK.
   - Enter a semi-colon (;) to complete the generated statement, the cursor should already be in the correct location.

   That should be all you need. The completed statement should now look like this:

```
WorkspaceModifyOperation op = new WorkspaceModifyOperation() {
  protected void execute(IProgressMonitor monitor)
    throws CoreException, InvocationTargetException, InterruptedException {
    // TODO Auto-generated method stub

  }
};
```

   This can now be used to create and customize the project.

2. Cusomize the `WorkspaceModifyOperation` structure so that it creates and customized the project.

   Move most of the first part of the existing `performFinish()` logic to the `execute()` method (the `newProject` variable stays outside). The `WorkspaceModifyOperation` structure should now look like this:


```
WorkspaceModifyOperation op = new WorkspaceModifyOperation() {
    protected void execute(IProgressMonitor monitor)
      throws CoreException, InvocationTargetException, InterruptedException {

      // Get project location as required
      IPath projectLoc = null;
      if (!projectPage.useDefaults())
        projectLoc = projectPage.getLocationPath();

      // Identify choices made on the filePage
      boolean log = filePage.loggingFile.getSelection();
      boolean readme = filePage.readmeFile.getSelection();

      // Create project using custom location if provided
      createProject(newProject, projectLoc);

      // If project exists, customize it
      if (newProject.exists())
        customizeProject(newProject, log, readme);
    }
};
```

3. Correct the error related to the `newProject` variable. Use the Quick Fix function (put cursor on field reference and press Ctrl+1 or click on the light-bulb icon) to change the `newProject` variable definition to `final`.

   Other errors will still exist, which is fine for now.

4. Add logic to the `performFinish()` method to actually run the `WorkspaceModifyOperation`. This logic should go after the `WorkspaceModifyOperation` structure but before the last `if` statement:

```
try {
  op.run(null);
} catch (InvocationTargetException e) {
  e.printStackTrace();
} catch (InterruptedException e) {
  e.printStackTrace();
}
```

   Note: Copy the logic above from the `JPages\Xtra_Part1_Wizard.jpage` file.

   The `run()` method takes the operation and passes it on to the workspace for processing.

5. Launch the runtime workbench to test the modified wizard finish logic.

   Enter a project name, and then go to the end and select Finish. The project should be created and customized based on the selected options on the `filePage`. If you were to peek at the resource change events that were triggered you would see a different scenario:


```
RCE: 1 -> Event triggered...
RCE: 1 -> Resource has been changed.
RCE: 1 ->   Resource / has changed.
RCE: 1 ->   Resource /custom.project was added.
RCE: 1 ->   ResourceDelta Kind: Opened
RCE: 1 ->   Resource /custom.project/.project was added.
RCE: 1 ->   Resource /custom.project/Change_log.txt was added.
RCE: 1 ->   ResourceDelta Kind: Marker Change
RCE: 1 ->      Marker delta kind: Added
RCE: 1 ->      Marker type: org.eclipse.core.resources.bookmark
RCE: 1 ->      Marker content: [custom.project: Change Log File, 11, 1, 10]
RCE: 1 ->   Resource /custom.project/readme was added.
RCE: 1 ->   Resource /custom.project/readme/custom.project.readme was added.
```

This is much better, the creation and customization of the project is now just one complex event.

Optional Part 2: Run the WorkspaceModifyOperation with a Monitor

When you directly run the operation you passed a null monitor in the `run()` method. It might be nice to have a monitor so that you could show progress through the operation and possibly support a cancel request if this was truly a long-running activity. You will add support for a monitor, but we will not worry about a cancel request.

Step 1. Provide a Monitor to the WorkspaceModifyOperation Instance

You can use the wizard framework to run this operation and reuse the monitor available in the wizard.

1. Customize the invocation of the operation to let the wizard (actually, the container for the wizard) invoke the runnable.

   In the `performFinish()` method, replace this statement:

   ```
   op.run(null);
   ```

   With this statement:

   ```
   getContainer().run(false, false, op);
   ```

   The operation will be invoked, but now it will have access to a monitor (as opposed to a null) in the operation's `run()` method because you are now using the wizard dialog to run the operation.

   The `Wizard getContainer().run()` method allows you to choose if the operation should be run on a new thread (where UI widgets would not be visible) and if it can be canceled. Ultimately, the operation is passed on to the workspace for processing.

   Note: If you changed the run() method so that the code wrapped in a `WorkspaceModifyOperation` was not running on the UI thread (`run(true, false, op)`) there would be no direct access to the SWT widgets. You would have to move this logic out of the `execute()` method logic for the operation and make the variables `final`.

   You can now customize the logic used to create and customize the project to actually accept and use the monitor during the required processing steps.

2. Modify the `CustomProjectWizard init()` method so that it contains this logic:

```
public void init(IWorkbench workbench, IStructuredSelection selection) {
  setNeedsProgressMonitor(true);
}
```

   This is required to tell the wizard container that you want to see (and use) the progress monitor as part of the wizard dialog.

3. Modify the `WorkspaceModifyOperation` defined in the `performFinish()` method to initialize the task being monitored and pass the monitor instance to the `createProject()` and `customizeProject()` methods. The revised `execute()` method logic should look like this:

```
protected void execute(IProgressMonitor monitor)
  throws CoreException, InvocationTargetException, InterruptedException {

  monitor.beginTask("Create Customized Project:", 3000);
  monitor.subTask("prepare");

  // Get project location as required
  IPath projectLoc = null;
  if (!projectPage.useDefaults())
     projectLoc = projectPage.getLocationPath();

  // Identify choices made on the filePage
  boolean log = filePage.loggingFile.getSelection();
  boolean readme = filePage.readmeFile.getSelection();

  monitor.worked(1000);
  ProjectCustomizer.pause(1000);

  // Create project using custom location if provided
  createProject(newProject, projectLoc,
    new SubProgressMonitor(monitor, 1000));

  // If project exists, customize it
  if (newProject.exists()) {
     monitor.subTask("customize");
     ProjectCustomizer.pause(1000);
     customizeProject(newProject, log, readme,
       new SubProgressMonitor(monitor, 1000));
  }
  monitor.done();
}
```

   Note: Copy the method above from the `JPages\Xtra_Part2_Wizard.jpage` file.

   The use of the `ProjectCustomizer.pause()` method slows down the processing so you can actually see the progress bar in the wizard react to the monitor logic during testing.

4. Modify the `createProject(…)` method signature to accept the monitor and the logic to use it during project creation. The modified method should look like this:

```
private void createProject(IProject newProject, IPath projectLoc,
    IProgressMonitor monitor) {

  monitor.beginTask("", 500);
  monitor.subTask("create project description");

  // Create project description
  IProjectDescription projectDesc =
   NewWizardPlugin.getWorkspace().newProjectDescription(newProject.getName());
  projectDesc.setLocation(projectLoc);

  monitor.worked(100);
  ProjectCustomizer.pause(1000);

  // Create project

  try {
    monitor.subTask("create project");
    newProject.create(projectDesc, monitor);
    monitor.worked(100);
    ProjectCustomizer.pause(1000);

    monitor.subTask("open the new project");
    newProject.open(monitor);
    monitor.worked(100);
    ProjectCustomizer.pause(1000);

    monitor.done();

  } catch (CoreException e) {
    // Auto-generated catch block
    e.printStackTrace();
  }
}
```

Note: Copy the method above from the `JPages\Xtra_Part2_Wizard.jpage` file.

5. Modify the `customizeProject(…)` method signature to accept the monitor and the logic to use it during project creation. The modified method should look like this:

```
private void customizeProject(IProject project,  boolean log, boolean readme,
   IProgressMonitor monitor) {

  monitor.beginTask("", 3000);

  // Create Folder
  monitor.subTask("add folder");
  IFolder readmeFolder = ProjectCustomizer.createFolder(project, "readme");

  monitor.worked(1000);
  ProjectCustomizer.pause(1000);

  // Create Logging File (if required)
  monitor.subTask("add log file");
  if (log)
    ProjectCustomizer.createChangeLogFile(project, "Change_log.txt");

  monitor.worked(1000);
  ProjectCustomizer.pause(1000);

  //  Create Readme File from template (if required)
  monitor.subTask("add readme file");
  if (readme) {
    IPath template = new Path("readme_template/readme_file_template.readme");
    String fileName = project.getName() + ".readme";
    ProjectCustomizer.copyTemplate(readmeFolder, fileName, template);
  }

  monitor.worked(1000);
  ProjectCustomizer.pause(1000);

  monitor.done();
}
```

Note: Copy the method above from the `JPages\Xtra_Part2_Wizard.jpage` file.

6. Launch the runtime workbench to test the modified wizard finish logic.

   Enter a project name, and then go to the end and select Finish. The project should be created and customized based on the selected options on the `filePage`. The f you were to peek at the resource change events that were triggered you would see a much better scenario:

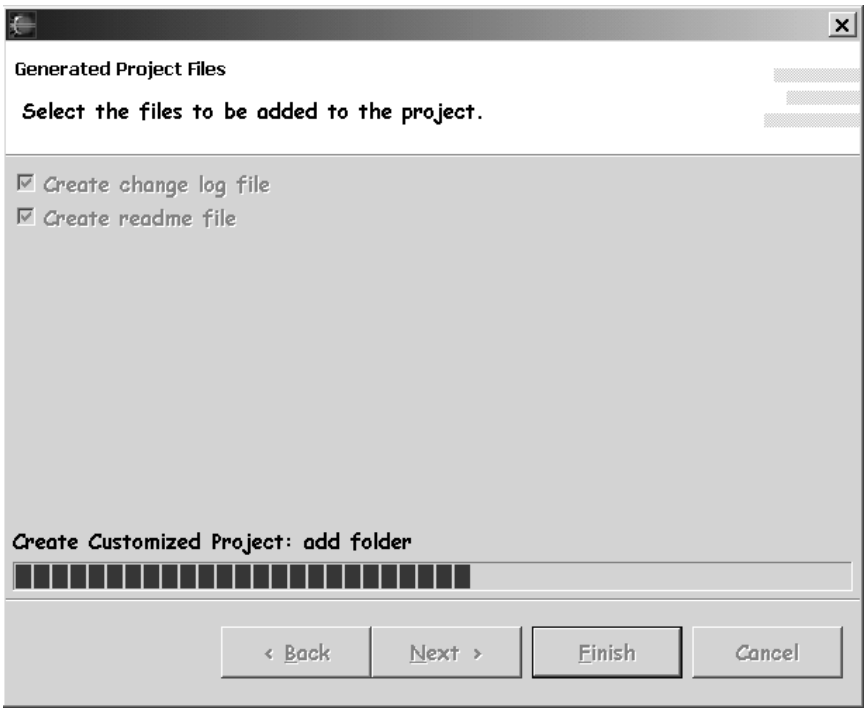


Figure 3-4
Wizard with Progress Monitor Bar

Close the runtime workbench after testing is complete.

Optional Part 3: Make the Customize Project Page and Processing Optional

The wizard you defined creates a project and customizes it as requested. If you want to make visiting the customization page optional, including the act of customization itself, then you need to change the wizard structure and add support for a dynamically defined page. This requires that you add a user interface control to the new project page to allow the user to determine if they want a customized project and adds the required page to the wizard flow if it was requested.

Step 1. Create a Customized Version of the New Project Page

The first step is to create a new project page that lets the user determine if they want to customize the project at the same time. Wizard pages such as the new project page, are both reusable and customizable.

1. Create a new class, `CustomWizardNewProjectCreationPage`, that extends the existing page. When you create this new class:

   Extend `WizardNewProjectCreationPage`

   Select the "**Constructors from superclass**" option

2. Override the `createControl()` method to add another widget to the user interface. If you begin typing in the class and enter `createc` and then press Ctrl+Space, you will have the option of generating the method stub.

   To add a toggle to the existing new project page by adding a field and customizing the method. The new code should look like this:


```
public Button needFilePage;
…
public void createControl(Composite parent) {
  // Allow superclass to create user interface
  super.createControl(parent);

  //  Additional UI controls added to control defined by super
  Composite pageui = (Composite) getControl();

  // Add required controls to existing control
  Composite buttonui = new Composite(pageui, SWT.NONE);
  RowLayout rowLayout = new RowLayout();
  rowLayout.type = SWT.VERTICAL;
  buttonui.setLayout(rowLayout);

  needFilePage = new Button(buttonui, SWT.CHECK);
  needFilePage.setText("Customize Project");
  needFilePage.setSelection(true);
}
```

Note: Copy the logic above from the `JPages\Xtra_Part3_Wizard.jpage` file.

Use the **Source > Organize Imports** option (**Ctrl+Shift+O**) to add the required import statement(s). Make sure you choose the SWT options when presented.

3. Adjust the wizard to reference the new customized new project page. This requires changes to the `projectPage` field and `addPages()` logic. Change `WizardNewProjectCreationPage` references to `CustomWizardNewProjectCreationPage`. Use the **Source > Organize Imports** option (**Ctrl+Shift+O**) to adjust the required import statement(s).

Step 2. Add the Customized Project Page On Request

Instead of adding the customizing page automatically in the wizard `addPages()` method you need to have a user control that determines when it should be added and logic that supports the request.

1. The wizard needs changed to allow access to the `filePage` field. Generate a getter method for this filed by selecting the field in the Outline view and using the **Source > Generate Getter and Setter…** context menu option. Add only the getter method.
2. Add logic to the toggle button in the new project page user interface so it can determine if the customization page must be shown as part of the wizard.

   The following logic should be added to the `createControl()` method, after the `needFilePage.setSelection(true);` statement, in the `CustomWizardNewProjectCreationPage` class:


```
needFilePage.addSelectionListener(new SelectionAdapter() {
  public void widgetSelected(SelectionEvent event) {
    CustomProjectWizard wizard = (CustomProjectWizard) getWizard();

    if (needFilePage.getSelection())
      wizard.getFilePage().setPageComplete(false);
    else
      wizard.getFilePage().setPageComplete(true);

    getContainer().updateButtons();
  }
});
```

This logic sets the `filePage` completion status based on the `needFilePage` selection. The `filePage` status was initialized to false as the `needFilePage` toggle was initialized to true. The wizard is then told to update the Next and Finish buttons so they reflect the need for a second page.

If you tested the wizard now you would be able to deselect the Customize Project option and then click Finish on the first wizard page. The project would still be customized at creation as the `performFinish()` method doesn't yet care about this selection.

3. Adjust the `performFinish()` logic in the `CustomProjectWizard` so that project customization is only performed when requested. The modified logic references the selection state before invoking the `customizeProject()` method:

```
// If project exists, customize it
if (newProject.exists() && projectPage.needFilePage.getSelection()) {
  monitor.subTask("customize");
  ProjectCustomizer.pause(1000);
  customizeProject(newProject,  log, readme,
    new SubProgressMonitor(monitor, 1000));
}
```

4. Launch the runtime workbench to test the modified wizard finish logic.

   The new project page now has a toggle which determines if the second wizard page must be visited (page complete state) and if the project should be customized:

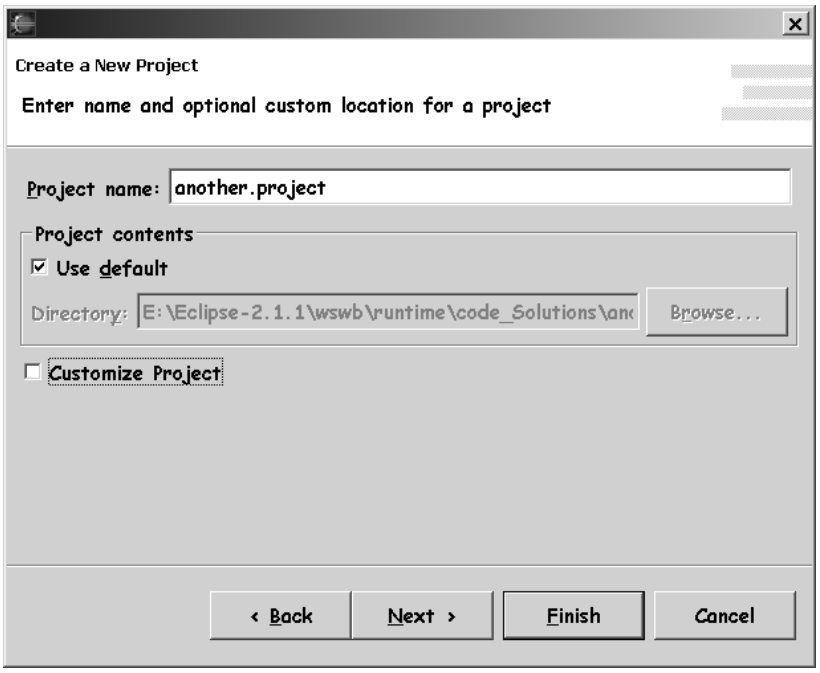


Figure 3-5
Wizard with Customization Toggle

Close the runtime workbench after testing is complete.

Step 3. Adjust Page Processing to Reflect the Need for the Customization Page
If the customizing page is optional, then it should not be part of the wizard. The page is added to the wizard in the `addPages()` method, but this does not mean it must exist in the flow of wizard processing. Since it has already been added, you will now just ignore it when appropriate by providing a customized `getNextPage()` method in the custom new project page.

1. Override the `getNextPage()` method in the `CustomWizardNewProjectCreationPage` class. The completed method should look like this:

```
public IWizardPage getNextPage() {
  if (needFilePage.getSelection())
    return super.getNextPage();
  else
    return null;
}
```

The typical `getNextPage()` method just asks the wizard for the next page (see the implementation in the `WizardPage` class. This logic only allows this to occur if the next page is required.

2. Launch the runtime workbench to test the modified wizard finish logic.

Now when you deselect the new project page toggle the Next button is disabled as there is no next page (a `null` was returned by `getNextPage()`):

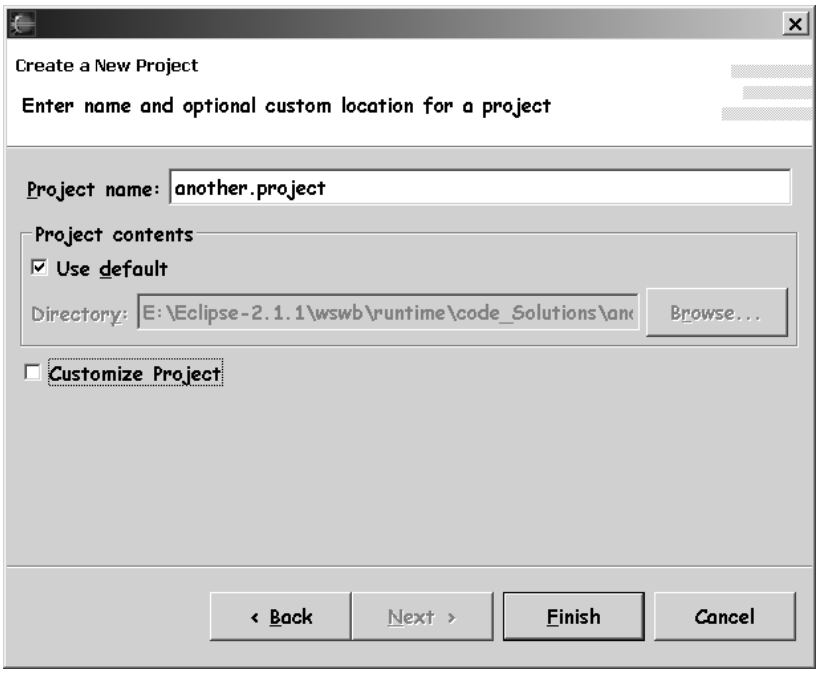


Figure 3-6
Wizard with Customization Toggle and Page Control Logic

Close the runtime workbench after testing is complete.

Exercise Activity Review

What you did in this exercise:

- Learned how to define a new wizard extension and generate a wizard class
- Implemented a wizard page with a customized user interface
- Extended a platform wizard, the new project wizard, to include your wizard page
- Extended the wizard finish logic to include the project customization processing
- Wrapped the project customization logic in a `WorkspaceModifyOperation` to manage the number of resource change events that are generated.
- Integrated the monitor provided by the wizard container with the `WorkspaceModifyOperation`
- Modified the reusable new project wizard page to customize the user interface
- Created a dynamic wizard by controlling wizard page progression based on user interaction.

Exercise 4: Implementing Preference Pages

Introduction

This exercise guides you through the process of using the dialog support in the Workbench to create value management pages for Preference values.

Exercise Concepts

During this exercise, you will implement one or more of the following:

- Plug-in definitions that support preference page extensions to the workbench.
- Persistence of data values using a preference store.

Skill Development Goals

At the end of this exercise, you should be able to:

- Add preference page extensions to a plug-in
- Define and implement a plug-in class
- Create a preference page that can save values

Exercise Setup

These setup tasks are required before you can begin this exercise:

- Setup the Workbench so that all external plugins are visible to support plugin development and testing.
- Import the com.ibm.lab.dialogs project to your workspace.

Note: You may have already performed these setup activities.

The dialogs lab template provides you with a starter plug-in (EDU Dialogs) and a base set of code, but the implementation of some code components is incomplete. You get to add new methods and references to other methods not yet invoked by the base code.

The com.ibm.lab.dialogs project contains the following files:

- `Plugin.xml` – a starter file to help you create the desired `plug-in.`
- `getstart_b.gif` – a graphic file that will be referenced by the plug-in manifest file.
- A set of Java scrapbook pages, organized by class, which contain code fragments referenced in this lab. You can use these to minimize the need to retype code defined in this document.
- Partially coded classes that will be completed during the lab.
- `EDU_plugin.xml` – plug-in definitions that you can reuse during the lab.

Other files, such as `.classpath` and `.cvsignore`, which are not used in this exercise.

Exercise Instructions

Part 1: Preference page definition

A preference page dialog is started using the **Window > Preferences** menu option available on the Workbench window. A plug-in can define any number of preference pages. The pages can be organized as a related set such that they share a navigation tree in the common preferences dialog window.

Step 1: Add preference page extensions

You will define an extension for a preference page that can be used to implement saved value support. This shows how the tool user can use the preference page to define tool specific values. These values can be used to control the logic you implement in your tool.

1. Add the following preference page plug-in extensions to the `plugin.xml` file.

   Note: You can copy these statements from the `EDU_plugin.xml` page:


```
<!-- Preferences Page1 Extension                                  -->
   <extension
  point="org.eclipse.ui.preferencePages">
  <page
    id="com.ibm.lab.dialogs.prefpage1"
    name="EDU: Basic Preference Page"
    class="com.ibm.lab.dialogs.MyBasicPreferencePage">
  </page>

   </extension>
<!-- vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv -->
```

This extension adds one page to the common preferences dialog available in the workbench.

2. Save the plugin.xml file to store the preference page definition.

Step 2: Define a preference page

You will now implement the preference page defined in the plug-in extension.

The `MyBasicPreferencePage` class will inherit the preference page processing implemented in `PreferencePage` as part of the dialog framework.

The `PreferencePage` superclass provides a default implementation of using a preference store to manage stored values. The controls you add to the property page can be initialized with default values and then use customized values obtained from the preference store.

Use of this dialog framework processing requires that you add logic to your preference page implementation class.

Note: You can use the PDE Extension page editor to generate the `MyBasicPreferencePage` class if you trigger the wizard using the drop down in the properties page for the class attribute. This is the process you followed in the `FirstPlugin` lab to generate the action class. If you use this approach you can skip to step 2.

1. Create a `MyBasicPreferencePage` class in the `com.ibm.lab.dialogs` package. This new class must extend the class:

   `org.eclipse.jface.preference.PreferencePage`

   and implement the interface:

   `IWorkbenchPreferencePage`

   Select “Inherited abstract methods” (but **do not** copy the constructors). This will add methods, such as the `createContents()` method, which will be used to customize the user interface for the preference page.

   Note: If your class does not have the required methods one or more errors will be generated. If you forget to inherit the methods of the superclass you can add them using a context menu action available in the Outline view. To quickly add the missing methods select the class in the Outline View and use the context menu option **Override Methods** to open a dialog. This dialog

will have already selected the required methods that are missing from your class so you can just click **OK**.

2. Add a Text field to the `MyBasicPreferencePage` class. This field will be referenced in methods that will be defined later: Any errors will be corrected when you add import statements in the next step.

```
private org.eclipse.swt.widgets.Text textInfo;
```

3. Customize the `createContents()` method so that you have added at least one widget to the preference page. Add these `import` statements to your class:

```
import org.eclipse.swt.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;
```

Replace the generated method with the one below to add a text widget:

```
protected Control
  createContents(org.eclipse.swt.widgets.Composite parent) {
  textInfo = new
    Text(parent, SWT.SINGLE | SWT.BORDER);
  textInfo.setText("Text String Value");
  GridData data = new
    GridData(GridData.FILL_HORIZONTAL);
  textInfo.setLayoutData(data);
  return textInfo;
}
```

4. Consider adding `System.out.println();` statements to selected methods in your preference page class to trace data values or method invocation. For example, you could add these two statements such as:

```
  System.out.println("MBPrefPage: ------> PrefPage init");

  System.out.println("MBPrefPage: ------> PrefPage createContents");
```

to the `init()` and the `createContents()` methods respectively.

5. Test your preference page class. Start the workbench (if required)

Use the Workbench **Window > Preferences** menu option to start the preferences dialog. Select the entry for your preference page (**Lab: Basic Preference Page)**. You should see the following:

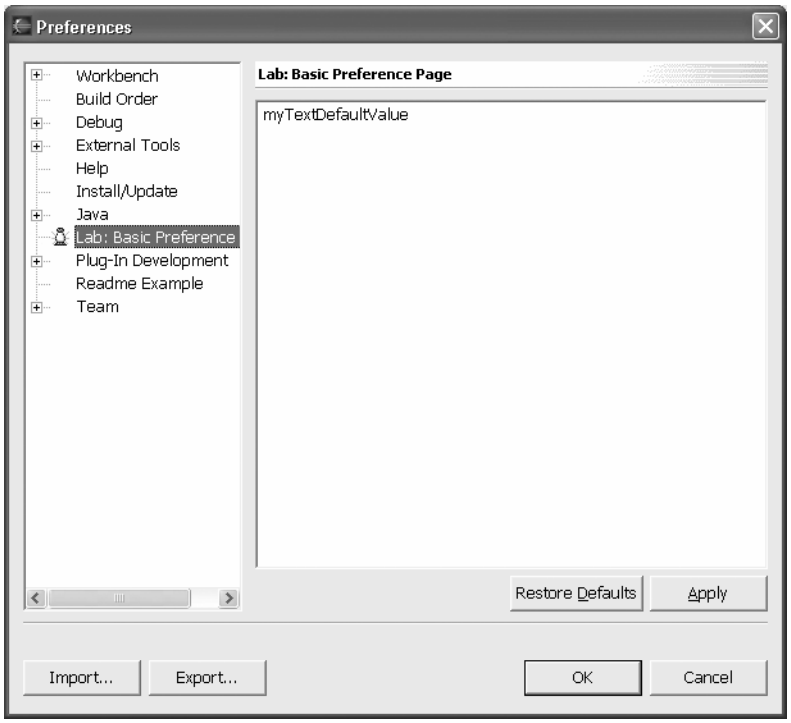


Figure 4-1
Basic Preference Panel [dialog_06.tif]

This is a custom preference page, but the value shown is hard coded (see your `createContents()` method).

Note: The **Restore Defaults, Apply, OK,** and **Cancel** buttons are added automatically, but they will not do anything as you have not yet implemented any support for the value management life cycle in your property page class.

Step 3: Adding value management life-cycle support

The dialog framework for both property pages and preference pages provides hook methods that support the life cycle processing for the values that will be shown and saved. The hook methods process the following user interactions:

- Obtain the current values
- Show default values
- Apply (save) the values
- Leave the page with either an **OK** or **Cancel** request.

By implementing the life cycle hook methods you will have a place to add the customized logic required to link the controls you created and the preference store that will be used to save values.

In this step you will add the hook methods to your preference page class.

1. Copy these methods to the `MyBasicPreferencePage` class from the helper file `EDU_MyBasicPreferencePage.jpage`:

   - `setVisible()`
   - `performDefaults()`
   - `performOk()`
   - `performApply()`
   - `performCancel()`

   Each copied method looks similar to:

```
public boolean performCancel() {
  System.out.println("PropCycle: " +
    "> in performCancel");
  return super.performCancel();
}
```

   This will allow us to follow the user interaction life cycle by reviewing the messages in the console. Note that the `setVisible()` method is triggered as the page opens and when the page selected in the preference dialog changes.

2. Start the test workbench, open the Preferences dialog, and select your preference page. Follow this process to trigger the life-cycle methods:

   - Select the Lab: Basic Preference Page (if required)
   - Select the **Restore Defaults** and **Apply** push buttons
   - Change current properties page (go to the Info page) and return (this forces the setVisible() method)
   - Select either the **OK** or **Cancel** push button

   Review the development workbench console window to see the processing life cycle as depicted by any `System.out.println()` statements you copied or added to the life cycle methods. You may want to watch the console as you interact with the preferences dialog.

Part 2: Using a plug-in to manage and store values

The best way to add saved value support to the preference page is to ask for help from a plug-in class. This requires that you have the right type of plug-in, one that extends the user interface support provided by the `AbstractUIPlugin`.

If you do not implement a real plug-in class, by subclassing `AbstractUIPlugin`, the workbench platform creates a Default plug-in for your tool. But what you want is a plug-in implementation that will provide access to the preference store services provided by the `AbstractUIPlugin` superclass.

Step 1: Find the current plug-in

To understand how plug-in processing is provided by the workbench platform, even if you do not define your own implementation, you will identify the active plug-in available to the property page.

1. Add the import statement to the `MyBasicPreferencePage` class and **highlighted** logic shown below to the `performDefaults()` method:


```
import org.eclipse.core.runtime.*;

protected void performDefaults() {
  super.performDefaults();
    System.out.println("PropCycle: "
            + "> in performDefaults");
    // get active plug-in (default or defined)
    Plugin myPlugin =
Platform.getPlugin("com.ibm.lab.dialogs");
    System.out.println("MBPrefPage: Current Plug-in: \n\t"
              + myPlugin.getClass());
}
```

You can use your plug-in id to ask the platform for a handle to the active plug-in. The `Plugin` class will accept any plug-in type (defined or default). Before you use a plug-in you need to know what type it is so you know what methods you can use.

2. In the runtime instance of the Workbench, open the preference dialog, select your preference page, and then click the Restore Defaults push button. You should see this in the console:

```
MBPrefPage: Current Plug-in:
  class com.ibm.lab.DialogsPlugin
```

This is because when your PDE project was created a plug-in class was generated for you. If a plug-in class was not included in your plug-in definition you would have seen the following:

MBPrefPage: Current Plug-in:
`class org.eclipse.core.internal.plugin.DefaultPlugin`

Step 2: Update code to use the Dialog Plug-in class

The platform plug-in class used in step 1 will not support preference processing. You need to have a handle to the DialogsPlugin class to support access to the stored preference processing methods provided when the plug-in class was generated using the PDE.

1. Modify the `performDefaults()` method in your preference page class so that you use your `DialogsPlugin` class.

   Replace this:

   ```
   Plugin myPlugin = Platform.getPlugin("com.ibm.lab.dialogs");
   ```

   With this:

   ```
   DialogsPlugin myPlugin = DialogsPlugin.getDefault();
   ```

2. Restart the test workbench to verify that your plug-in class is being used. Open the preference dialog, select your preference page, and then click the **Restore Defaults** push button. You should see now see this in the console:

   ```
   MBPrefPage: Current Plug-in:
         class com.ibm.lab.dialogs.DialogsPlugin
   ```


Part 3: Using a Preference Store

In this exercise you will add logic to store the value shown on the preference page created previously. This includes creating a PreferenceStore and then using it to manage default values and saved values for a key. These values will be shown on the preference page user interface.

Note: At this point you could skip ahead and try *Part 4: Implementing a field editor preference page* exercise that will let you use the preference store provided by the plug-in class you just defined and try the preference page implementation that uses `FieldEditors` to automate the implementation of saved values. You can always return here later.

Step 1: Create a Preference Store

Implement the hook method `doGetPreferenceStore()` to support creation of a preference store for this preference page dialog. The `Dialog` framework processing calls this method when you ask for the `PreferenceStore` using the `getPreferenceStore()` method.

Your `doGetPreferenceStore()` implementation will ask the plug-in for a `PreferenceStore`. This `PreferenceStore` will then become active for the preferences dialog.

1. Add the following import statements to `MyBasicPreferencesPage`:

```
import org.eclipse.jface.preference.*;
import org.eclipse.ui.plugin.*;
```

Add the highlighted logic to the init() method (shown below) in your `MyBasicPreferencePage` class.

```
public void init(IWorkbench workbench) {
    DialogsPlugin myPlugin = DialogsPlugin.getDefault();
    setPreferenceStore(myPlugin.getPreferenceStore());
  }
```

2. Add the lines of code highlighted below to the `setVisible()` method to request a `PreferenceStore`:

```
public void setVisible(boolean visible) {

    super.setVisible(visible);
  System.out.println(
      "PropCycle: " + "> in setVisible(" + visible + ")");

  // get prefStore reference
  IPreferenceStore myPS = this.getPreferenceStore();
    System.out.println(
      "PropCycle: "
        + "My PrefStore in setVisible: "
        + myPS.toString());
}
```


The class instance data written as part of the `System.out.println()` for the `myPS` variable will help you determine if you are using the same instance of a preference store when restarting the preference page.

3. Test the property page logic and check the console to see that you have created a preference store.

   Note: You have not actually saved the preference value yet. This is done in the next step.

   You should be able to see, in the console, that the same preference store handle is returned when you change the focus between your preference page and another preference page.

   You can step through the `doGetPreferenceStore()` method to see how preference store is created the first time and then retrieved as the current store from the dialog (this logic is driven by the `getPreferenceStore()` method in the `PreferencePage` super class).

Each time you start the dialog, the same preference store is used, this is implemented through the use of the plug-in class (a subclass of AbstractUIPlugin) to access to the preference store. The plug-in class creates the object when required, and keeps a handle to the object until the platform is shut down.

Step 2: Add preference store value management logic

The preference store is used to store and retrieve values by key. You also have the ability to define default values by key for use when no value has been defined (returned when the preference store keyed value is null).

This preference store value management logic will be added to the preference page life cycle methods `setVisible()` and `performOk()` and the plug-in method `initializeDefaultPreferences()`.

1. Begin by using the preference store to define a default value for the key that will be stored. Add this method to the `DialogsPlugin` class:

```
protected void initializeDefaultPreferences(
org.eclipse.jface.preference.IPreferenceStore store) {
  store.setDefault("text_field_key", "myTextDefaultValue");
  System.out.println("set preference defaults");
}
```

2. The `setVisible()` method is run when the preference page is about to be shown. Add the highlighted code below to the `MyBasicPreferencePage setVisible()` method.

```
public void setVisible(boolean visible) {

  super.setVisible(visible);
  System.out.println(
    "PropCycle: " + "> in setVisible(" + visible + ")");

  // get prefStore reference
  IPreferenceStore myPS = this.getPreferenceStore();
  System.out.println(
    "PropCycle: "
```



```
      + "> in setVisible: "
      + myPS.toString());

  if (visible) {
    textInfo.setText(
   getPreferenceStore().getString("text_field_key"));
  }
}
```

This will set the text field in the user interface to the value saved in the preference store (or the default if no value was defined).

3. To save the user interface data as a keyed value in the preferences store add the highlighted statements to the `performOk()` method:

```
public boolean performOk() {
  System.out.println("PropCycle: " + "> in performOk");

  getPreferenceStore().setValue(
    "text_field_key", textInfo.getText());

  return super.performOk();
}
```

4. To restore the default value add the highlighted statements to the `performDefaults()` method:

```
protected void performDefaults() {
  super.performDefaults();

  System.out.println(
"PropCycle: " + "> in performDefaults");

  // get active plug-in (default or defined)
  Plugin myPlugin =
Platform.getPlugin("com.ibm.lab.dialogs");
  System.out.println("Current Plug-in: \n\t" +
    myPlugin.getClass());

  textInfo.setText(
  getPreferenceStore().getDefaultString("text_field_key"));

}
```

5. Test the preference page logic and check the console to see that you have interacted with the preference store.

   The user interface text entry value will contain the default value at first (returned when the keyed value is null).

   Enter a value and change pages. If you modify the text entry content, but do not use the Apply push button, the originally saved (or default) value returns after a page focus change. This is

because the `setVisible()` method loads the value from the preference store each time the page is shown.

Enter a value and then use the Apply push button. You should see the new value return after you change the focus between your preference page and the Info property page. The Defaults push button will reset the visible value to the assigned default. You still have to use Apply to have these default values saved.

Exit the preferences dialog when testing is complete.

Part 4: Implementing a field editor preference page

A field editor preference page automates the process of linking the values obtained from visual controls with a keyed value in a preference store.

Step 1: Add preference page extensions

The first task is adding another preference page extension.

1. Add the following preference page plug-in extensions to the plugin.xml file.

   Note: You can copy these statements from scrapbook page EDU_plugin.xml:

```
<!-- VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV -->
<!-- Preferences Page2 Extension Field Editor               -->
<!-- ===================================================== -->
   <extension
  point="org.eclipse.ui.preferencePages">
  <page
    id="com.ibm.lab.dialogs.fieldPrefpage"
    name="EDU: Basic Preferences SubPage"
    class="com.ibm.lab.dialogs.MyFieldEditorPrefPage"
    category="com.ibm.lab.dialogs.prefpage1">
  </page>
   </extension>
<!-- vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv -->
```

   This extension adds a sub page to the preferences page you defined earlier.

2. Save the plugin.xml file to store the new preference page definition.

Step 2: Implement a FieldEditorPreferencePage

You will now implement a field editor preference page to map to the plug-in extension.

The MyFieldEditorPrefPage class will inherit the field editor preference page processing implemented as part of the dialog framework.

1. Create a MyFieldEditorPrefPage class in the com.ibm.lab.dialogs package. This new class must extend org.eclipse.jface.preference.FieldEditorPreferencePage and implement the IWorkbenchPreferencePage interface.

   Let the wizard generate the “Inherited abstract methods” but NOT the “Constructors from the superclass” (you will supply your own). This will add methods, such as the

`createContents()` method, which will be used to customize the user interface for the preference page.

Note: A compilation error will exist until you create the constructor in the next step.

2. Add the following import statement and constructor to the MyFieldEditorPrefPage class:

```
import org.eclipse.jface.preference.*;

public MyFieldEditorPrefPage() {
  super(GRID);
  setDescription("My Field Editor Preference Page \n");
}
```

You can test the page now but it will not have anything but a title.

3. Assign the preference store to the class by adding the highlighted logic to the constructor:

```
public MyFieldEditorPrefPage() {
  super(GRID);
  setDescription("My Field Editor Preference Page \n");
  IPreferenceStore store =
    DialogsPlugin.getDefault().getPreferenceStore();
  setPreferenceStore(store);
}
```

Note: This requires that you have implemented the plug-in class as described in Part 2. U*sing a plug-in to manage and store values*.

Step 3: Add Field Editors to the preference page

The `createFieldEditors()` method in the `MyFieldEditorPrefPage` class is called to populate the user interface with the controls you want on your preference page. You must customize this method to add the fields. Each field has a type and a key to be used in the preference store assigned to the preference page.

1. Review the available field editors by opening up the common super class (org.eclipse.jface.preference.FieldEditor). As you can see, there are many types of field editors:

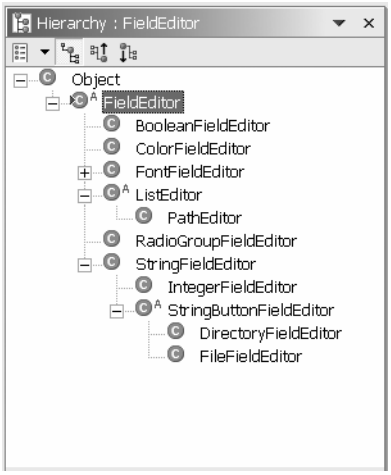


Figure 4-2
Field Editor Hierarchy [dialog_07.tif]

2. Customize the `createFieldEditors()` method to add field editors to your preference page.

```
protected void createFieldEditors() {

  // Note: The first String value is the key used in
  // the preference store and the second String value
  // is the label displayed in front of the editor.

  ColorFieldEditor colorField =
      new ColorFieldEditor(
          "COLOR_KEY", "COLOR_KEY_Label",
          getFieldEditorParent());

  BooleanFieldEditor choiceField =
      new BooleanFieldEditor(
          "BOOLEAN_KEY", "BOOLEAN_KEY_Label",
          org.eclipse.swt.SWT.NONE,
          getFieldEditorParent());

  FileFieldEditor fileField =
      new FileFieldEditor(
          "FILE_KEY", "FILE_KEY_Label",
          true,
          getFieldEditorParent());
  fileField.setFileExtensions(
      new String[] { "*.jar", "*.txt", "*.zip" });

  addField(colorField);
  addField(choiceField);
  addField(fileField);
}
```


Step 4: Testing the field editor preference page

1. Start the runtime workbench, open a preference dialog, and select your preference page. It should look like this:

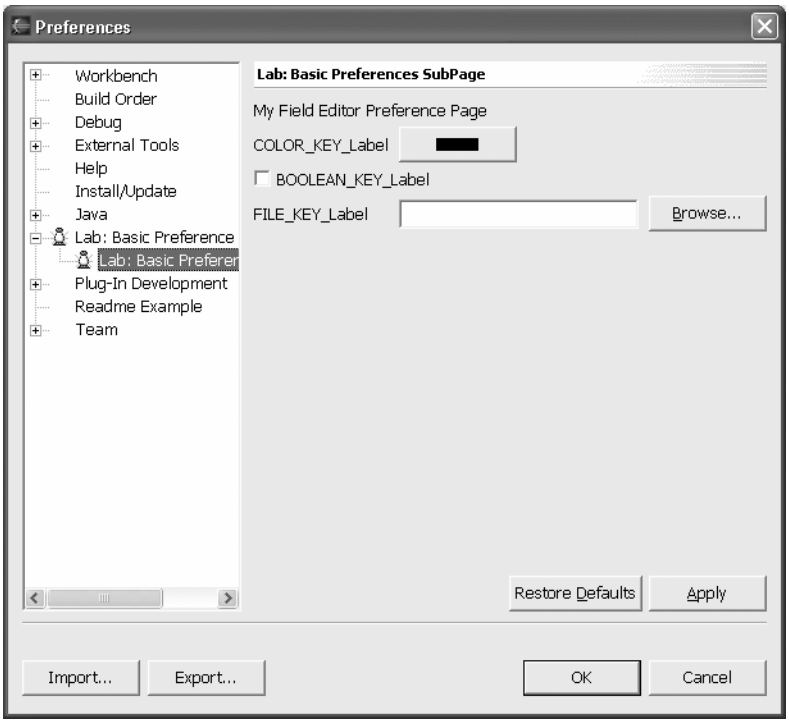


Figure 4-3
Customized Field Editor Preference Page [dialog_08.tif]

2. Manipulate the user interface to trigger the life-cycle methods. You can follow this process if you want:

   - Select the Lab: Basic Preferences Sub page
   - Manipulate the controls shown to set values
   - Select the **OK** push button
   - Shutdown the Workbench
   - Find the pref_store.ini file in this directory:

```
…\eclipse\
      runtime-workspace\
         .metadata\.plugins\com.ibm.lab.dialogs
```

   You should be able to see your field editor values in the file used to store preferences for your plug-in.

Exercise Activity Review

The dialog framework for preference page implementation was the focus area for this exercise.

You have worked with the basic functions of the dialog framework for preference pages by creating extension definitions for preference pages and defining Java implementations of preference pages and implementation of value storage using a preference store.

During this exercise you have learned how to:

- Add preference page extensions to a `plugin.xml` manifest file
- Create a preference page that could restore values to the user interface from a preference store
- Implement complete management of a preference store by adding support methods to the `DialogsPlugin` class.

Exercise 5: Implementing Property Pages

Introduction

This exercise guides you through the process of using the dialog support for property pages in the Workbench to create value management pages for resource property values.

Exercise Concepts

During this exercise, you will implement one or more of the following:

- Plug-in definitions that support property page extensions to the workbench.
- Persistence of data values using the resource API for properties.

Skill Development Goals

At the end of this exercise, you should be able to:

- Add property page extensions to a plug-in
- Create a property page that can save values

Exercise Setup

These setup tasks must be complete before you begin the exercise:

- Setup the Workbench so that all external plugins are visible to support plugin development and testing.
- Import the com.ibm.lab.dialogs project to your workspace.
- Choose the perspective you like to work with during tool development. Both the Java and Plug-in Development perspectives are useful for tool building.

Note: You may have already performed these setup activities if you worked through the wizard lab exercise.

The dialogs lab template provides you with a starter plug-in (EDU Dialogs) and a base set of code, but the implementation of some code components is incomplete. You get to add new methods and references to other methods not yet invoked by the base code.

The com.ibm.lab.dialogs project contains the following files:

- a`Plugin.xml` – a starter file to help you create the desired `plug-in.`
- `getstart_b.gif` – a graphic file that will be referenced by the plug-in manifest file.
- A set of Java scrapbook pages, organized by class, which contain code fragments referenced in this lab. You can use these to minimize the need to retype code defined in this document.
- Partially coded classes that will be completed during the lab.
- `EDU_plugin.xml` – a scrapbook of plug-in definitions that you can reuse during the lab.

`Other files, such as` .classpath and .cvsignore, which are not used in this exercise.

Exercise Instructions

Part 1: Property page definition

A property page dialog is started using the Properties option on the context menu for the file resource.

Step 1: Add property page extensions

You will define extensions for a properties page that can be used to implement saved value support for properties on resources found in the navigator.

1. Add the following properties plug-in extensions to the plugin.xml file.
   Note: You can copy these statements from scrapbook page EDU_plugin.xml:


```
<!-- VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV -->
<!-- Properties Page Extension                                    -->
<!-- ============================================================ -->
   <extension point="org.eclipse.ui.propertyPages">
  <page
    id="com.ibm.lab.dialogs.proppage1"
    name="EDU File Properties"
    objectClass="org.eclipse.core.resources.IFile"
    class="com.ibm.lab.dialogs.MyPropertyPage"
    nameFilter="*.edu">
  </page>
   </extension>
<!-- vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv -->
```

This defines a properties page associated to file resources with a name matching the *.edu pattern.

2. Save the plugin.xml file to store the property page definition.

Step 2: Define a property page

You will now implement one of the pages defined in the plug-in extension.

The `MyPropertyPage` class will inherit the property page processing implemented as part of the dialog framework and be available as a property page for files with the *.edu naming pattern.

Note: You can use the PDE Extension page editor to generate the `MyPropertyPage` class if you trigger the wizard using the drop down in the properties page for the class attribute. This is the process you followed in the FirstPlugin lab to generate the action class. If you use this approach you can skip to step 2.

1. Create a `MyPropertyPage` class in the `com.ibm.lab.dialogs` package. This new class must extend the class `org.eclipse.ui.dialogs.PropertyPage` and implement the interface `IWorkbenchPropertyPage`.

   Let the class wizard generate the “Constructors from the superclass” and “Inherited abstract methods”.

   By default, the PropertyPage super class provides an implementation for using a preference store to manage stored values. The controls you add to the property page can be initialized with default values and then used to store customized values in a preference store.

   Use of this dialog framework processing requires that you add logic to your property page implementation class.

2. Add a field to the MyPropertyPage class so you can interact with it in methods that will be defined later. Any errors will be corrected when you add import statements in the next step.

   ```
   private org.eclipse.swt.widgets.Text textInfo;
   ```

3. Customize the createContents() method so that you have added at least one widget to the property page. If you add these import statements to your class:


```
import org.eclipse.swt.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;
```

You can replace the generated method with this one to add a text widget like this:

```
protected org.eclipse.swt.widgets.Control createContents(
  org.eclipse.swt.widgets.Composite parent) {
  textInfo = new Text(parent, SWT.SINGLE | SWT.BORDER);
  textInfo.setText("Text String Value");
  GridData data = new GridData(GridData.FILL_HORIZONTAL);
  textInfo.setLayoutData(data);
  return textInfo;
}
```

Note: Remember, just as with other dialog type pages, the `createContents()` method must return a widget that is the parent of any other widgets that you add to the page.

4. Consider adding `System.out.println();` statements to selected methods in your Property Page class to trace data values or method invocation. For example, statements such as:

```
System.out.println("PropPage: ------> Constructor");
System.out.println("PropPage: ------> createContents");
```

could be added to the constructor and the `createContents()` method.

5. Start your runtime workbench to test your property page definition and implementation.

6. Use the Workbench new file wizard to create a file named myfile.edu in a project or folder. If you have not yet created a project, use the menu option **File** > **New** > **Project** to create a project and then create a file named something like myfile.edu.

   Select the new file in the navigator and use the Properties context menu option to see the category you created. When you select the category you should see the following:

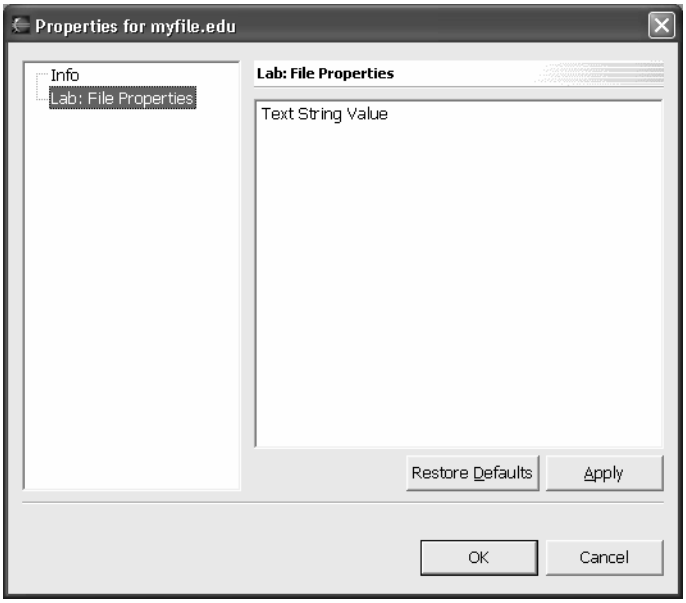


Figure 5-1
Basic Property Page [dialog_09.tif]

This is a custom property page, but there are only hard coded values shown. Note that the **Defaults, Apply, OK,** and **Cancel** buttons are added automatically, but you have not yet implemented any support for the value management life cycle in your property page class.

Part 2: Storing property values by resource

When you define a property page you have two options for storing values:

- PreferenceStore – good for when the value is not specific to the resource but to the resource type
- PersistentProperty – good for when the value is specific to the selected resource instance

To use a PreferenceStore you would follow the process outlined for a preference page (see *Part 3: Using a Preference Store* in the Implementing Preference Pages Exercise).

The processing required to save a value as a persistent property for a resource is similar to that used for preference pages (you use the same life-cycle methods), but the saved value is associated to the selected resource using the resource API defined as part of the workbench platform.

Step 1: Add property value management logic structure

In this step, you will add the hook methods to your property page class.

1. Copy these methods to the MyPropertyPage class from the scrapbook page:

```
setVisible()
performDefaults()
performOk()
performApply()
performCancel()
```


2. If you wish, you can test the integration of these value management methods. Use the same process described in the Implementing Preference Pages Exercise, *Step 3: Adding value management life-cycle support*.

Step 2: Add resource API logic

The persistent property resource API is used to store and retrieve values by key.

This preference store value management logic will be added to the property page life cycle methods setVisible() and performOk().

1. Add the field and import statements shown below to prepare for using the resource API to store a persistent property:

```
  import org.eclipse.core.resources.*;
  import org.eclipse.core.runtime.*;

  private static QualifiedName PROP_NAME_KEY =
    new QualifiedName("com.ibm.lab.dialogs", "My_Prop_Name");
```

2. The setVisible() method is run when the properties page is about to be shown. By adding the highlighted code below, the setVisible() method will set the text field in the user interface to the value saved as a persistent property (or a value that indicates that no value was found).

```
public void setVisible(boolean visible) {

  super.setVisible(visible);

  System.out.println("PropCycle: " +
    "> in setVisible(" + visible + ")");

  IResource resource = (IResource) getElement();

  try {
    String propValue =
  resource.getPersistentProperty(PROP_NAME_KEY);
    if (propValue != null)
      textInfo.setText(propValue);
    else
      textInfo.setText("No prop value found");
  } catch (CoreException e) {
      System.out.println(
        "PropPage: -----> Trouble in get persistentProp");
      System.out.println(
        "PropPage: -----> " + e);
  }
}
```

3. Save the user interface data as a keyed value associated with the selected resource by adding the highlighted statements to the `performOk()` method:


```
public boolean performOk() {
  System.out.println("PropCycle: " + "> in performOk");
   IResource resource = (IResource) getElement();

  try {

    resource.setPersistentProperty(PROP_NAME_KEY, textInfo.getText());

  } catch (CoreException e) {
    System.out.println(
      "PropPage: -----> Trouble in get persistentProp");
    System.out.println("PropPage: -----> " + e);
  }

  return super.performOk();
}
```

4. Test the property page logic and check the console to review any messages that were posted. The user interface text entry value will contain a hard-coded value at first (returned when the keyed value is null).

   Enter a value and change pages. If you modify the text entry content, but do not use the Apply push button, the originally saved (or default) value returns after a page focus change. This is because the setVisible() method loads the value from the persistent property each time the page is shown.

   Enter a value and then use the Apply push button. You should see the new value return after you change the focus between your property page and the Info property page. The Defaults push button could be used to reset the visible value to an assigned hard-coded default (not implemented yet).

   Exit the properties dialog when testing is complete.

   Can you find where the property value saved has been stored by the Workbench platform? What does this choice of physical location mean?

Exercise Activity Review

The dialog framework for property page implementation was the focus area for this exercise.

The dialog framework for property pages was the focus area for this exercise.

You have worked with the basic functions of the dialog framework for property pages by creating extension definitions for property pages and defining Java implementations of a property page.

Tasks completed include:

- Added property page extensions to a plugin.xml manifest file
- Created a property page that could save and restore values using the property page user interface and the resource API for property values.

Exercise 6 Defining a JFace Component

This exercise takes you through the process of defining and using a JFace component in a user interface. In this exercise, the user interface component will be a simple dialog opened using a context menu action on any `IFolder` resource named readme in the **Navigator** or **Package Explorer** views.

The JFace component includes a viewer, content provider, and label provider which work together to render information about a defined input object. The input object for the viewer will be identified by the user interface part that uses the JFace component, a dialog in this exercise.

As you will see later, the JFace component defined will be reusable in other user interface parts.

Introduction

JFace is a fundamental building block for user interfaces when programming in Eclipse. You can use SWT, JFace viewers, or a mix of both to define the user interface for a dialog of any kind, a view, or an editor.

JFace viewers are adapters on SWT widgets that provide a specific framework for how widget content (data to be shown) is obtained from an input identified to the viewer and known to the content provider. The widget behind the viewer will be populated with objects that the content provider finds in the input domain. The label provider will provide text and image values that are shown for each element instance returned by the content provider.

Exercise Concepts

The exercise begins with a working action and dialog, but the dialog shows nothing but the name of the folder defined as input to the dialog by the action.

You will test this existing code, then build a JFace component that includes a viewer, content provider, and label provider. This implementation will expose a portion of the viewer API so that the JFace component can accept an input, add selection change listeners, use a predefined table in the viewer, and allow alternate content provider and label provider implementations to be identified.

Skill Development Goals

This exercise looks at the definition and use of JFace-based user interface so that you can understand:

- The role of a viewer.
- The role of a content provider.
- The role of a label provider.
- How a viewer reports selection of an object being displayed.

The steps defined in each section are continuous. That is, they must be done in sequence; you cannot jump straight to Section 2 or 3.

Exercise Setup

Before you can begin this exercise, you must have access to the `com.ibm.lab.JFace` template plug-in project. This project should already have been imported into your workspace. If not, import the plug-in from the exercise templates location on your workstation.

Exercise Instructions

Note: Each part must be performed in sequence. The definitions in Part 1 are used in Part 2; the result of Part 2 sets up the system for Part 3.

Part 1: Test Current Dialog opened by Popup Menu Contribution

In Part 1 you will simply validate the function of the code given to you at the start of this exercise:

- An Action Contribution targeted at folders named readme
- A Dialog that is created with a reference to the selected folder and opened by the Action Contribution

Step 1. Review the Action Contribution Implementation

1. The plugin.xml defines a popup menu contribution.

   How is the contribution defined such that it only exists for folders named readme?

2. Review the Java implementation of the action contribution.

   How is it saving a reference to the selected folder?

Step 2. Review the Dialog Implementation

1. Review the structure of the `MembersDialog` class.

   Where is the user interface defined?

   How is the folder known to the action passed to the dialog?

2. Identify where the folder reference is used in the `MembersDialog` class.

Step 3. Test the Existing Function

1. Start the runtime workbench.
2. If required, create a folder named `readme` and add some content to this folder. You can use the example file creation wizard to create a few `.readme` files if you wish. The **Readme File** creation wizard can be found in the **Examples > Example Creation Wizards** category when you have the Eclipse examples installed. If not, create several `.readme` files using the standard **New File** wizard.
3. Select the readme folder and then select the context menu:
   Edu: JFace > Edu: Open JFace Dialog.

   The result should be a dialog that looks like this:

Figure 6-1
Template JFace Dialog

Close the runtime workbench after testing is complete.

Part 2: Implement JFace Component

The next task is to build a JFace component that uses a viewer, content provider, and label provider to create the user interface content for a given input. You will build this as a reusable component to both learn how to configure a JFace viewer, but also help reinforce the idea that JFace viewers can be used anywhere you need a user interface. Anywhere Eclipse gives you a composite widget (`createControl(), createPartControl(), createDialogArea()`), you can create a viewer.

This JFace component will be used again in the **View Programming** exercise. Some elements of the JFace component API you will define will not be used until then.

Step 1. Define Base JFace Component

A standalone class will be used to create the JFace viewer as well as implement the associated content provider and label provider. This class will wrap a JFace viewer and expose some of the viewer's framework methods so we can use this class as a full function viewer with the internally implemented content provider and label provider.

The easiest way to do this is let the PDE do most of the work and change the generated code to suit our needs.

1. Create a new plug-in project with a view that contains table content, without any of the optional view features, as generated by the PDE.

   Do the following to create this plug-in project with the correct view code:

   - Open the New Wizard selection page (**Ctrl+N**), choose the **Plug-in Development** category and **Plug-in Project** option, and then select the **Next >** button.
   - Enter any value you want for the Project name and then select the **Next >** button twice to get to the Plug-in Code Generators page.
   - Select the **Plug-in with a view** entry in the list and then select the **Next >** button three times to get to the View Features page.
   - Deselect all the toggle options on the View Features page and then select the **Finish** button.

   This generates a `SampleView` class that will be moved to the JFace project and simplified to become the JFace component.

2. Select the `SampleView` class and drag it to the package in the `com.ibm.lab.jface` project. The **Move** dialog will be opened, just choose **OK**.

   Once done delete the plug-in project you created to get the view code generated.

3. Open the `SampleView` class in the Java editor and delete the following methods:

   - `createPartControl(Composite parent)`
   - `showMessage(String message)`
   - `setFocus()`

   If you want you can move the `SampleView()` constructor to the top of the file.

4. Modify the class definition to remove the superclass reference; the class should not extend anything. Use the **Source > Organize Imports** option (**Ctrl+Shift+O**) to clean things up and then save the `SampleView` class source.

5. Use the Refactor > Rename context menu in the Package Explorer view to rename the `SampleView` class to `JFaceComponent`.

   You now have a class that can be reworked to be a reusable JFace component.

Step 2: Add Viewer Access and Delegate Methods to the JFace Component

The `viewer` field in the `JFaceComponent` class needs to be accessible by the user interface part that wants to reuse this class. A get method and delegate methods for the `viewer` will be added to the `JFaceComponent` API.

1. Generate a get method for the `viewer` field.

Select the `viewer` field in the Outline view and choose the context menu **Source > Generate Getter and Setter…** context menu. Generate a get method only.

2. Generate several delegate methods for the `viewer` field.

   Select the `viewer` field in the Outline view and choose the context menu **Source > Generate Delegate Methods…** context menu. Generate delegates for the following methods:

   - `addSelectionChangedListener(ISelectionChangedListener)`
   - `refresh()`
   - `refresh(Object)`
   - `setContentProvider(IContentProvider)`
   - `setInput(Object)`
   - `setLabelProvider(ILabelProvider)`
   - `update(Object, String[])`
   - `update(Object[], String[])`

   These methods will allow the user interface parts to communicate with the associated viewer control when they use the `JFaceComponent` class.

Step 3: Add Viewer Creation and Configuration Support to the JFace Component

When the viewer inside the JFaceComponent is created it must be given a parent composite or table reference. Constructors will be used to pass in on of these references for this purpose.

Methods that support the configuration of the viewer with either a default or custom content provider and label provider will also be added.

1. Add the following constructors to the `JFaceComponent` class:

```
/**
 * @param composite
 */
// Parent composite used to create viewer
public JFaceComponent(Composite composite) {

  this.viewer = new TableViewer(composite);
}

/**
 * @param composite, table
 */
// Table used to create viewer - table has a parent composite
public JFaceComponent(Table table) {
  this.viewer = new TableViewer(table);
}
```

2. Use the **Source > Organize Imports** option (**Ctrl+Shift+O**) to add the required import statements. Select the SWT composite and JFace table if prompted. You are prompted if the Java editor finds duplicate classes so you have to choose between those offered.

3. Add the following `configureViewer()` methods to the `JFaceComponent` class:

```
/**
 * Viewer gets a default content provider and label provider
 */
public void configureViewer() {
  setContentProvider(new ViewContentProvider());
  setLabelProvider(new ViewLabelProvider());
}

/**
 * Viewer gets a custom content provider and label provider
 */
public void configureViewer(IContentProvider cp, IBaseLabelProvider lp) {
  setContentProvider(cp);
  setLabelProvider(lp);
}
```

The `JFaceComponent` class is ready to be used by a user interface part. The viewer currently displays only static data (see the `getElements()` method in the content provider), but this will still validate that it is functional when used in the user interface.

Part 3: Use JFace Component to Display Content in a Dialog

The current dialog just shows the name of the folder passed as input by the action. It is time use the JFace component to customize the dialog user interface.

The JFace component will be added first, so it can be tested, then the default content provider will be customized to obtain content from the input identified to the viewer.

Step 1. Add JFace Component to Dialog Composite

Begin by adding the viewer to the existing user interface in the dialog.

1. Add the following code to the `addJFaceViewer()` method:

```
JFaceComponent jcomp = new JFaceComponent(viewerArea);
jcomp.configureViewer();
jcomp.setInput(focusFolder);
```

   This method is called by the `createDialogArea()` method in the `MembersDialog` class. The JFace component constructor will create the viewer using the passed composite as the parent, the configure process defines the default content provider and label provider, and the input known to the dialog is passed to the viewer.

   That's all it takes! You will customize this method later to add selection processing.

2. Launch the runtime workbench to test the new dialog user interface. Select the readme folder and then select the context menu **Edu: JFace > Edu: Open JFace Dialog**. The dialog shown should look like this:

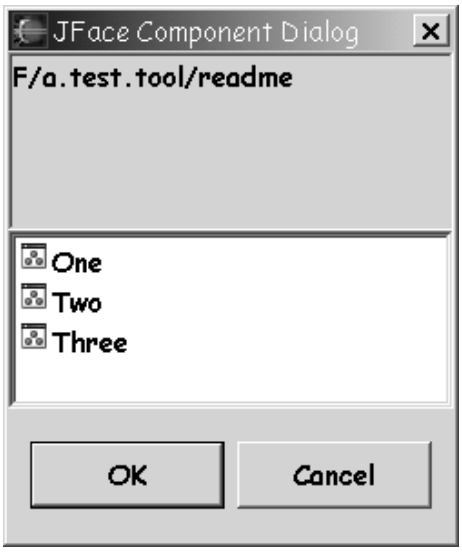


Figure 6-2
JFace Component Viewer in the Dialog User Interface

The dialog displays input known to the dialog and the static data defined in the `getElements()` method of the content provider.

Close the runtime workbench after testing is complete.

Step 2. Customize the Content Provider for the Type of Input Object
The current dialog just shows the name of the folder passed as input by the action. This value has been passed to the viewer using the `setInput()`method, but the content provider has not used the input to find content.

It is time to customize the default content provider in the JFace component to find content based on the defined input. When the input is defined to the viewer, it in turn tells the current content provider using the `inputChanged()` method. The viewer then asks for content using the `getElements()` method. These methods need some customization.

1. Modify the `inputChanged()` method in the content provider (the inner class in the `JFaceComponent` class).

   Add the `input` field and modify the method to match the logic shown below:

```
private IResource input;
… … … …
public void inputChanged(Viewer v, Object oldInput, Object newInput) {
   System.out.println(
      "Viewer-> ContentProvider.inputChanged() - Input: " + newInput);
   this.input = (IResource) newInput;
}
```

   Use the **Source > Organize Imports** option (Ctrl+Shift+O) to add the required import statement(s).

2. Modify the `getElements()` method in the content provider. Modify the method to match the logic shown below:

```
public Object[] getElements(Object parent) {
  System.out.println(
    "Viewer-> ContentProvider.getElements() - for parent: " + parent);
  if (input instanceof IContainer) {
    IContainer inputContainer = (IContainer) input;
    IResource[] members = null;
    try {
      members = inputContainer.members();
    } catch (CoreException e) {
      // Auto-generated catch block
      e.printStackTrace();
    }
    if (members.length != 0) {
      return (Object[]) members;
    };
  }
  return new String[] { "no members to display" };
}
```

   The input known to the dialog is a folder in the resource tree (`IContainer`), but the logic shown above is prepared for any kind of resource (project/folder/file).

   Use the **Source > Organize Imports** option (Ctrl+Shift+O) to add the required import statement(s).

3. Launch the runtime workbench to test the modified dialog user interface. Select the readme folder and then select the context menu **Edu: JFace > Edu: Open JFace Dialog**. The dialog shown should look like this:

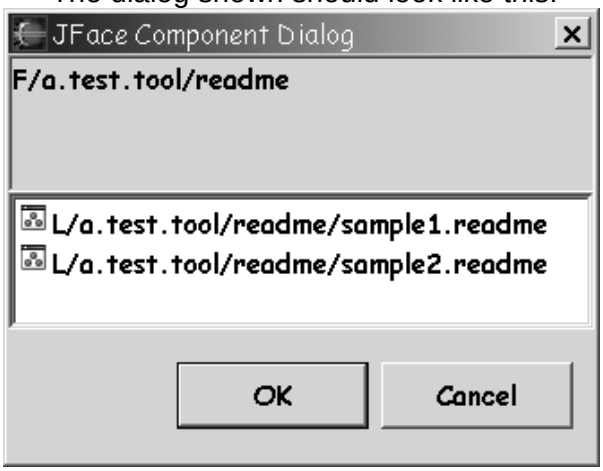


Figure 6-3
JFace Component Viewer with Customized Element Content

   The viewer in the dialog displays the data returned by the `getElements()` method of the content provider. The values shown are the result of `toString()` on the objects. This is the default label provider processing.

   Close the runtime workbench after testing is complete.

Step 3. Customize the Label Provider for the Object Elements
The default label provider processing needs to be customized. As with the content provider, the label provider also needs to know about the type of object shown in the viewer.

1. Modify the `getColumnText()` method to the label provider (the inner class in the `JFaceComponent` class). Add the method logic shown below:

```
public String getColumnText(Object obj, int index) {
  if (obj instanceof IResource) {
    IResource res = (IResource) obj;
    return res.getName();
  }
  return getText(obj);
}
```

   If the current element object is not an `IResource`, the default value (`toString()`) is returned by the superclass implementation of `getText()`.

2. Modify the `getImage()` method in the label provider to customize the type of image returned. The method should match the logic shown below:

```
public Image getImage(Object obj) {
   if (obj instanceof IResource) {
     IResource res = (IResource) obj;
     switch (res.getType()) {
       case IResource.PROJECT :
         return PlatformUI.getWorkbench().getSharedImages().getImage(
           ISharedImages.IMG_OBJ_PROJECT);
       case IResource.FOLDER :
         return PlatformUI.getWorkbench().getSharedImages().getImage(
           ISharedImages.IMG_OBJ_FOLDER);
       case IResource.FILE :
         return PlatformUI.getWorkbench().getSharedImages().getImage(
           ISharedImages.IMG_OBJ_FILE);
     }
   }
   return null;
}
```


3. Launch the runtime workbench to test the modified dialog user interface. Select the readme folder and then select the context menu **Edu: JFace > Edu: Open JFace Dialog**. The dialog shown should look like this:

Figure 6-4
JFace Component Viewer with Customized Display Content

The data displayed in the viewer is the result of the modified `getColumnText()` and `getImage()` methods in the label provider.

Close the runtime workbench after testing is complete.

Step 4. Add a Listener to the Content Provider to Synchronize the Viewer with Model Changes

The content provider is responsible for listening to the model and telling the viewer when changes have occurred. The model could be the input object passed to the content provider in the `inputChanged()` method, or be the domain in which the input is a participating object.

If the input represents the model instance itself, a listener would be added to the new input, and removed from the old input, each time the `inputChanged()` method was called.

For this content provider the input is an object in the workspace model, so all we need is a single listener, but we also need to remove the listener when the content provider is disposed.

1. Modify the content provider `inputChanged()` method to define the content provider as a workspace resource change listener:

```
public void inputChanged(Viewer v, Object oldInput, Object newInput) {
   System.out.println(
      "Viewer-> ContentProvider.inputChanged() - Input: " + newInput);

   // If this is the first time we have been given an input
   if (oldInput == null) {
      IResource resource = (IResource) newInput;
      // Content Provider hook to model changes
      ResourcesPlugin.getWorkspace().addResourceChangeListener(this);
   }
   this.input = (IResource) newInput;
}
```

   You will also need to add `implements IResourceChangeListener` to the `ViewContentProvider` inner class definition. The compiler will warn you about a missing method (`resourceChanged()`), but you will add that shortly.

   Use the **Source > Organize Imports** option (Ctrl+Shift+O) to add the required import statement(s).

2. Modify the content provider dispose() method to remove the resource change listener:

```
public void dispose() {
  // Remove listener when content provider is disposed
  ResourcesPlugin.getWorkspace().removeResourceChangeListener(this);
}
```

3. Tell the viewer that the model has changed after a resource change event:

```
    public void resourceChanged(IResourceChangeEvent event) {
      System.out.println("resourceChanged");
      final IResourceDelta delta = event.getDelta();

      // Content Provider tells viewer about model change
      Control ctrl = viewer.getControl();
      if (ctrl != null && !ctrl.isDisposed()) {
        // Do a sync exec, not an async exec, since the resource delta
        // must be traversed in this method. It is destroyed
        // when this method returns.
        ctrl.getDisplay().syncExec(new Runnable() {
          public void run() {
            viewer.refresh();
          }
        });
      }
    }
```

   The resource change event is sent on a non-UI thread so we have to get on the UI thread before telling the viewer to refresh. The `refresh()` method tells the viewer to rebuild the content. The `refresh(element)` or the available `update()` methods

would provide a more granular notification of change and allow the viewer to be more efficient. You should consider these alternatives in your JFace viewer programming.

This logic cannot be tested yet as the dialog blocks user changes to the workspace. When the JFace component is used in a view the role of this logic will be more apparent.

Step 5. Add Viewer Selection Processing to the Dialog
One last customization to dialog that makes use of the viewer capabilities. A viewer can report element selection to an interested party.

1. Add this logic to the end of the `addJFaceViewer()` method in the `MembersDialog` class. This code should go after the viewer input has been defined:

```
jcomp.addSelectionChangedListener(new ISelectionChangedListener() {
   public void selectionChanged(SelectionChangedEvent event) {
      // Send selection to label
      System.out.println(">Dialog hears selection" + event.getSelection());
      IStructuredSelection ssel = (IStructuredSelection) event.getSelection();
      selected.setText("");
      if ((ssel != null) && (ssel.getFirstElement() instanceof IResource)) {
         IResource res = (IResource) ssel.getFirstElement();
         selected.setText(res.getName());
      }
   }
});
```

   Use the **Organize Imports** option to correct any missing import statements.

2. Launch the runtime workbench to test the modified dialog user interface. Select the readme folder and then select the context menu **Edu: JFace > Edu: Open JFace Dialog**. In the dialog select an entry in the viewer.

   The dialog shown should look like this:

Figure 6-5
JFace Component Viewer with Viewer Selection Processing

The data displayed in the viewer is the result of the modified `getColumnText()` and `getImage()` methods in the label provider.

Close the runtime workbench after testing is complete.

Exercise Activity Review

What you did in this exercise:

- Learned how to use a viewer in a user interface.
- Customized a content provider to get data from the input source and a label provider to work with the objects shown in the viewer.
- Added viewer selection processing to demonstrate how you can find and react to the selection known to the viewer.

Exercise 7
Defining a View Part

This exercise takes you through the process of defining and using a View part in the workbench. In this exercise, the user interface component will be based on the JFace component you created earlier. Additional function, as appropriate for a view part, will be added to the component.

Note: If you did not complete the JFace component exercise, you can use the `com.ibm.soln.jface` project instead. You may need to import this project from the plug-in development solutions folder.

Introduction

View parts are the fundamental user interface building block in the Eclipse workbench. You will probably define more views than editors as part of your tool.

Views typically include JFace viewers, as a viewer is prepared to be integrated into the workbench and can be configured so they communicate with other workbench parts. Views can also have their own task and menu bars that supplement the context menu that can be added to a viewer.

Some of the code you will write in this exercise applies only to a view part, but other code, such as context menu creation applies to a viewer as it may be used in any type of user interface (view part, dialog, editor).

Exercise Concepts

The exercise begins with a working action and dialog, but the dialog shows nothing but the name of the folder defined as input to the dialog by the action.

You will test this existing code, then build a JFace component that includes a viewer, content provider, and label provider. This implementation will expose a portion of the viewer API so that the JFace component can accept an input, add selection change listeners, use a predefined table in the viewer, and allow alternate content provider and label provider implementations to be identified.

Skill Development Goals

This exercise looks at the definition and use of JFace-based user interface so that you can understand:

- The role of a viewer.
- The role of a content provider.
- The role of a label provider.
- How a viewer reports selection of an object being displayed.

The steps defined in each section are continuous. That is, they must be done in sequence; you cannot jump straight to Section 2 or 3.

Exercise Setup

Before you can begin this exercise, you must have access to the `com.ibm.lab.resourceview` template plug-in project. This project should already have been imported into your workspace. If not, import the plug-in from the exercise templates location on your workstation.

Primary Exercise Instructions

Note: Each part must be performed in sequence. The definitions in Part 1 are used in Part 2; the result of Part 2 sets up the system for Part 3.

Part 1: Implement View Part

The first task is to define and create the view part. First, you add the extension definition, then you create a class by letting it be generated by the PDE.

Step 1. Define View Extension

Edit the plugin.xml and add the view extension, this includes a category for the view (how it will be found in the Show View dialog) and the view itself.

1. Edit the `plugin.xml` file in the `com.ibm.lab.resourceview` project you imported earlier. Select the Extensions tab. You will now specify the information needed for the View extension.
2. Define the view extension.

   Select the **Add…** button. Select **Generic Wizards > Schema-based Extensions**. Press **Next**. Scroll down the list and select the extension point for views, **org.eclipse.ui.views** and Press **Finish**.
3. Define a view category.

   Select the **org.eclipse.ui.views** entry in the extensions list and choose the **New > Category** context menu. Using the **Properties** view, modify the id for the category to be **com.ibm.lab.view.category** and the name for the category to be **Edu: Views**. The parent category will be blank.
4. Define the view

   Select the **org.eclipse.ui.views** entry in the extensions list and choose the **New > View** context menu.

   Using the Property View, specify the following:

   - category of `com.ibm.lab.view.category`
   - icon of `icons\sample.gif`
   - id of `com.ibm.lab.view.resourceview`
   - name of `Resource View`

   Ignore the class property for now, you will specify a value later.
5. Save the `plugin.xml` file. The XML for the new view should look like this in the Source page:

```
<extension
     point="org.eclipse.ui.views">
   <category
        name="EDU: Views"
        id="com.ibm.lab.view.category">
   </category>
   <view
        name="Resource View"
        category="com.ibm.lab.view.category"
        class="com.ibm.lab.resourceview.ViewPart1"
        id="com.ibm.lab.view.resourceview">
   </view>
</extension>
```

Step 2. Generate View Part

The PDE can generate classes for many extension types. You will now use the PDE to generate a class for your view part.

1. Return to the extensions page of the plugin.xml file and select the **Resource View (view)** entry. Generate the View class using the PDE.

In the Property View, generate the class by selecting the continuation entry (...) in the class field. In the Java Attribute Editor, specify that you want to generate a new Java class. The class name is `ResourceView` and you want to let the wizard open an editor on the class after it is created. Leave the source folder and package name at their default settings.

Select **Finish** to generate the class.

When complete the wizard will open an editor open on the `ResourceView` class. The wizard generated the required `createPartControl()` method and a setFocus() method.

2. Add an SWT control to the View's user interface in the `createPartControl()` method. Your choice of SWT control content. For fun try using the SWT Layouts example view to generate a bit of SWT code you can paste into the method. You will have to modify the SWT Layout view generated code to use the composite passed to the `createPartControl()` method instead of the `shell` referenced included in the generated code.

   Use the **Source > Organize Imports** option (**Ctrl+Shift+O**) to add the required import statement(s).

   Your `createPartControl()` method should now look something like this (created using the FillLayout page in the SWT Layout view):

```
public void createPartControl(Composite parent) {
  RowLayout rowLayout = new RowLayout();
  FillLayout fillLayout = new FillLayout ();
  parent.setLayout (fillLayout);
  List list0 = new List (parent, SWT.BORDER);
  list0.setItems (new String [] {"Item 1", "Item 2", "Item 3"});

  Tree tree1 = new Tree (parent, SWT.BORDER);
  TreeItem treeItem1 = new TreeItem (tree1, SWT.NONE);
  treeItem1.setText ("Item1");
  TreeItem treeItem2 = new TreeItem (tree1, SWT.NONE);
  treeItem2.setText ("Item2");

  Table table2 = new Table (parent, SWT.BORDER);
  table2.setLinesVisible (true);
  TableItem tableItem1 = new TableItem (table2, SWT.NONE);
  tableItem1.setText ("Item1");
  TableItem tableItem2 = new TableItem (table2, SWT.NONE);
  tableItem2.setText ("Item2");
}
```

   Note: Copy the code above from the `JPages\Part1_ViewPart.jpage` file.

3. Launch the runtime workbench to test the view user interface.

Open the view: **Window > Show View > Other…**and then select the Resource View in the **Edu Views** category. The view shown will look like this if you used the code shown above:

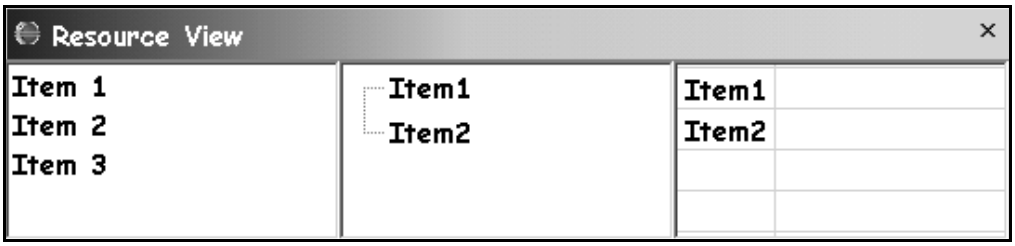


Figure 7-1
Resource View with SWT-based User Interface

You now have a class that can be reworked to be a functional View part.

The SWT controls will be replaced by a viewer created from the reusable JFace component defined in a previous exercise (or you can use the JFace component solution provided).

Close the runtime workbench after testing is complete.

Part 2: Add the JFace Component to the View Part

The current view just shows the SWT controls you added to test the view. It is time use the JFace component to customize the view user interface.

The JFace component will be added first with a default input value so it can be tested, then the view will be customized to obtain the input from a source outside the view.

Note: If you did not complete the JFace component exercise you will need to load the solution and use that code during this exercise. Use the same technique outlined in Exercise 4. Lab Imports but import only the project com.ibm.soln.jface from the \PluginSolutions directory.

Step 1. Add JFace Component to View User Interface

Begin by adding the viewer in when the view's user interface is created. This must begin with a modification to the plugin.xml to add the JFace component project to the list of required projects, only then can we get compile time and runtime visibility to the classes in that project.

1. Modify the `plugin.xml` dependencies by adding the `com.ibm.lab.jface` or `com.ibm.lab.soln.jface` project to the list of required projects. The `com.ibm.lab.resourceview plugin.xml` requires element should look like this when you are done:

```
<requires>
   <import plugin="org.eclipse.core.resources"/>
   <import plugin="org.eclipse.ui"/>
   <import plugin="com.ibm.lab.jface"/>
</requires>
```

2. Modify the `plugin.xml` runtime definition for the `com.ibm.lab.jface` project to share the runtime code with other plug-ins. Select the jar file name on the **runtime** page and then

select the **Export the entire library** option. If you are using the `com.ibm.lab.soln.jface` project this has already been done.

The `com.ibm.lab.jface plugin.xml` runtime element should look like this when you are done:

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

3. Add this field to the `ResourceView` class definition:

```
// JFaceComponent reference for listeners to be defined later
JFaceComponent jcomp;
```

4. Modify the `createPartControl()` method so that it contains only the following code:

```
jcomp = new JFaceComponent(parent);
jcomp.configureViewer();
jcomp.setInput(ViewPlugin.getWorkspace().getRoot());
```

The JFace component constructor will create the viewer using the passed composite as the parent, the configure process defines the default content provider and label provider, and the workspace root is defined as the input to the viewer.

Use the **Source > Organize Imports** option (**Ctrl+Shift+O**) to revise the import statements to match the code.

That's all it takes! You will customize this method later to add selection processing.

5. Launch the runtime workbench to test the new view user interface. The view shown should look like this:

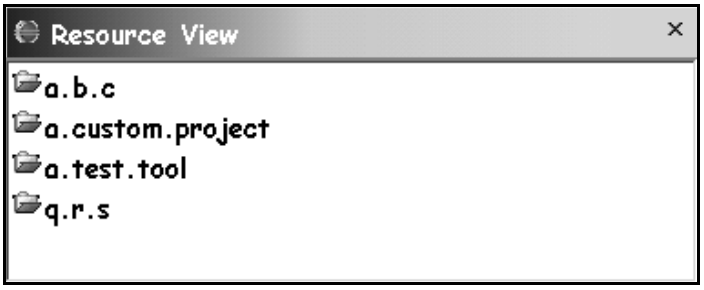


Figure 7-2
Resource View with JFace Component Viewer in the User Interface

The view displays content based on the identified input, the full workspace root.

Since the content provider in the JFace component was constructed with a resource change listener, the content in the Resource View will change if you add a new project to the workspace.

Close the runtime workbench after testing is complete.

Step 2. Customize the View to React to Viewer Selection Changes
You did this in the JFace component exercise, but this time you will do something a bit more interesting. The view part has access to the workbench site, and from there can contribute content to the status bar.

1. Modify the `createPartControl()` method in the view part by adding the logic shown below after the viewer input has been defined:

```
jcomp.addSelectionChangedListener(new ISelectionChangedListener() {
   public void selectionChanged(SelectionChangedEvent event) {
      System.out.println(">View hears selection" + event.getSelection());
      IStructuredSelection ssel = (IStructuredSelection) event.getSelection();

      //Get common Workbench status bar
      IStatusLineManager status =
        getViewSite().getActionBars().getStatusLineManager();

      // Push IResource location to status bar
      if ((ssel != null) && (ssel.getFirstElement() instanceof IResource)) {
        IResource res = (IResource) ssel.getFirstElement();
        status.setMessage(res.getLocation().toString());
      } else {
        status.setMessage(null);
      }
      status.update(false);
   }
});
```

Note: Copy the code above from the `JPage\Part2_JFace_Comp.jpage` file.

Use the **Source > Organize Imports** option (**Ctrl+Shift+O**) to revise the import statements to match the code.

2. If you wish to simplify the `createPartControl()` method logic you can refactor the code. Extract a method from the `addSelectionChangedListener()` logic just added. Name the new method `processViewerSelection()`. This method will be invoked as part of the `createPartControl()` method.
3. Launch the runtime workbench to test the modified view user interface. Open the Resource View if required (it may be open from before). The workbench shown should look like this when an element in the view is selected:

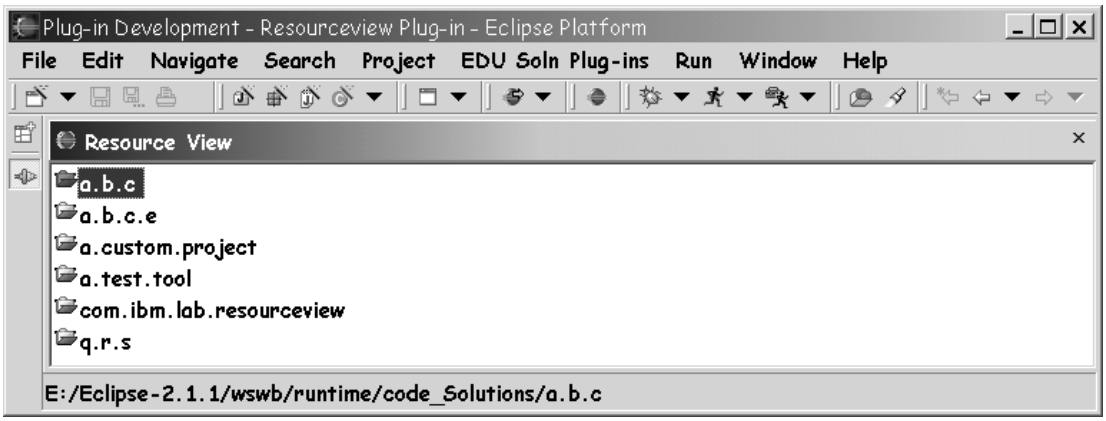


Figure 7-3
Status Bar Content from Viewer Selection in View

The workbench status bar displays information based on the selected element in the JFace viewer in the view. Close the runtime workbench after testing is complete.

Part 3: Add Workbench Selection Processing to the View Part: Listen and Share

The current view reacts to the element selected in the viewer contained in the view part. The JFace architecture allows workbench parts that want to participate, the ability to both listen to selections outside their own part and share the current selection of their viewer with other workbench parts.

This level of communication is what allows new views to react to the selection in existing views, and how common Outline and Properties views determine what they will show in their content area.

You will now modify your Resource View to both share its own selection with any workbench part that cares to listen and then listen and react to the selection in the Navigator view.

Step 1. Share Viewer Selection with Other Workbench Parts

Lets make one last customization to view that makes use of the viewer capabilities. A viewer can report element selection to an interested party.

1. Modify the `createPartControl()` method in the view part by adding the logic shown below at the end of the method:

```
// Share Viewer Selection with other workbench parts
getViewSite().setSelectionProvider(jcomp.getViewer());
```

This will allow other views, such as the properties view, to react to the current selection if that selection can contribute properties view content. If you review the content provider in the `JFaceComponent`, you will see that the `getElements()` method returns an `IResource` array when members are found in the input.

2. Launch the runtime workbench to test the modified Resource View.

   Open the Properties view and arrange the views so you can see both the Resource view and Properties view. Select an entry in the Resource view and inspect the contents of the Properties view. You should see something like this:

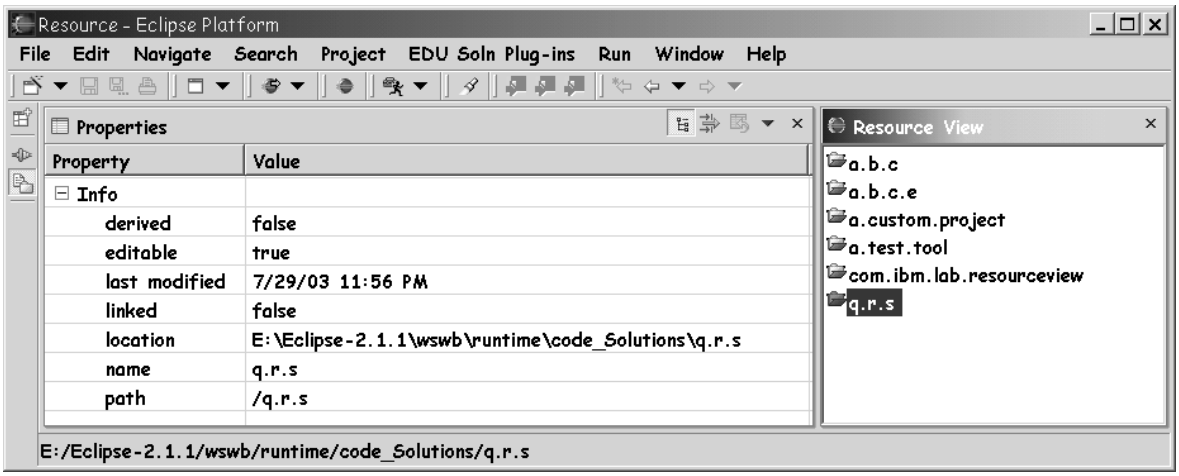


Figure 7-4
Properties View Reacting to Selection Shared by View Part

How does it work? The `IResource` object implements the `IAdaptable` interface, this allows the `PropertySheet` to ask if the object has an `IPropertySource` adapter. The resource responds with the `ResourcePropertySource` class, which provides the content in the Properties view.

If you were to return your own object array in the content provider, your object would have to either implement the `IPropertySource` interface or have an adapter that does.

Step 2. Listen to Selection in Another Workbench Part – the Navigator View
The current viewer input is defined as the workspace root. The Resource View can change the input dynamically; it just needs to have an alternative. You will modify the Resource View so that it listens to the selection in the Navigator view and uses this as input to the viewer.

1. Modify the `createPartControl()` method in the view part by adding the logic shown below at the end of the method:

```
//  React to selection in other workbench parts
getSite().getPage().addSelectionListener(
    "org.eclipse.ui.views.ResourceNavigator", this);
```

The view part can get to the workbench page and add a listener. The listener added above is only for selections in the Navigator view.

A compile error will be generated. This requires that the `ResourceView` class definition implement the required interface.

2. Modify the `ResourceView` class definition so that the `ISelectionListener` interface is implemented:

```
public class ResourceView extends ViewPart implements ISelectionListener {
…
}
```


3. Save the class so that the compile adds the error about the missing abstract method. Use the quick fix support (**Ctrl+1**) to generate the missing `selectionChanged()` method.
4. Customize the `selectionChanged()` method by adding the logic shown below:

```
public void selectionChanged(IWorkbenchPart part, ISelection selection) {
  System.out.println("View heard outside selection " + selection);

  // Set viewer input based on current Navigator view selection
  IStructuredSelection ssel = (IStructuredSelection) selection;
  if (ssel.getFirstElement() instanceof IContainer)
    jcomp.setInput((IContainer) ssel.getFirstElement());
  else
    jcomp.setInput(ViewPlugin.getWorkspace().getRoot());
}
```

Note: Copy the code above from the `JPage\Part3_Selection.jpage` file.

If the selected resource is an `IContainer` it is defined as the input to the viewer. If not, the workspace root is defined as the viewer input. The `selectionChanged()` logic above is simplified by the fact that only one workbench part can be sending selection input. If the listener was added for the selection of any workbench part, the associated logic might have to reflect the possibility of a non-resource selection or even selection in the `ResourceView` itself.

5. Launch the runtime workbench to test the modified Resource View.
6. Open the Package Explorer view and arrange it so it can be seen at the same time as the Navigator and Resource views. If you have not created a Java project in the workspace, do so now.
7. Select a project or folder in the Navigator view; you should see the content of the Resource View change to reflect the new selection. Trace messages will also be written to the console identifying the selected object.

   Select a project in the Package Explorer view, the Resource View should not change. If you deselect the Navigator view selection or select a file, the Resource view input will change back to the workspace root.

   Close the runtime workbench after testing is complete.

Optional Exercise Instructions

Note: Each exercise part in this section can be done independently. You may be able to pick one to do now, or only be able to do these outside of the class timeframe, depending on how long you spent on the primary exercise activity.

Optional Part 1: Add Menu and Task Bar Action to View and Viewer

The current view reacts to the viewer selection contained in its user interface. The JFace architecture allows workbench parts that want to participate in the selection framework the ability to both listen to selections outside their own part and share their current selection with other workbench parts.

Step 1. Create Actions for View Part Menu, Task Bar Items, and Context Menu
The creation of skeleton actions is straightforward. You will copy code to build these

1. Add these fields to the `ResourceView` class. They can be copied from the JPage file.

```
private Action action1;
private Action action2;
private Action doubleClickAction;
```

Use the **Organize Imports** option to correct any missing import statements. When prompted, choose the `org.eclipse.jface.action.Action` option.

2. Add this method to the `ResourceView` class. It should be copied from the JPage file.

```
private void makeActions(final StructuredViewer viewer) {

  action1 = new Action() {
    public void run() {
      showMessage(viewer, "Action 1 executed");
    }
  };
  action1.setText("Action 1");
  action1.setToolTipText("Action 1 tooltip");
  action1.setImageDescriptor(
    PlatformUI.getWorkbench().getSharedImages().getImageDescriptor(
      ISharedImages.IMG_OBJS_INFO_TSK));

 action2 = new Action("Action 2", IAction.AS_RADIO_BUTTON) {
    public void run() {
       if (this.isChecked())
          showMessage(viewer, "Action 2 executed - now checked");
       else
          showMessage(viewer, "Action 2 executed - now unchecked");
    }
 };
 action2.setToolTipText("Action 2 tooltip");
 action2.setChecked(false);
 action2.setImageDescriptor(
    PlatformUI.getWorkbench().getSharedImages().getImageDescriptor(
       ISharedImages.IMG_OBJS_TASK_TSK));

  doubleClickAction = new Action() {
    public void run() {
      ISelection selection = viewer.getSelection();
      Object obj = ((IStructuredSelection) selection).getFirstElement();
      showMessage(viewer, "Double-click detected on " + obj.toString());
    }
  };
}
```

Note: Copy the code above from the `JPage\Xtra_Part1_Actions.jpage` file.

Use the **Organize Imports** option to correct any missing import statements. The compile errors will be corrected in the next step.

3. Add the `showMessage()` method, it is referenced by the actions created in the previous step.

```
private void showMessage(StructuredViewer viewer, String message) {
  MessageDialog.openInformation(
    viewer.getControl().getShell(),
    "Sample View",
    message);
}
```

Note: Copy the code above from the `JPage\Xtra_Part1_Actions.jpage` file.

Use the **Organize Imports** option to correct any missing import statements.

4. Modify the `createPartControl()` method in the view part by adding the logic shown below at the end of the method:

```
// Create actions for use in menus and the tool bar
makeActions(jcomp.getViewer());
```

The view part can get to the workbench page and add a listener. The listener added above is only for selections in the Navigator view. If the selected resource is an `IContainer` it is defined as the input to the viewer. If not, the workspace root is defined as the viewer input.

You cannot test this code yet; the actions need to be added to the view and viewer first.

Step 2. Get and Populate the View Menu and Tool Bar

1. Add this method to the `ResourceView` class:

```
private void contributeToActionBars() {
  IActionBars bars = getViewSite().getActionBars();

  IMenuManager manager1 = bars.getMenuManager();
  manager1.add(action1);
  manager1.add(new Separator());
  manager1.add(action2);

  IToolBarManager manager = bars.getToolBarManager();
  manager.add(action1);
  manager.add(action2);
}
```

Note: Copy the code above from the `JPage\Xtra_Part1_Actions.jpage` file.

Use the **Organize Imports** option to correct any missing import statements. When prompted, choose the `org.eclipse.jface.action.Separator` option.

2. Modify the `createPartControl()` method in the view part by adding the logic shown below at the end of the method:


```
// Add actions to the view menu and the tool bar
contributeToActionBars();
```

3. You can test the view now if you want (open it if it was not already open). There will be a menu pull down and two tool bar icons you can select. They map to the same actions (`action1, action2`), and do nothing beyond showing a dialog to say they are here (using the `showMessage()` method).

Step 3. Create and Populate the Viewer Context Menu

1. Add this method to the `ResourceView` class:

```
private void hookContextMenu(StructuredViewer viewer) {
  MenuManager menuMgr = new MenuManager("#PopupMenu");
  menuMgr.setRemoveAllWhenShown(true);

  menuMgr.addMenuListener(new IMenuListener() {
    public void menuAboutToShow(IMenuManager manager) {
      manager.add(action1);
      manager.add(action2);

      // Other plug-ins can contribute there actions here ("additions")
       manager.add(new Separator(IWorkbenchActionConstants.MB_ADDITIONS));
    }
  });

  Menu menu = menuMgr.createContextMenu(viewer.getControl());
  viewer.getControl().setMenu(menu);
}
```

Note: Copy the code above from the `JPage\Xtra_Part1_Actions.jpage` file.

Use the **Organize Imports** option to correct any missing import statements. When prompted, choose the `org.eclipse.swt.widgets.Menu` option.

Note that in the `menuAboutToShow()` method you could be reacting to the current viewer selection to determine what context menu actions to add.

2. Modify the `createPartControl()` method in the view part by adding the logic shown below at the end of the method:

```
// Add actions to the view menu and the tool bar
hookContextMenu(jcomp.getViewer());
```

You can test the view now if you want. It now has a context menu for entries in the viewer with two actions you can select:

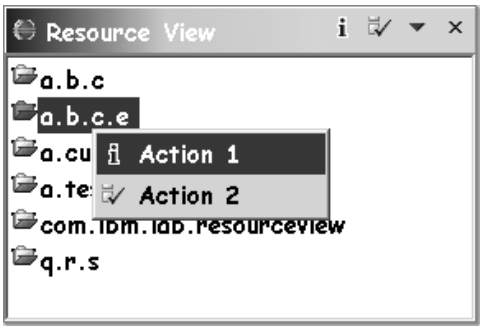


Figure 7-5
Context Menu for Viewer

These actions map to the same actions used in the menu and tool bar.

Close the runtime workbench after testing is complete.

Step 4. Share Context Menu with the Workbench

1. Modify the `hookContextMenu()` method in the view part by adding the logic shown below at the end of the method:

```
// Register viewer menu with workbench
getSite().registerContextMenu(menuMgr, viewer);
```

2. Test the view now. The context menu has entries for actions from your view part and from other plug-ins who's contributions map to the type of object being shown in the viewer:

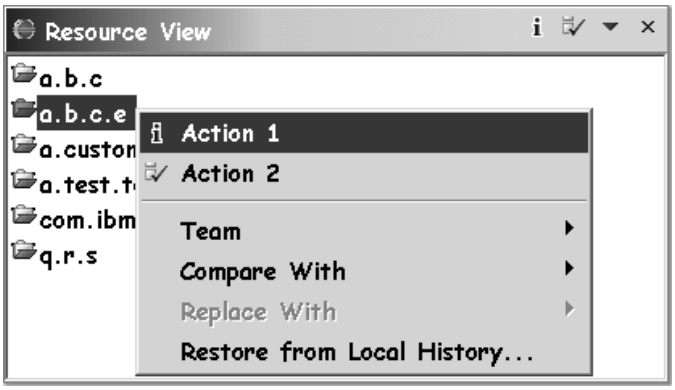


Figure 7-6
Context Menu for Viewer

Note: The actions shown on the context menu will depend on what plug-ins you have configured at test time. Some of the class solutions also contribute to the object shown in the viewer.

Do you remember what type of object this is? Hint: Go see the `getElements()` method in the content provider.

If the content provider returned an object that was not known to others, then even if the viewer menu was registered with the workbench, object contributions would not find an object match and therefore not be shown. Contributions that targeted the view by id would still be shown (discussed further in the contributions topic).

Close the runtime workbench after testing is complete.

Optional Part 2: Add Sorter and Filter Support to the Viewer

Viewers can have an active sorter and any number of filters defined to control how the elements returned by the content provider are displayed by the viewer.

You will now define and add a sorter and a filter to the viewer you have created. This can be done without having completed Optional Part 1, but you will not be able to do all of the last step where you link the use of the filter to the toggle action.

Step 1. Define a Sorter Class and Use it in the Viewer

1. Before you begin, you might want to test the current code so that you can compare the current sort order of entries in the view with what will be displayed after you define the sorter. Here is an example of a project with several folders and files displayed in the default order:

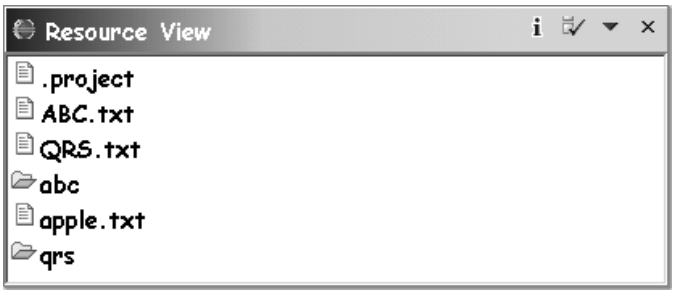


Figure 7-7
Default Sort Order for a Table Viewer

   The default order is as the objects were returned by the content provider. If there is a defined order, it is determined by the `members()` method that was used in the content provider to get the viewer elements.

   Close the runtime workbench after testing is complete.

2. Add an inner class named `ResourceSorter` to the `ResourceView`. This inner class should extend `ViewerSorter`. No fields or methods are required at this point so this is all that is required:

```
// Sorter for Viewer
  public class ResourceSorter extends ViewerSorter {
  }
```

   Use the **Organize Imports** option to correct any missing import statements.

3. Add the sorter to the viewer by adding the logic shown below to the end of the `createPartControl()` method:

```
// Add sorter to viewer
jcomp.getViewer().setSorter(new ResourceSorter());
```

4. Test the view now. The viewer content should now be sorted:

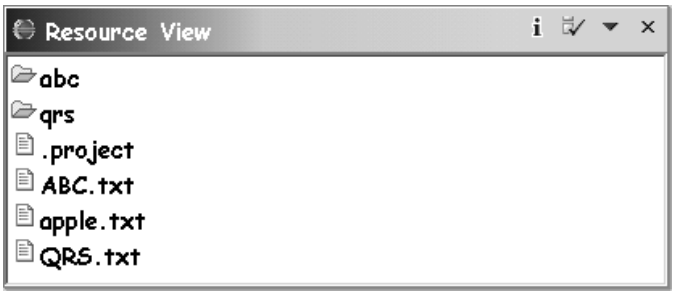


Figure 7-8
Table Viewer with Sorted Content

The sort order is defined just by adding the sorter. As you can see the sort order is not based on the resource name. By default the sorter uses the value returned by the content provider's `getText()` method.

The default `getText()` method returns the `toString()` value for the resource. For the list above this would be:

```
F/a.sorted.project/abc
F/a.sorted.project/qrs
L/a.sorted.project/.project
L/a.sorted.project/ABC.txt
L/a.sorted.project/apple.txt
L/a.sorted.project/QRS.txt
```

So the sorted order is based on the `toString()` value which just so happens to put folders first.

Close the runtime workbench after testing is complete.

Step 2. Add Category Support to the Viewer
The sorter can refine the sort process by assigning categories to each element. The default is that all elements are in the same category. You will now adjust the sorter to define three categories.

1. Add the following fields to the `ResourceSorter` class in the `ResourceView`:

```
int dotted_cat = 0;
int container_cat = 1;
int file_cat = 2;
```

2. Add this `category()` method to the `ResourceSorter` class in the `ResourceView`:

```
public int category(Object element) {
  // Category defined by type (Container/File/File starting with a .)
  if (element instanceof IContainer)
    return container_cat;
  else {
    IFile file = (IFile) element;
    if (file.getName().startsWith("."))
      return dotted_cat;
    else
      return file_cat;
  }
}
```


Note: Copy the code above from the `JPage\Xtra_Part2_Sorter_Filter.jpage` file.

Use the **Organize Imports** option to correct any missing import statements.

3. Test the view now. The viewer content should now be sorted within categories:

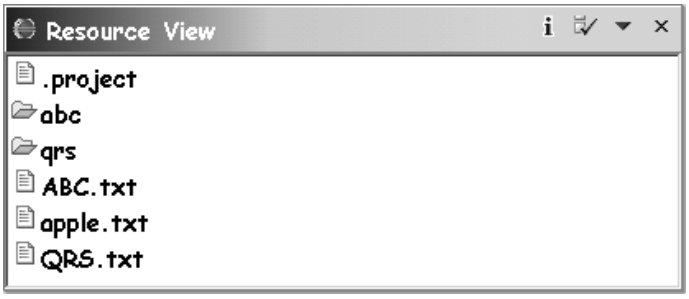


Figure 7-9
Table Viewer with Content Sorted by Category

The sort order is defined just by adding the sorter. As you can see the sort order is not based on the resource name. By default the sorter uses the value returned by the

Close the runtime workbench after testing is complete.

Review the `ViewerSorter` superclass for more information on how you can further control sorting. Methods of interest include:

- `isSorterProperty()` – called as part of the `viewer.update()` logic to determine if a change to viewer content should trigger a re-sort.
- `compare()` – called to determine sort order between two elements. You can override this method to implement customized sort processing.

Step 3. Define and Integrate a Viewer Filter
A filter can be used to determine which elements are actually displayed in the viewer. In this step you will define a filter and add it to the viewer.

1. Add the following field to the `ResourceView`:

```
// field to reference filter in action logic
private ResourceFilter filter = new ResourceFilter();
```

The compile error will be resolved in the next task.

2. Add an inner class named `ResourceFilter` to the `ResourceView`. This inner class should extend `ViewerFilter` and implement the required `select()` method:

```
// Filter for Viewer
public class ResourceFilter extends ViewerFilter {
  public boolean select(Viewer viewer, Object parentElement, Object element) {
     // Return true for all but dotted files (starts with a .)
     if (element instanceof IFile) {
        IFile file = (IFile) element;
```



```
        if (file.getName().startsWith("."))
           return false;
        else
           return true;
     } else
        return true;
  }
}
```

Note: Copy the code above from the `JPage\Xtra_Part2_Sorter_Filter.jpage` file.

Use the **Organize Imports** option to correct any missing import statements.

3. Add the filter to the viewer using one of these techniques:

   - Hardcode the filter by adding the logic shown below to the end of the `createPartControl()` method:

```
// Add filter to viewer
jcomp.getViewer().addFilter(filter);
```

   - Add the filter as part of the toggle action (`action2`) created during ***Optional Part 1.*** Adjust the `action2` definition in the `makeActions()` method to look like this:

```
action2 = new Action("Action 2", IAction.AS_RADIO_BUTTON) {
   public void run() {
      if (this.isChecked())
         // Add filter to viewer
         viewer.addFilter(filter);
      else
         // Remove filter to viewer
         viewer.removeFilter(filter);
   }
};
```

Note: Copy the code above from the `JPage\Xtra_Part2_Sorter_Filter.jpage` file.

4. Test the view now. When the filter is active the viewer content will not include dotted files:

Figure 7-10
Table Viewer without Dotted File Content

Optional Part 3: Modify Viewer to Use Multi-Column Table

The JFace component used in the view part creates a default table viewer. This viewer has one column in the table. You can implement a multi-column viewer but you have to create the table first and use it when creating the viewer. In this part you will do that, and adjust the content provider and label provider logic to support multiple columns.

Step 1. Create Table with Columns and Pass to JFace Component for Viewer Creation

1. Modify the `createPartControl()` method to pass a table to the `JFaceComponent` when creating the viewer. Replace this logic (`jcomp = new JFaceComponent(parent);`) with:

```
Table table = createTableControl(parent);
jcomp = new JFaceComponent(table);
```

   Use the **Organize Imports** option to correct any missing import statements.

2. Implement the following `createTableControl()` method:

```
private Table createTableControl(Composite parent) {
  Table table = new Table(parent,
        SWT.H_SCROLL | SWT.V_SCROLL | SWT.MULTI | SWT.FULL_SELECTION);
  table.setLinesVisible(true);

  TableLayout layout = new TableLayout();
  table.setLayout(layout);

  table.setHeaderVisible(true);
  String[] STD_HEADINGS = { "Resource Name", "Resource Data" };

  layout.addColumnData(new ColumnWeightData(5, 40, true));
  TableColumn tc0 = new TableColumn(table, SWT.NONE);
  tc0.setText(STD_HEADINGS[0]);
  tc0.setAlignment(SWT.LEFT);
  tc0.setResizable(true);

  layout.addColumnData(new ColumnWeightData(10,true));
  TableColumn tc1 = new TableColumn(table, SWT.NONE);
  tc1.setText(STD_HEADINGS[1]);
  tc1.setAlignment(SWT.LEFT);
  tc1.setResizable(true);

  return table;
}
```

   Note: Copy the code above from the `JPage\Xtra_Part3_Table_Columns.jpage` file.

   This logic adds two columns with columns headers to the table. The columns are given appropriate information to size appropriately in the table layout (`ColumnWeightData`).

   Use the **Organize Imports** option to correct any missing import statements. When prompted, choose the option that begins with `org.eclipse`.

3. Test the view now. The custom table the view now has two columns:

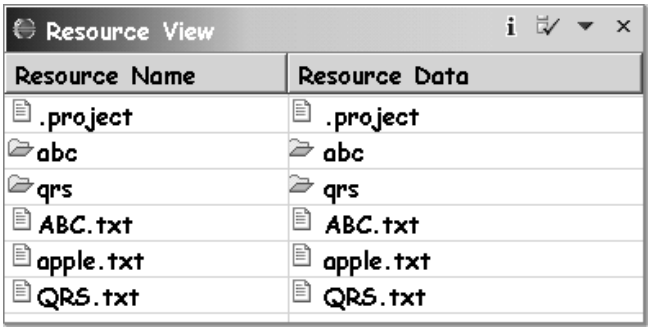


Figure 7-11
Table Viewer with Two Columns

Both columns have the same content, this is because you have not customized the label provider to return values based on the current column index.

Step 2. Create a Label Provider that Supports Multiple Columns

1. Modify the `getColumnText()` method in the `LabelProvider` to return an alternate value for the second column:

```
public String getColumnText(Object obj, int index) {
  // If an IResource then process for name
  if (obj instanceof IResource) {
     IResource res = (IResource) obj;
     if (index == 0)
        return res.getName();
     else
        return res.toString();
  }
  return getText(obj);
}
```

2. Modify the `getColumnImage()` method in the `LabelProvider` to return a null value for the second column:

```
public Image getColumnImage(Object obj, int index) {
  if (index == 0)
     return getImage(obj);
  else
     return null;
}
```


3. Test the view now. The custom table the view now has two columns with custom content:

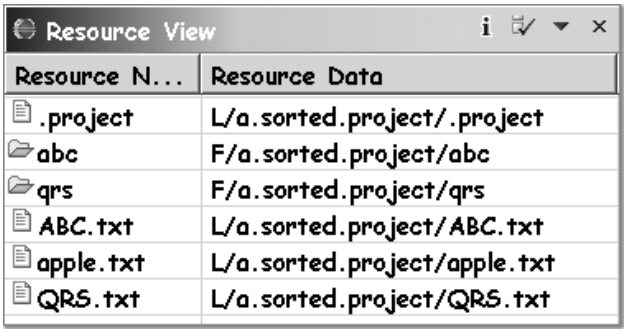


Figure 7-12
Customized Content in the Table Viewer Columns

Both columns now have customized content.

Exercise Activity Review

What you did in this exercise:

- Learned how to define a view part with a simple SWT user interface
- Customized a view part with a viewer-based user interface
- Added workbench part communication to a view part
- Dynamically modified the input to a viewer based on external selection
- Defined menu and task actions for a view part
- Added a context menu to a viewer
- Implemented sort and filter support for a viewer
- Implemented a multicolumn table user interface in a viewer

Exercise 8 Editor Development

This exercise takes you through the process of defining and implementing a new editor. The editor will have a custom user interface created using a JFace table viewer and be able to open, display, and modify the contents of an appropriate resource in a workspace project.

Introduction

This exercise will lead you through the process of developing a basic editor (as opposed to a JFace Text editor, which is designed to edit source code).

Exercise Concepts

Eclipse makes building editors easier by providing an editor framework. You concentrate on building the unique behavior for your editor and the framework handles the common behavior.

This exercise starts with the basics of creating an editor. Many of the implementation steps are the same as those described in the Views discussion and exercise. While the roles that views and editors fulfill for the user are quite different, you will find that from an implementation viewpoint, they are very similar. Let us summarize the classes that will be involved:

ClassName	Description
com.ibm.lab.msseditor.core	
IMiniSpreadsheetListener	Notification interface between MiniSpreadsheet and their interested parties.
MiniSpreadsheet	Simple model for a "mini-spreadsheet" containing text/integer cell values.
MiniSpreadsheetRow	Represents a single row of a mini-spreadsheet.
com.ibm.lab.msseditor.ui	
MiniSpreadsheetContentProvider	Mediate between the table viewer's requests for data and the underlying mini-spreadsheet model.
MiniSpreadsheetEditor	Editor Class
MiniSpreadsheetImages	Convenience class for the UI plug-in's image descriptors.
MiniSpreadsheetLabelProvider	Map between a mini-spreadsheet and its displayable label.
MiniSpreadsheetUIPlugin	The main Plugin class
com.ibm.lab.msseditor.ui.actions	
AppendRowAction	Action to append a new row to the mini-spreadsheet.
ChangeAlignmentAction	Action to change the column alignment of the mini-spreadsheet editor.
ClearAllAction	Action to clear all rows in the mini-spreadsheet.
MiniSpreadsheetEditorAction	Common superclass for all mini-spreadsheet editor actions.
MiniSpreadsheetEditorActionBarContributor	Coordinate the addition of actions on behalf the mini-spreadsheet editor.
MiniSpreadsheetRowActionFilter	Action filters allows an action delegate to be hidden/disabled without having to create an instance of its target action. This comes in particularly handy when a plug-in extension isn't yet loaded, but its contributed actions must reflect current state.
MiniSpreadsheetRowActionFilterFactory	Adapter factory to support basic UI operations for MiniSpreadsheetRow.
RemoveRowAction	Action to remove the selected row(s) from the mini-spreadsheet.
ShowTotalAction	Action to display the sum of all integer cells of the mini-spreadsheet.

Note: Not all these classes can be found in the template project source folder. Some of the classes in the `com.ibm.lab.msseditor.ui.actions` package will be copied into the package during the exercise.

Skill Development Goals

At the end of this lab, you should have an understanding of how to create an editor for use in Eclipse.

Exercise Setup

Before you can begin this exercise, you must have access to the `com.ibm.lab.MSSEditor` template plug-in project. This project should already have been imported into your workspace. If not, import the plug-in from the exercise templates location on your workstation.

Exercise Instructions

You will develop a "mini-spreadsheet editor. You will declare an editor extension and define a toolbar button and pull-down menu actions to perform calculations. When finished, the mini-spreadsheet editor will look like this:

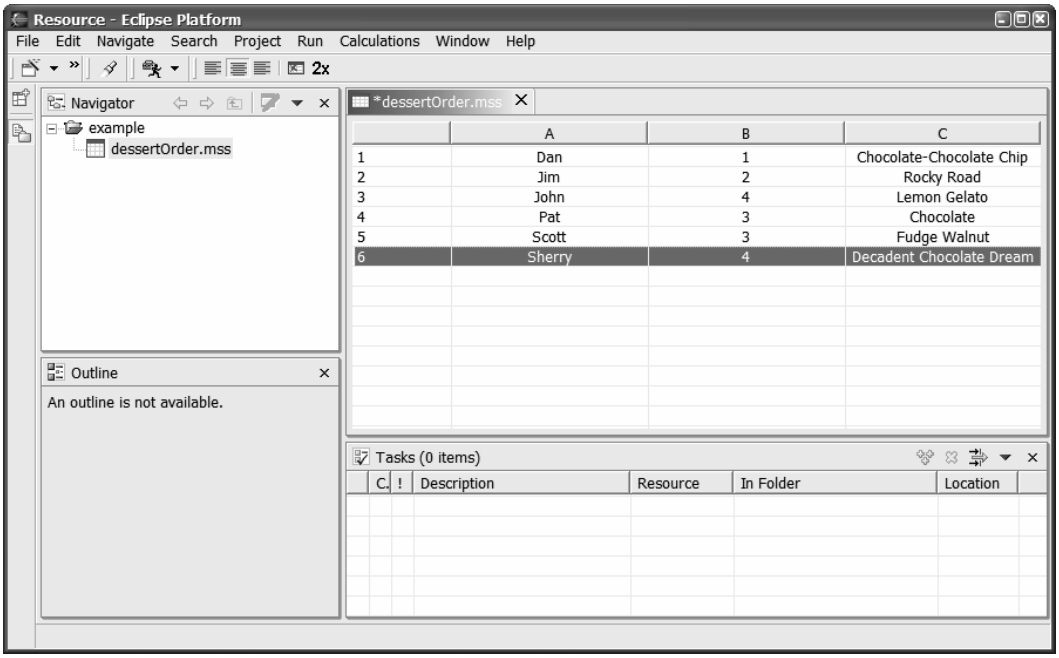


Figure 8-1
Completed Mini-Spreadsheet Editor

These are the steps to be followed in this exercise to implement an editor:

1. Implement an editor part (extension and class).
2. Create a customized user interface.
3. Instantiate the model and manage user modifications of the editor input.
4. Connect the model with the user interface.
5. Save the input to the editor.
6. Add editor actions.

Part 1: Implement an Editor Part

In addition to the normal attributes of `id`, `name`, and `class`, an editor extension also defines an `extensions` attribute. This attribute is a comma separated list of file extensions, these identify the types of files this editor accepts as input. The `name` attribute will be shown in the **Open With** menu choices presented when the context menu for a resource. When the user selects a file with an *.mss extension, the open choice **Mini-Spreadsheet** will correspond to the `MiniSpreadSheetEditor` implementation of an editor.

Step 1. Declare the Editor Extension

1. Edit the `plugin.xml` file in the `com.ibm.lab.MSSEditor` project you imported earlier. Select the Extensions tab. You will now specify the information needed for the editor extension.
2. Define the editor extension.

   Select the **Add…** button. Select **Generic Wizards > Schema-based Extensions**. Press **Next**. Scroll down the list, select the extension point for new wizards, **org.eclipse.ui.editors**, and Press **Finish**.

3. Select the new entry and use the **New > Editor** option to add an `editor` entry. This entry identifies the new editor you will create.

   In the Properties view, enter these values for the attributes `extensions`, `icon`, `id`, and `name`:

Property	Value
extensions	mss
filenames	
icon	icons/spreadsheet.gif
id	com.ibm.lab.minispreadsheeteditor
launcher	
name	Mini-Spreadsheet

   Note: The editor extension can also be defined by copying the extension from the code snippet in `P1_Implement_Editor.jpage`.

Step 2. Generate the Editor Class

The PDE can generate classes for many extension types. You will now use the PDE to generate a class for your editor. This class will extend `EditorPart` and implement the basic editor structure.

1. Return to the extensions page of the `plugin.xml` file to generate the wizard class using the PDE and select the Mini-Spreadsheet (editor) entry in the list.
2. In the Properties View, generate the class by selecting the continuation entry (...) in the class field.

   In the **Java Attribute Editor**, specify that you want to generate a new Java class. Use a class name of `MiniSpreadsheetEditor` and `com.ibm.lab.msseditor.ui` as the package name (you can use the Browse button to select the package). Let the wizard

open an editor on the class after it is created. Leave the source folder and package name at their default settings.

3. Select **Finish** to generate the class.

   When complete, the **Java Attribute Editor** will open an editor on the `MiniSpreadsheetEditor` class. A set of methods will have been generated by the **Java Attribute Editor** wizard.

4. Save the `plugin.xml` file.

Step 3. Complete and Test the Bare-Bones Implementation of a Basic Editor

1. Add an SWT control to the Editor's user interface in the `createPartControl()` method. Choose any SWT control content you want. For fun, try using the SWT Layouts example view to generate a bit of SWT code you can paste into the method. You will have to modify the SWT Layout view generated code to use the composite passed to the `createPartControl()` method instead of the `shell` referenced included in the generated code.

   Use the **Source > Organize Imports** option (**Ctrl+Shift+O**) to add the required import statement(s). Be sure to select the SWT table when you have a choice.

   Your `createPartControl()` method should now look something like this (created using the FillLayout page in the SWT Layout view):

```
public void createPartControl(Composite parent) {
  RowLayout rowLayout = new RowLayout();
  FillLayout fillLayout = new FillLayout ();
  parent.setLayout (fillLayout);
  List list0 = new List (parent, SWT.BORDER);
  list0.setItems (new String [] {"Item 1", "Item 2", "Item 3"});

  Tree tree1 = new Tree (parent, SWT.BORDER);
  TreeItem treeItem1 = new TreeItem (tree1, SWT.NONE);
  treeItem1.setText ("Item1");
  TreeItem treeItem2 = new TreeItem (tree1, SWT.NONE);
  treeItem2.setText ("Item2");

  Table table2 = new Table (parent, SWT.BORDER);
  table2.setLinesVisible (true);
  TableItem tableItem1 = new TableItem (table2, SWT.NONE);
  tableItem1.setText ("Item1");
  TableItem tableItem2 = new TableItem (table2, SWT.NONE);
  tableItem2.setText ("Item2");
}
```


Note: You can copy the code above from the `P1_Implement_Editor.jpage` file.

2. Complete the required setup in the `init()` method.

   An editor must define the site and input as these values are used by the framework. Add this logic to the `init()` method:

```
public void init(IEditorSite site, IEditorInput input)
  throws PartInitException {
    setSite(site);
    setInput(input);
}
```

3. Launch the runtime workbench to test the view user interface.

   If required, create a new project and then copy the `test.mss` file from the `com.ibm.lab.MSSEditor` source project to the project in the runtime instance of the workbench.

   Double click on the file to open the Mini-Spreadsheet editor. The editor that opens will look like this if you used the code shown above:

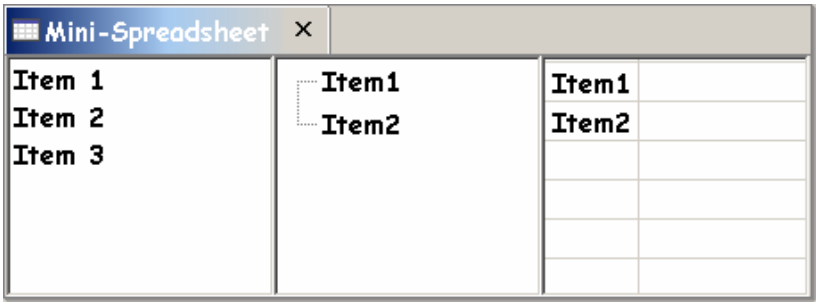


Figure 8-2
Mini-Spreadsheet Editor with Placeholder User Interface

You now have a class that can be reworked to be a functional Editor part.

The placeholder SWT controls will be replaced by a user interface composed of a table viewer and associated logic than can read the input resource and map it to the user interface. Additional editor functions, such as the **Save** and **Save as…** menu options will also be implemented later.

Close the runtime workbench after testing is complete.

Part 2: Create a Customized User Interface

As you have seen, the `createPartControl` method is used to create your editor's user interface. This can be done with SWT widgets, JFace viewers, or a mix of these elements. This editor needs a user interface that can display tabular data. It will be constructed from a JFace table viewer and other parts as provided in the template project.

Step 1. Customize the Editor User Interface
In this step, you will create a custom user interface which at its core has a table viewer which will display the contents of the input resource.

1. Add fields to `MiniSpreadSheetEditor` that will represent the user interface viewers and widgets, support user interface display and control, and represent the model contained in the input resource. These fields can be copied from the `P2_Implement_User_Interface.jpage` file provided in the template project.

   Use the context menu: **Source > Organize Imports** to fix the missing class references.

2. Add the helper methods provided in the `P2_Implement_User_Interface.jpage` file to the `MiniSpreadSheetEditor.` These methods are used to help create the table viewer that will be used in the user interface.

   Copy the code snippet from the provided JPage file to add these methods:
   - `initializeTableLayout()`
   - `initializeCellEditors()`
   - `createTable()`
   - `createColumns()`
   - `setAlignment()`
   - `getDefaultAlignment()`

   Use the context menu: **Source > Organize Imports** to fix the missing class references. Be sure to choose the JFace and SWT packages when prompted about duplicate classes.

3. Revise the `createPartControl()` method so that it creates the table, the table viewer, and other control elements of the editor's user interface. The revised `createPartControl()` method logic will reference the helper methods that you just added to the class.

   Replace the existing method with the code snippet from the provided JPage file and use the context menu: **Source > Organize Imports** to fix the missing class references.

   The user interface has been defined using a JFace viewer. This includes a Table and Cell editor viewers. This extends the concepts covered in *Exercise 6, Defining a JFace Component*.

   Update the `setFocus` method so that when the editor opens it puts focus on the table in the user interface. Add this code to the method:
   ```
   table.setFocus();
   ```
   The editor can now be tested, but it will only display an empty user interface which allows you to click on the column headers. The editor has not yet used the input to load the model that will be used by the editor.

4. Save the `MiniSpreadSheetEditor` and test it if you wish.

Part 3: Model Instantiation and Modification Management

Like views, editor parts often display content from a model. For an editor the model is typically based on the contents of an input file. Eclipse creates an instance of your editor class, and calls its `init()` method to pass it the input.

Your editor must be prepared to react to changes that might occur to both the model and the input behind the model. To be aware of when changes occur, your editor must listen for modifications made to the model and the resource used as input to create the model. When a change is detected the editor content must either be marked as modified (so it can be saved later) or somehow adjust to a change that occurred directly to the input resource (outside the editor's control).

The model used by the editor should also be capable of reporting changes that may have occurred. The model provided to you as part of the template project has this support; you will add model listener logic to the editor.

Eclipse includes a means of detecting changes to workspace resources, called resource change listeners. This notification strategy is inherent to the nature of the Eclipse user interface, since it allows—and even encourages—multiple views of the same resource. Your editor, if it is based on a resource, must react to the possibility that your model's underlying resource could be modified or deleted by means outside your editor's control, bypassing your model's change events. A robust editor implementation must address these possibilities. You will add resource change listener support to the editor and logic that deals with any direct resource modification that may occur.

Step 1. Identify when the Editor Makes Modifications to the Input

When model changes occur, the editor must know that this has happened so that saving the changes in the resource is possible. The generated editor structure includes an isDirty() method. When this method returns true, the option to save the file is enabled. If a save is requested, the editor's `doSave()` method will be called.

1. Add logic to the editor to return the current value of the `isDirty` field in the `isDirty()` method:

```
public boolean isDirty() {
  return isDirty;
}
```

   The `isDirty` field was added earlier to support the determination of when editor modifications have been made.

2. Add the `setIsDirty()` method to set the `isDirty` field to true so that this method can be called when a modification has been made.

```
protected void setIsDirty(boolean isDirty) {
  this.isDirty = isDirty;
  firePropertyChange(PROP_DIRTY);
}
```


This method can be copied from the P3_Manage_Model.jpage file.

Step 2. Instantiate the Model Using the Editor Input

The editor must inspect the input provided in the `init()` method and either use the input to populate the user interface or reject the input if it is not the type of input that is expected. If the input is unacceptable, the editor must throw a `PartInitException` exception.

You will now add logic to your editor to support the creation of a model from the input and then add logic to inspect and use the input (the tasks are ordered this way to avoid unnecessary compile errors).

1. Add the `setContents()` method to the editor. This method can be copied from the `P3_Manage_Model.jpage` file.

   The `setContents()` method takes the input and both initialized the `isDirty` field and creates a `MiniSpreadsheet` model instance. The statements commented out in the `setContents()` method will be added later.

2. Update the `init()` method and add its helper method (`init()` with a different signature) to the editor. These methods can be copied from the `P3_Manage_Model.jpage` file. The statement commented out in the additional `init()` method will be added later. Use **Source > Organize Imports** to fix the missing class references. Choose `java.io` when prompted.

   By checking the type of the input, the `init()` method defends against what would probably be a programming error, that is, someone reusing your editor for unexpected input. The editor could also reject the input if an I/O error occurs, or it finds that the input is corrupt or invalid.

   When the input is rejected, all processing stops. The workbench then informs the user with an error dialog showing the message associated with the `PartInitException` exception that was thrown.

Step 3. Implement Support for Reacting to Model Modifications

To react to changes that could occur to the model, the editor must be listening for these changes. When changes are detected, the editor must be told.

1. Adjust the `MiniSpreadSheetEditor` definition so that it implements the `IMiniSpreadsheetListener` interface. Use **Source > Organize Imports** to fix the missing class reference.
2. Add the methods that are required to support the `IMiniSpreadsheetListener` interface to the `MiniSpreadSheetEditor` class. The `rowsChanged()` and `valueChanged()` tell the editor by changing the `isDirty` setting.

   These methods can be copied from the `P3_Manage_Model.jpage` file.

3. Return to the `setContents()` method and remove the comment markers from the lines that remove and add the editor as a `IMiniSpreadsheetListener` listener.

   This will remove any existing listener and add a new listener to the model as the model is created.

Step 4. Implement Support for Reacting to Direct Modification of the Resource
As changes can occur directly to the resource the editor must be listening to the workspace and react accordingly.

1. Adjust the `MiniSpreadSheetEditor` so that it implements the `IResourceChangeListener` interface.
2. Add the `resourceChanged()` method, which is required to support the `IResourceChangeListener` interface, to the `MiniSpreadSheetEditor` class. This method can be copied from the `P3_Manage_Model.jpage` file. Use **Source > Organize Imports** to fix the missing class references.

   The implementation of the `resourceChanged()` method includes the logic that controls how the editor will react to specific events.
3. Return to the `init(IEditorSite site, IStorageEditorInput editorInput)` method and remove the comment markers from the `ResourcesPlugin.addChangeListener(…)` statement. This will add the editor as a resource change listener when it is initialized.

Part 4: Connect the Model with the User Interface Viewer

You have made numerous changes to the editor, but it would still look the same if you tested it. This is because the viewer used for the user interface is not yet communicating with the model.

Step 1. Creating Content and Label Providers
The purpose of the content provider is to be the mediator between the viewer and model. A content provider reacts to a new input reference and then:

- Aligns itself with the new input (adds a listener)
- Removes references to the old input (removes any existing listener)

The Label provider for the viewer is responsible for showing a `String` and icon representation in the viewer's control and therefore must at least implement these methods:

- `getText(element)`
- `getImage(element)`

The actual implementation for the content and label providers required by the `MiniSpreadSheetEditor` has been provided. See the `MiniSpreadsheetContentProvider` and `MiniSpreadsheetLabelProvider` classes.

1. Connect the `MiniSpreadsheetContentProvider` and `MiniSpreadsheetLabelProvider` classes to the viewer by adding this logic to the end of the `createPartControl()` method:

```
tableViewer.setContentProvider(new MiniSpreadsheetContentProvider());
tableViewer.setLabelProvider(new MiniSpreadsheetLabelProvider());
```

Step 2. Associating the Viewer with the Model
The `setInput(object)` method is used to identify the input that should be used for a viewer. The input links your model to the viewer as `object` is a reference to your model. The `setInput()` method triggers a call to the content provider's, `inputChanged()` method so that the content provider knows what to use when returning elements to the viewer.

1. Add this line of code to the end of the the `createPartControl()` method to identify the input for the viewer:
```
tableViewer.setInput(miniSpreadsheet);
```
2. Test the current `MiniSpreadsheetEditor` implementation. Start the run-time workbench instance and using the file copied to a project earlier, open the editor. You should see something like this:

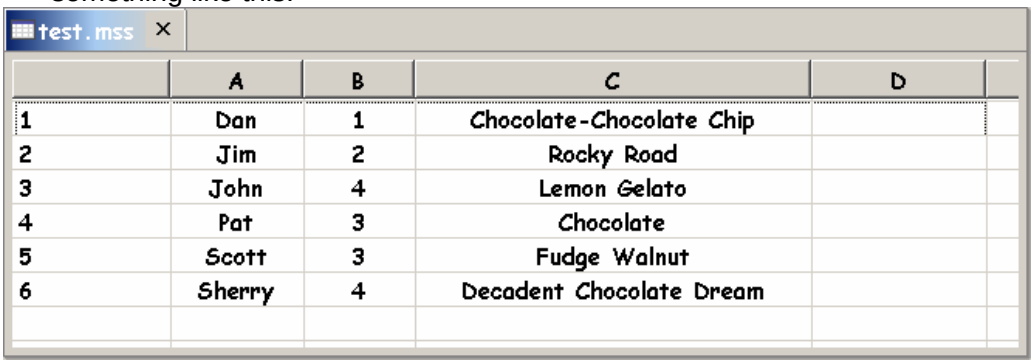


Figure 8-3
Mini-Spreadsheet Editor with Customized User Interface and Input

Notice we have not implemented the **Save** and **Save as…** functions. The framework will call the `doSave()` method, but the logic to actually save the content is not there yet. You will get to that part next part of the exercise.

Part 5: Saving the Editor Input

It is your resource so you perform the **Save** request when required. You can also decide if you want your editor to support **Save as…** processing.

Step 1. Add Save Processing

Save processing is called when required (the `isDirty()` method returns `true`) and is implemented in the `doSave()` method.

1. Add the `saveContents()` method to the `MiniSpreadsheetEditor` editor (after the `doSave()` method is a good place). Use the code snippet found in the P5_Save.jpage file.
2. Replace the `doSave()` method in the `MiniSpreadsheetEditor` editor using the code snippet found in P5_Save.jpage. Organize imports to add the required import statements and ignore the message about the `doSaveAs()` method. This error will be corrected in the next step.

Step 2. Add Save As… Processing

If you want to support **Save as…** processing you need to override the `isSaveAsAllowed()` method to return true, and then implement the `doSaveAs()` method.

In the `doSaveAs()` method, your editor presents a Save As dialog, followed by the same processing as your `doSave()` method.

If the **Save as…** is successful, and the desired result is that your editor changes its input to the last saved location, your editor should create a new editor input referencing the new location and call `setEditorInput()`, followed by `firePropertyChange(PROP_INPUT)`. If appropriate, you should also update your editor's tab text with `setTitle()`.

1. Add the `createNewFile()` method to the `MiniSpreadsheetEditor` editor (after the `doSaveAs()` method is a good place). Use the code snippet found in the P5_Save.jpage file. Organize imports to add the required import statements.
2. Add the `doSaveAs(String message)` method to the `MiniSpreadsheetEditor` editor (after the `doSaveAs()` method is a good place). Use the code snippet found in the P5_Save.jpage file. Organize imports to add the required import statements.
3. Update the `doSaveAs()` method in the `MiniSpreadsheetEditor` to add this line of code:
   ```
   doSaveAs(null);
   ```
4. Adjust the `isSaveasAllowed()` method so it returns `true`.
5. Test the revised `MiniSpreadsheetEditor` implementation. Start the run-time workbench instance and using the file copied to a project earlier, open the editor. The file should now support saving modifications to the file system and **Save as…** processing.

Part 6: Adding Editor Actions

To simplify the management of a common toolbar and menu, editors include an editor action contributor as part of their definition. This class is specified in the `contributorClass` attribute of the `<editor>` tag. The class identified must implement thet `IEditorActionBarContributor` interface.

As you might expect, the action contributor is responsible for contributing the editor's actions to the toolbar and main menu bar. Eclipse informs the action contributor to add its actions by calling its `init()` method. The standard implementation of the `IEditorActionBarContributor` interface, `EditorActionBarContributor`, implements an `init()` method that calls convenience methods your subclass may override, `contributeToToolbar()` and `contributeToMenu()`.

A single instance of the action contributor class is created when an editor of a given type is first opened, and it will remain until the last one is closed. That is, it will be shared among all editor instances having the same id. The `setActiveEditor()` method is fired with a parameter that represents the active editor instance. This allows the action contributor class to know what editor to interact with when actions are invoked.

Note: The viewers used in an Editor can have their own pop up menus, just as viewers in a view. The steps to create and register them are the same.

Step 1. Define the Action Contributor

The MiniSpreadsheetEditorActionBarContributor class is provided in the lab template. Take a moment and review that class now.

1. Copy the predefined action contributor and context menu actions that have been provided for you from the `add_to_ui.actions_later` folder to the target package (`com.ibm.lab.msseditor.ui.actions`).

   These classes were hidden to avoid the display of compile errors while you finished the editor code to this point.

2. Edit the `plugin.xml` file to add the `contributorClass` attribute to editor extension. This will identify that the `MiniSpreadsheetEditorActionBarContributor` class is the action contributor.

The modified editor entry in the extension should now look like this:

```
<editor
   name="Mini-Spreadsheet"
   icon="icons/spreadsheet.gif"
   extensions="mss"
   class="com.ibm.lab.msseditor.ui.MiniSpreadsheetEditor"
   contributorClass="com.ibm.lab.msseditor.ui.actions.MiniSpreadsheetEditorActionBarContributor"
   id="com.ibm.lab.minispreadsheeteditor">
</editor>
```

3. Add the `getMiniSpreadsheet()` method to the `MiniSpreadsheetEditor` class. Use the code snippet found in the P6_EditorActions.jpage file.

Step 2. Add Additional Actions to the Editor

1. Update the `MiniSpreadsheetEditor` class by adding the `createContextMenu()` and `fillContextMenu()` methods. Use the code snippet found in the P6_EditorActions.jpage file.

   These, as named, will create and fill the context menu for the viewer. Organize imports to reflect required references (choosing SWT when prompted).

2. Add a call to the `createContextMenu()` method at the bottom of the `createPartControl()` method. This invokes the two methods you just added.

3. Time to Test!!! There should now be a set of global menu and task actions visible in the Workbench along with an active and functional context menu for the editor:

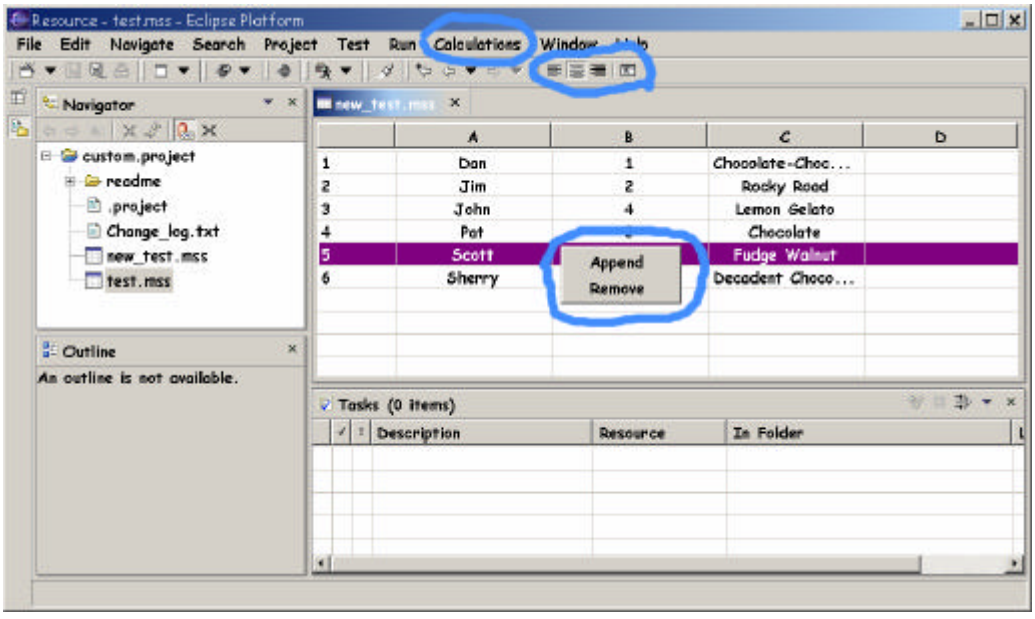


Figure 8-4
Completed Mini-Spreadsheet Editor with Action Contributions and a Context Menu

Exercise Activity Review

Building a customized editor can be a time-consuming task. The Eclipse user interface frameworks make it easier by providing the common editor behavior. During this exercise you:

- Implemented and Editor extension to Eclipse and used JFace controls to create the user interface for an editor.
- Learned how to instantiate the model using the editor input.
- Handled user modifications of the editor input and saved the modified resource as required.
- Added an action contribution class and viewer associated actions to an editor.

Exercise 9 Perspective Development

This exercise takes you through the process of defining a perspective extension and a new perspective. Each has their place, this way you know how to use both.

Introduction

Perspectives are the organizing unit of the user interface. The define what views are visible, where they are with respect to an editing area, and what global actions, and shortcuts for wizards, views, and other perspectives should exist.

As you add tools to an existing workbench platform you have to decide if your components should join an existing perspective or be defined as part of a new perspective. There are extensions points that allow you to customize and existing perspective or create a new one, depending on your needs.

Exercise Concepts

The exercise allows you to do one or both parts. You can add some of the code you have developed to an existing perspective or create a new perspective; or both if you have the time.

Skill Development Goals

This exercise looks at the definition and use of perspectives in the workbench.

Exercise Setup

Before you can begin this exercise, you must have access to the `com.ibm.lab.perspective` template plug-in project. This project should already have been imported into your workspace. If not, import the plug-in from the exercise templates location on your workstation.

Exercise Instructions

Each part can be done by itself, you can do one or both.

Part 1: Define and Test a Perspective Extension

In Part 1 you will define a perspective extension, which takes only XML to exist. The perspective extension will adjust the Resource Perspective to add these items:

- First plug-in action (already done)
- Custom project wizard shortcut
- Resource view shortcut
- Resource view location placeholder (for when opened)

The instructions that follow are light, you should understand work required for each task by now.

Step 1. Define the PerspectiveExtension

1. Edit the `plugin.xml` and add a new extension, the `perspectiveExtension`.

   Use the **Add…** button to open the **New Extension** wizard to do this.

2. Select the new entry and add a `perspectiveExtension` entry. Yes, the same thing twice it seems. The first is the extension, the second is the entry to identify the perspective you want to extend.

   In the Properties view, enter a **targetID** of `org.eclipse.ui.resourcePerspective`.

3. Add an `actionSet` entry under the new `perspectiveExtension` entry. This type of entry is used to define a global action set that should be part of the extension.

   In the Properties view, enter `com.ibm.lab.plugins.firstplugin.actionSet` as the **id**. This adds the associated actions to the default menu tree for the resource perspective.

   Note: This action set was defined with an attribute of `visible="true"`, which forces the action set into all perspectives. To actually have the `perspectiveExtension` reference work you would have to adjust the action set definition in the `firstplugin` to be `visible=″false″`.

4. Add a `perspectiveShortcut` entry under the new `perspectiveExtension` entry. This allows you to add entries to the **Window > Open Perspective** shortcut menu.

   In the Properties view, enter `com.ibm.lab.perspective.eduPerspective` as the id. This would add this perspective to the **Window > Open Perspective** shortcut menu

in the resource perspective, if it existed. It will exist if you do the next part of this exercise.

5. Add a `newWizardShortcut` entry under the new `perspectiveExtension` entry. This allows you to add entries to the **File > New** shortcut menu.

   In the Properties view, enter `com.ibm.lab.newWizard.customProject` as the id. This adds the custom project wizard to the **File > New** shortcut menu in the resource perspective.

6. Add a `viewShortcut` entry under the new `perspectiveExtension` entry. This allows you to add entries to the **Window > Show View** shortcut menu.

   In the Properties view, enter `com.ibm.lab.resourceview.resourceview` as the id. This adds the resource view to the **Window > Show View** shortcut menu for the resource perspective.

7. Add a `view` entry under the new `perspectiveExtension` entry. This allows you identify a location for where a view should be in the perspective. You can then either force the display of the view when the perspective is opened or just let the user open the view but have it be shown in the location you have identified.

   In the Properties view, enter these values for the attributes:

Property	Value
id	com.ibm.lab.soln.resourceview.resourceview
ratio	
relationship	stack
relative	org.eclipse.ui.views.TaskList
Tag name	view
visible	false

   This stacks the view on the existing **Tasks** view and leaves it out of the perspective for now. When opened, the view will stack on the **Tasks** view.

Step 2. Test the PersectiveExtension Implementation

1. Launch the runtime workbench.
2. If required, open the Resource perspective. Reset the perspective (**Window > Reset Perspective**) so that it reflects your design and not any customization that may have been done earlier.
3. Check to see if the first plug-in action set is part of the menu (from your very first exercise).
4. Check to see if the wizard shortcut is on the **File > New** menu.
5. Check to see if the view shortcut is on the **Window > Show View** menu.
6. Open the **Edu: Resource View**. Does it join the perspective where you expected?
7. Try adjusting the `view` settings in the `perspectiveExtension` entry. See if you can change the location of the view. Try ratio values of `.50` or `.25` when something other than `stack` has been selected as the `relationship` value.
8. Close the runtime workbench after testing is complete.

Part 2: Implement a Custom Perspective

If required, you can build a new perspective to integrate the components defined in your tool. This should be done when you need an arrangement that is significantly different from an existing perspective or the type of activity in your tool does not fit anywhere else.

Step 1. Define a New Perspective

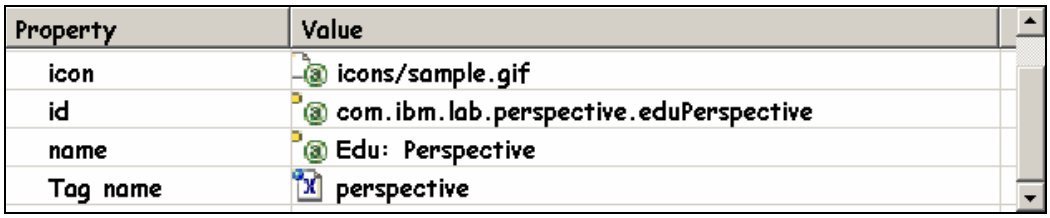
A new perspective takes more than XML; it takes XML and Java code. Define a perspective extension and generate the Java class.

1. Edit the `plugin.xml` file in the `com.ibm.lab.perspective` project you imported earlier. Select the Extensions tab. You will now specify the information needed for the perspective extension.
2. Define the perspective extension.

   Select the **Add…** button. Select **Generic Wizards > Schema-based Extensions**. Press **Next**. Scroll down the list, select the extension point for new wizards, **org.eclipse.ui.perspectives**, and Press **Finish**.

3. Select the new entry and add a `perspective` entry. Yes, the same thing twice it seems. The first is the extension, the second is the entry to identify the new perspective you will create.

   In the Properties view, enter these values for the attributes:



Property	Value
icon	icons/sample.gif
id	com.ibm.lab.perspective.eduPerspective
name	Edu: Perspective
Tag name	perspective

Step 2. Generate Perspective Class

The PDE can generate classes for many extension types. You will now use the PDE to generate a class for your perspective.

4. Return to the extensions page of the `plugin.xml` file to generate the wizard class using the PDE and select the Edu: Perspective (perspective) entry in the list.
5. In the Properties View, generate the class by selecting the continuation entry (...) in the class field.

   In the **Java Attribute Editor**, specify that you want to generate a new Java class. The class name is `CustomPerspective` and you want to let the wizard open an editor on the class after it is created. Leave the source folder and package name at their default settings.

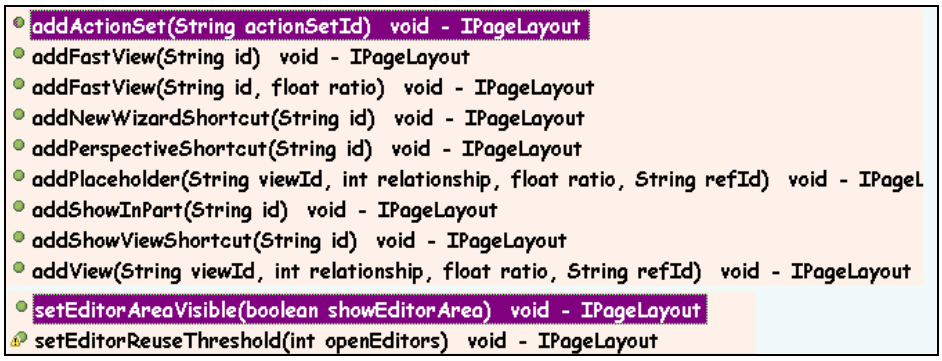
6. Select **Finish** to generate the class.

   When complete, the **Java Attribute Editor** will open the `CustomPerspective` class in an editor. The `createInitialLayout()`method was generated by the **Java Attribute Editor** wizard.

7. Save the `plugin.xml` file.

Step 3: Customize the Perspective Class

The `createInitialLayout()`method needs to be customized to define the perspective. As you can see from a portion of the code assist list, the layout passed to the method has many methods available that support customization of the perspective:



1. Add these fields to the CustomPerspective class.

```
public static final String ID_FIRSTACTION =
  "com.ibm.lab.plugins.firstplugin.actionSet";
public static final String ID_RESOURCEVIEW =
  "com.ibm.lab.resourceview.resourceview";
public static final String ID_PROJECTWIZARD =
  "com.ibm.lab.newWizard.customProject";
public static final String ID_EDU_JAVA_PERSPECTIVE =
  "org.eclipse.jdt.ui.JavaHierarchyPerspective";
```

These fields represent the IDs of the tool components that will be referenced in this perspective. It makes it easier to define them up front.

2. Create an editor area reference and make the editor area invisible. Many of the layout decisions are based on the editor area. Add this code to create a variable for the editor area as known to this layout and hide the editor area as well:

```
// Get the editor area
String editorArea = layout.getEditorArea();
layout.setEditorAreaVisible(false);
```

3. Add wizard, view menu, and perspective shortcuts to the layout by adding this code next:

```
// Add shortcuts
layout.addNewWizardShortcut(ID_PROJECTWIZARD);
layout.addShowViewShortcut(IPageLayout.ID_TASK_LIST);
layout.addPerspectiveShortcut(ID_JAVA_PERSPECTIVE);
```


4. Create a folder for the top left side of the layout with two views stacked in the folder by adding this code:

```
// Top left: Resource Navigator and Bookmarks view
IFolderLayout topLeft =
  layout.createFolder("topLeft", IPageLayout.LEFT, 0.3f, editorArea);
topLeft.addView(IPageLayout.ID_RES_NAV);
topLeft.addView(IPageLayout.ID_BOOKMARKS);
```

   Organize imports to add the missing `IFolderLayout` reference.

5. Add the Outline view to the bottom left side of the layout by adding this code:

```
// Bottom left: Outline view
layout.addView(IPageLayout.ID_OUTLINE, IPageLayout.BOTTOM, 0.6f, "topLeft");
```

6. Add the Resource view under the editor area by adding this code:

```
// Right: Edu: Resource View
layout.addView(ID_RESOURCEVIEW, IPageLayout.BOTTOM, 0.4f, editorArea);
layout.addView(
    IPageLayout.ID_PROP_SHEET, IPageLayout.BOTTOM, 0.4f, ID_RESOURCEVIEW);
```

   Note that the Properties view is added with a reference to the Resource view.

7. Add the Properties view and a placeholder for the Tasks view under the Resource view on the right side of the layout by adding this code:

```
// Right bottom: Property Sheet view and Tasks view placeholder
IFolderLayout bottomRight = layout.createFolder(
    "bottomRight", IPageLayout.BOTTOM, 0.4f, ID_RESOURCEVIEW);
bottomRight.addView(IPageLayout.ID_PROP_SHEET);
bottomRight.addPlaceholder(IPageLayout.ID_TASK_LIST);
```


8. Launch the runtime workbench to test the new perspective. If you did the first part of the lab, there will be a shortcut to open the new perspective. Otherwise, find Edu: Perspective in the full list and open it. The initial display, given an existing project, should looks something like this after the Tasks view has been opened manually:

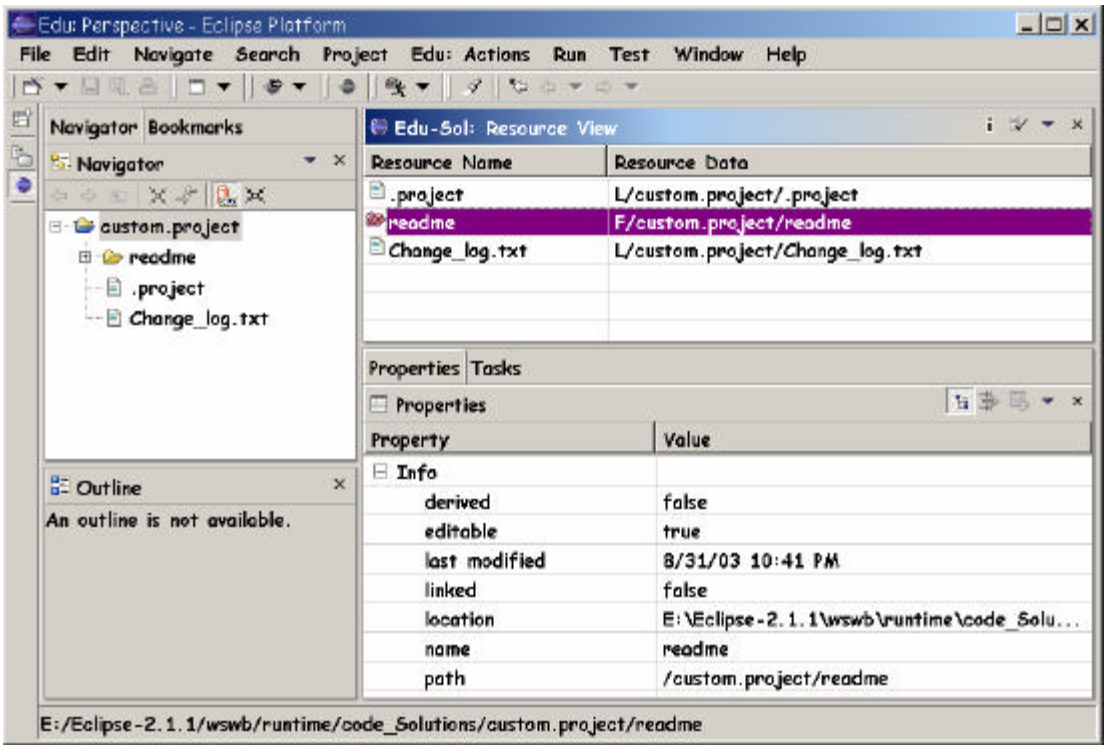


Figure 9-1
Custom Edu: Perspective

The wizard, perspective, and view short cuts should all be there as well. The Resource view is below the invisible editor area. If an editor was launched it would push the right side views down.

Note: The Resource View solution was used to capture the image above.

Close the runtime workbench after testing is complete.

Exercise Activity Review

What you did in this exercise:

- Learned how to extend an existing perspective so that your tool components and shortcuts are known.
- Learned who to define a new perspective.

Exercise 10 Working with Resource Extensions

This exercise walks through a resource extension solution. This solution includes a Nature, Builder, and Marker.

Introduction

Resource extensions allow you to define new Natures, Builders, and Markers. You can use these extensions to customize a project, extend existing marker function, or implement new kinds of makers that are specific to your tool.

Exercise Concepts

The exercise begins with a plug-in project solution that you should have imported into your workspace. The function and usage scenarios for each type of extension is discussed. The solution is demonstrated and the code that implements the solution reviewed.

Skill Development Goals

This exercise looks at the definition and use of resource extensions so that you can understand:

- How Natures are defined and associated with a project
- How builders are defined and associated with a project
- How builders react to resource changes
- How markers are defined and how they can build on exiting marker function.

Exercise Setup

Before you can begin this exercise, you must have access to the `com.ibm.lab.soln.resources` solution plug-in project. This project should already have been imported into your workspace. If not, import the plug-in from the exercise templates location on your workstation.

Working with Natures and Builders

The implementation of a nature and builder requires that you define them as extensions in your plug-in, provide an implementation of these extensions, and then either create projects that include the nature or add the nature to existing projects. The `plugin.xml` defines the extensions and the code is provided in the `com.ibm.lab.soln.resources.nature_builder` package.

This example focuses on the use of the `CustomNature` and `ReadmeBuilder`.

Part 1: Running the Solution

The Natures and Builders solution demonstrates the function and techniques available when defining natures and using them to customize the behavior of a project by adding builders.

To run the solution launch the run-time instance of Eclipse (Run > Run As > Run-time Workbench).

Step 1. Create a Project with the CustomNature

1. Create a project that includes the CustomNature by either:
   - Using the provided new project wizard to create a new project which will include the nature.

     Open the wizard using File > New > Project, selecting Soln: Resource Wizards and then New Project (w/CustomNature). Enter a project name. Select Finish to create the project. Message dialogs will confirm that the ReadmeBuilder and CustomNature have been added to the project.

   - Using the provided Add Custom Nature project pop-up menu choice to add the nature to an existing project.

     If you already have a project, you can add the CustomNature to that project by selection the project in the Navigator view and using the Soln: Resource Tools > Soln: Add CustomNature pop-up menu option.

In either case, when the CustomNature is added to the project, its configure method is invoked to give it the opportunity to prepare the project. In this example, the configure method adds the ReadmeBuilder to the project.

You can check if the nature and associated builder have been added to a project by opening the .project file or using one of the contributed project actions.

Once the nature has been added to the project, and it has configured the builder, the builder will be invoked during build processing when changes have been made to resources in the project..

2. Add a readme folder to the project.

   The ReadmeBuilder will process only the anyname.readme files it finds in a readme folder.

   Note: The project structure action can be used to create the readme folder.

3. Add an anyname.readme file to the readme folder. If the Perform build automatically on resource modification Workbench preference is selected, a build will be triggered.

   The ReadmeBuilder build method is triggered when a build occurs. The build method determines what kind of build is being processed: full or incremental.

   - A full build uses the processFull method to find out if a readme folder exists and processes any files with a .readme file type.
   - An incremental build uses the processDelta method to determine of the IResourceDelta contains any changes in a readme folder. If yes, this subset of the IResourceDelta is passed to the the ReadmeVisitor whose visit method then processes the IResourceDelta to find any files with a .readme file type.

   The end result will be a corresponding file with a file type of .html created for each modified .readme file, where the content of the .readme file is wrapped in simple HTML.

Step 2. Configure the Builder so that it Builds another Simple Project

1. Create a new simple project (simple.project), one that does not include the CustomNature or ReadmeBuilder, and then use the project structure action can to add a readme folder to this new project.
2. Implement support for .readme file type processing for the simple.project by teaching the existing ReadmeBuilder associated with the first project you created to reach out and process this new project. The ReadmeBuilder includes support for processing referenced projects, that is, those that are referenced by the project that includes the builder.

   So, to do this, open the Properties dialog for the first project and on the Project References page select the simple.project entry.

3. Add an anyname.readme file to the readme folder in the simple.project. Modify and save this file.

   The ReadmeBuilder for the first project will be invoked as it has resource deltas that it can process. Not for the project the builder is defined for, but for the projects it has

registered interest in processing. The ReadmeBuilder will detect the anyname.readme file modification and create a matching .html file.

Part 2: Roadmap to the Solution

This solution includes a nature and builder extension, a new project wizard, and several pop-up action contributions that demonstrate how natures and builders can be used to customize projects and process resources. the use of the workspace API. The extension definitions can be found in the plugin.xml. The classes that implement these extensions are shown below.

Extension	Implementation
Nature: Custom Nature	The nature is implemented by the `CustomNature` class.
Builder: Readme Builder	The builder is implemented by the `ReadmeBuilder` class.
New project wizard: New Project (w/CustomNature)	The wizard is implemented by the `NewProjectWizard` class.

When the nature is configured as part of a project it adds the builder. The builder implements a simple resource transformation by reacting to new or modified .readme files and creating associated .html files. This processing is implemented by the ReadmeVisitor class.

The `com.ibm.lab.soln.resources.nature_builder` package includes several contributed actions in the Navigator view that allow you to interact with the projects and their defined natures and builders:

Figure 10-1
Nature and Builder Context Menu

Not all of these actions were referenced in the example instructions above. These actions provide the following function:

Pop-up Action	Description / Implementation
Soln: Add Custom Nature	Adds the `CustomNature` to the selected project. `ActionAddCustomNature` implements this function. Responds with the success or failure of the request.
Soln: Remove Custom Nature	Removes the `CustomNature` from the selected project. This action is only shown if the selected project has the customer nature. `ActionRemoveCustomNature` implements this function. Responds with the success or failure of the request.
Soln: Add Builder	Can be used to add the `ReadmeBuilder` directly to a project. This action will only add the builder if the `CustomNature` exists as part of the project (and the builder was previously removed). `ActionAddBuilderToProject` implements this function. Responds with the success or failure of the request.
Soln: List Builders	Lists the builders associated with the selected project. `ActionListBuilders` implements this function. Responds with the success or failure of the request.
Soln: Remove Builder	Can be used to remove the `ReadmeBuilder` from the selected project. This action can remove a builder added by the `CustomNature`. `ActionRemoveBuilderFromProject` implements this function. Responds with the success or failure of the request.

Working with Markers

You can choose to reuse the markers defined as part of Eclipse or define your own. New marker types can do one of two things:

- Build on existing maker definitions and behavior
- Be implemented as specialized markers where all behavior is implemented as part of your tool logic.

This marker example defines a new marker type (`com.ibm.lab.soln.resource.recentEdits`) that builds on both the existing Eclipse bookmark and problem marker types. This means that this kind of marker will automatically be visible in the Bookmarks and Tasks views. The `recentEdit` markers are created for resources that have been recently modified.

The plug-in also extends the `org.eclipse.ui.startup` extension point so that it may add the resource change listener used to create recent edit markers.

The `plugin.xml` defines the marker extensions and the code is provided in the `com.ibm.lab.soln.resources.markers` package.

Part 1: Running the Solution

The Markers solution demonstrates the function and techniques available when defining new markers and how they can be associated with resources. The built in behavior that exists when you define a new maker that extends an existing marker is also demonstrated.

To run the solution launch the run-time instance of Eclipse (Run > Run As > Run-time Workbench).

Step 1. Review Recent Edits Marker Support

1. The `recentEdits` marker logic has default processing rules, but if the `com.ibm.lab.soln.dialogs` project is available, these rules can be modified. If you have the `com.ibm.lab.soln.dialogs` plug-in active in the test instance of the Workbench, you can open the Soln: Basic Preference Page and view or modify the setting that determines how many `recentEdits` markers to keep active and if these `recentEdits` markers should be saved in the workspace (custom logic will use the `IMarker.TRANSIENT` attribute to override the `<persistent> value="true" />` setting in the marker extension):

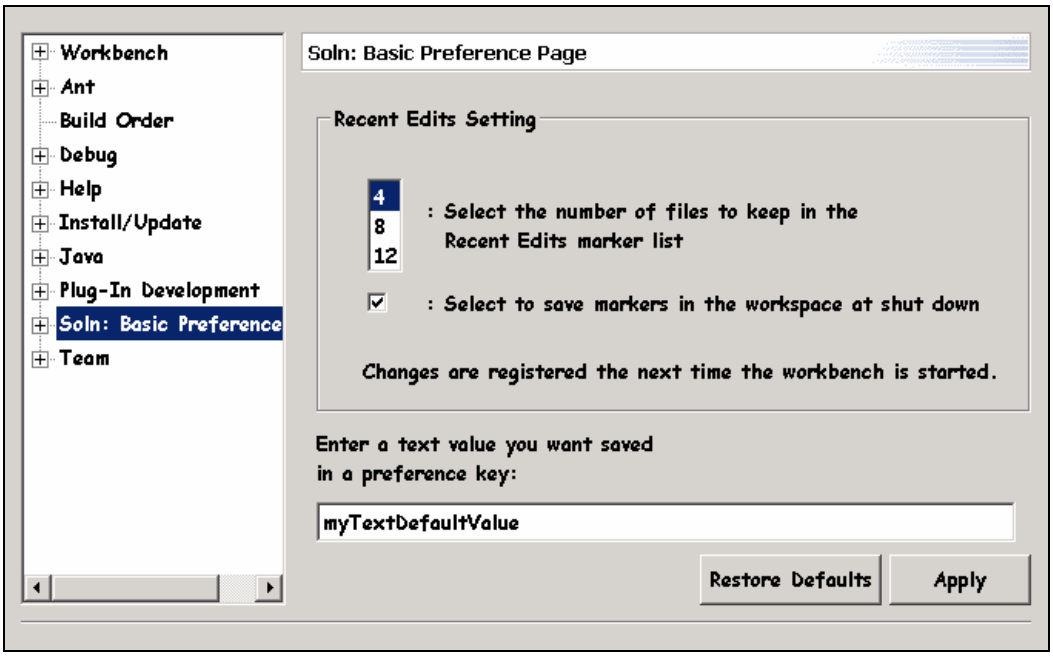


Figure 10-2
Recent Edits Marker Preference Page

If the `com.ibm.lab.soln.dialogs` plug-in is not available in the Workbench instance being tested, a hardcoded value of four is used as the `recentEdits` marker limit and the marker extension definition is used to determine if the marker is persistent.

2. The `com.ibm.lab.soln.resources` plug-in includes an `org.eclipse.ui.startup` extension, this means the plug-in starts when Eclipse starts. Because of this you may already have `recentEdits` markers showing in the Bookmarks and Tasks views:

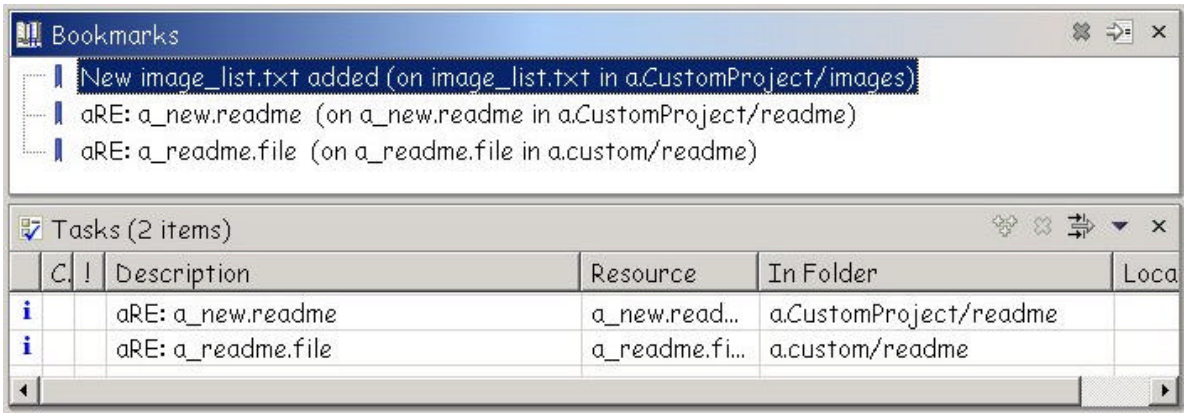


Figure 10-3
Bookmarks and Tasks Views with Recent Edits Markers

Note: If the display of the `recentEdits` markers in the Tasks view is bothersome, you have two choices:

- Sort the list in the Tasks view so that these information items are at the bottom
- Modify the plugin.xml and remove this part of the marker extension definition:
  `<super type="org.eclipse.core.resources.problemmarker"> </super>`

This change will remove the `recentEdit` markers from the Tasks view. They will only be shown in the Bookmarks view.

Step 2. Create Recent Edits Markers

1. If you do not have any `recentEdits` markers showing, then open a file in your workspace and then edit/save the file. If you open/save enough files, you will reach the maximum number of `recentEdits` markers that can be saved. The oldest marker is deleted to make room for a new marker.

   A resource change listener (RecentEditsRCL) is used to detect file modifications and the RecentEdits class manages the recentEdits markers.

2. The `recentEdit` marker support can also be disabled for individual files. If you have the `com.ibm.lab.soln.dialogs` project loaded, you can open the properties page Soln: File Properties, and view or modify the setting that determines if a `recentEdits` marker will be created for the file:

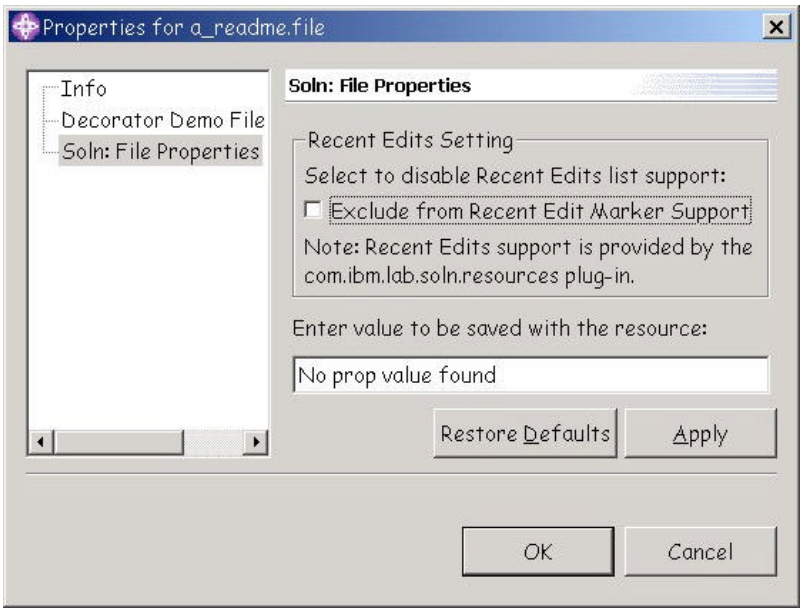


Figure 10-4
Resource Property Page with Recent Edits Control Options

When selected, the exclude setting will cause the recent edits listener to ignore this file during add marker processing.

Part 2: Roadmap to the Solution

This solution includes a marker extension and several pop-up action contributions that demonstrate how natures and builders can be used to customize projects and process resources. the use of the workspace API. The extension definition can be found in the plugin.xml. The classes that use this extension are shown below.

Extension / Object	Implementation
Marker extension: recentEdits	A marker is defined; there is no implementation class.
Resource Change Listener	The resource change listener is implemented by the `RecentEditsRCL` class.
Recent Edits Manager	The recent edits manager is implemented by the `RecentEdits` class.

When the nature is configured as part of a project, it adds the builder. The builder implements a simple resource transformation by reacting to new or modified .readme files and creating associated .html files. This processing is implemented by the `ReadmeVisitor` class.

Exercise Activity Review

What you did in this exercise:

- Learned how natures are added to a project
- Learned how builders are added to a project and how they react to resource changes
- Learned how markers can extend existing marker function

Exercise 11 Developing Action Contributions

This exercise takes you through the process of contributing an action to the Eclipse text editor. The action will be in two forms, as a toolbar action and as a context menu. The action is quite simple. It counts the words in the text selected by the user and displays the count in a simple dialog.

Introduction

The first step in creating a contribution is to specify in the plug-in manifest where the action will be shown in the Eclipse user interface, and specify its label, icon, and enablement conditions. This allows Eclipse to show the action as a menu item, toolbar button, or a context menu item without having to load your code. In other words, Eclipse creates an initial action that "stands in" for your action. Your action is referred to as an **action delegate** and it provides the code that performs the real action if the user chooses it.

The next step is to define your action delegate class using some form of the interface `IActionDelegate`. This action will be loaded, if required, and run when the user selects the corresponding choice. Once your action delegate has been loaded, it takes responsibility for the future appearance, enablement state, and visibility of the action.

Exercise Concepts

This exercise will illustrate how to define actions in your plug-in manifest using two different extension points. You will implement the word counting action as an action delegate and discover a bit more about the lazy loading of plug-ins. When completed, you will have an action on the toolbar and as an editor context menu that produces results that look something like this.

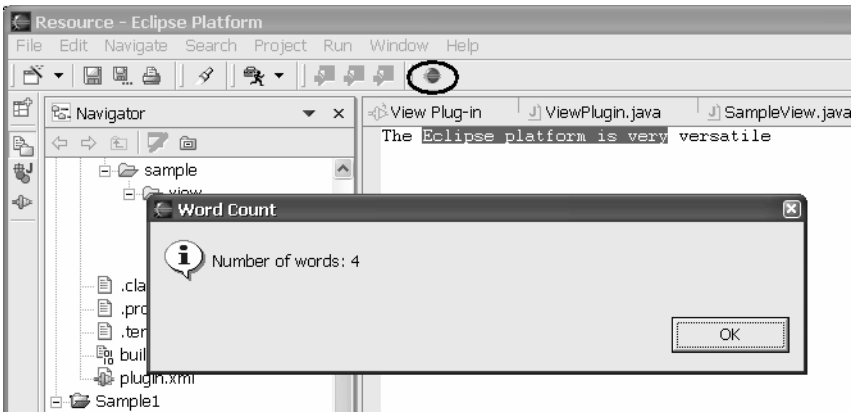


Figure 1
Example of an Editor Contribution to the Workbench Toolbar

Skill Development Goals

- Learn now to use two action contribution extension points: `org.eclipse.ui.editorActions` and `org.eclipse.ui.popMenus`
- Create an action delegate class
- Observe lazy loading of plug-ins

Exercise Setup

Before you can begin this exercise, you must have access to the `com.ibm.lab.contributions` template plug-in project. This project should already have been imported into your workspace. If not, import the plug-in from the exercise templates location on your workstation.

There are four parts to this exercise. You should try to finish the first two parts. The last two parts are quite short, so you probably can finish the entire exercise. The steps defined in each section are continuous. That is, they must be done in sequence. You should not jump ahead or skip steps.

To assist you in the lab you can copy code and XML snippets from file `contributions.jpage` in the `com.ibm.lab.contributions` project template. You will have to provide the necessary import statements. By now you should be familiar with all the JDT tricks to help you.

Part 1: Define the Toolbar Action in your Plug-in Manifest

1. Define your action using the extension point `org.eclipse.ui.editorActions` by adding this XML to your plug-in manifest after the `</requires>` tag.


```
<extension point="org.eclipse.ui.editorActions">
  <editorContribution
    targetID="org.eclipse.ui.DefaultTextEditor"
    id="com.ibm.lab.ec.DefaultTextEditor">
    <action
      label="Word Count"
      icon="icons/sample.gif"
      tooltip="Show word count of selected text"
      class="com.ibm.lab.contributions.TextEditorWordCountAction"
      toolbarPath="Normal/additions"
      enablesFor="+"
      id="com.ibm.lab.ec.TextEditorWordCount">
      <selection
        class="org.eclipse.jface.text.ITextSelection">
      </selection>
    </action>
  </editorContribution>
</extension>
```

What does this mean? In the `targetID` tag we specified that our action applies to the platform text editor. Our action will not appear in other editors. We have given the normal identification using `label`, `icon`, and `tooltip`. The action will appear in the toolbar using the standard path and insertion point of `normal/additions`. The action will be enabled if one or more characters are selected (`enablesFor="+"`) and the selection is a text selection. We insure that the action only works on a text selection by using the selection tag and specify the `ITextSelection` class to filter our action to a range of characters (alternatively, you could have created a trivial action expression using the `enablement` and `objectClass` tags to achieve the same result but this is too simple to bother with that approach). The action delegate class that does the work is `TextEditorWordCountAction`. We will work on that in the next step.

At this point you could test your plug-in. The action would appear in the toolbar because your extension is sufficient to display your action and determine its enablement conditions. Since there is not code to back it up proceed to Part 2.

Part 2: Implement the Word Counting Action and Test it

1. In the package `com.ibm.lab.contributions` create a class named `TextEditorWordCountAction`. It should implement `IEditorActionDelegate` which is the action delegate interface appropriate to our editor extension. It includes a `setActiveEditor` method in addition to the normal `run` and `selectionChanged` methods.
2. Add the following fields to this new class. One will be used to store our text editor. The `WORD_DELIMITERS` field will help us parse the text into words using the standard Java string tokenizer.

```
static final String
  WORD_DELIMITERS = " .,'\"/?<>;:[]{}\\|`~!@#$%^&*()-_+=\n\r";
TextEditor textEditor;
```

3. The class `TextEditorWordCountAction` implements `IEditorActionDelegate`. Editor actions are shared by other instances of the same type of editor. If, for example, three default

text editors are open, they are all using the same editor action instances on the toolbar. This sharing simplifies the Workbench code, since it doesn't have to worry about duplicate entries. When the user activates (or reactivates) an editor, the Workbench invokes the `IEditorActionDelegate.setActiveEditor` method to tell the action which open editor the user is working in. The action saves this editor as a `TextEditor` so it can refer back to it in its `run` method if the user selects the action. To set the active editor in our action, add this line to the `setActiveEditor` method which has passed us the current editor in the parameter `IEditorPart targetEditor`.

```
textEditor = (TextEditor) targetEditor;
```

4. Without going into the details of editors, it is possible to determine what the text selection was directly from the editor. So we don't have to get it using the `selectionChanged` method. We will have some fun with that method later. Let's go straight to the `run` method and get this action cooking.
5. Add the following code to the `run` method. Again, without going into the details of text editors, this code will provide us with the document our editor is working on as an `IDocument` object using the `TextEditor` we saved in the `setActiveEditor` method. Additionally, we retrieve the selected text as an `ITextSelection` object. We now have everything we need to complete our action.

```
IDocument document =
  textEditor.getDocumentProvider().getDocument(textEditor.getEditorInput());
ITextSelection ts =
      (ITextSelection) textEditor.getSelectionProvider().getSelection();
```

6. Let's wrap this up by adding the following code to the end of the run method to count the words in the selected text and display the result. As you can see we have an easy way to get the selected text as a string labeled `text`. We run the text through the `StringTokenizer` class and display the final count using a simple message dialog. We are now ready to test it.

```
int tokenCount;
  try {
    String text = document.get(ts.getOffset(), ts.getLength());
    tokenCount =
      new StringTokenizer(text, WORD_DELIMITERS).countTokens();
  } catch (BadLocationException e) {
    tokenCount = 0;
  }
  MessageDialog.openInformation(
    null,
    "Word Count",
    "Number of words: " + tokenCount);
```

7. Test your action by selecting your plug-in project and menu item **Run > Run As > Run-time Workbench** (or use the **Run** toolbar pulldown). In the test instance of Eclipse create a project and a file with an extension of `.txt` to force the default text editor to open. Your action should

appear. It will be grayed out. When you select some text it will enable. Selecting the action should display results similar to (figure ref) above.

Part 3: Test Eclipse Lazy Loading

1. We stated earlier that we would come back to the `selectionChanged` method. You will add the following code to explore the various states of the user selection. As you can see it retrieves the selected text which is passed as an `ISelection` object. If your `ISelection` object is not null and is an instance of `ITextSelection` you retrieve it into field `ts`. The rest of the code tests the selection and reports its status as well as sets the enablement state of the action. It is important to note that once your action class is active you can change or override the action's state and definitional characteristics that were initially defined in your plug-in manifest.

```
if (selection != null &&
    selection instanceof ITextSelection) {
  ITextSelection ts = (ITextSelection) selection;
  if (ts.getLength() == 0) {
    System.out.println("TextEditorWordCountAction disabled");
    action.setEnabled(false);
  } else {
    System.out.println("TextEditorWordCountAction enabled");
    action.setEnabled(true);
  }
} else {
  System.out.println("TextEditorWordCountAction disabled");
  action.setEnabled(false);
}
```

2. Restart the run-time Workbench again and test your plug-in by opening a file using the default text editor. Select some text. You should not see anything written to the development Workbench console by the `selectionChanged` method. Why is that? Your plug-in is not loaded! Eclipse has exposed your contribution based entirely on the XML in your plug-in manifest. Select some text again and invoke the **Word Count** action; the console should light up with enablement status messages, now that your action delegate is instantiated. Watch the busy host console and your action's toolbar icon as you select and deselect text!

   Another way to know if you plug-in is loaded or not is by inspecting the **Plug-in Registry** view in the test instance of Eclipse. Loaded plug-ins display a "running man" decorator in the plug-in icon.

Part 4: Create a Context Menu Action

1. This is an anticlimactic conclusion of the lab. You will simply redefine your action as context menu action in your editor using the `org.eclipse.ui.popupMenus` extension point and reuse your action delegate class. Add the following extension to your plug-in manifest. Observe that by specifying `menubarPath = "additions"` we are adding the action to the

menu location reserved by the editor for contributions (also known as the insertion point). Also note that is important to keep `ID` values unique between the two action definitions. We used `ec` and `pm` as name qualifiers in the two different extensions.

```
<extension
    point="org.eclipse.ui.popupMenus">
  <viewerContribution
    targetID="#TextEditorContext"
    id="com.ibm.lab.pm.contributions.TextEditorContext">
      <action
        label="Word Count"
        class="com.ibm.lab.contributions.TextEditorWordCountAction"
        menubarPath="additions"
        enablesFor="+"
        id="com.ibm.lab.pm.TextEditorWordCount">
        <selection
          class="org.eclipse.jface.text.ITextSelection">
        </selection>
      </action>
  </viewerContribution>
</extension>
```

2. Test your new action in the test instance of Eclipse. Open a text file using the text editor and click your right-mouse button. You should see the **Word Count** action listed. It will be disabled until a text selection is made. It should work identically to the toolbar action.

Exercise Activity Review

What you did in this exercise:

- Learn now to use two action contribution extension points to create a toolbar and context menu action in an editor
- Created an action delegate class that could be used by both extension points
- Observed lazy loading of plug-ins

Note that the solution `com.ibm.lab.soln.contributions` offers examples of the use of additional action contribution extension points.

Exercise 12 Creating New Extension Points

Introduction

The Workbench is composed of extensible plug-ins. These plug-ins are built using plug-in extension points. In fact, menu items and toolbar items are just plug-in extension of the Workbench UI. Plug-in authors can also implement extension points of their plug-in functions.

Exercise Concepts

This exercise is a case study in creating an extensible plug-in. The goal of this exercise is to first show you how to write an extensible plug-in that offers the possibility for other plug-ins to extend. The second part of the exercise demonstrates how one extends the plug-in created in the first part.

Skill Development Goals

At the completion of this exercise, you should be able to:

- Declare an Extension Point and then to use it with an extension.
- You will learn the declaration of a new Extension point in the markup and the implementation in the markup and in Java.
- Process the markup data
- You will learn to read the markup from the new Extension Point and handle it.
- Extend the new extension point from plug-in in a different project.
- You will learn how to specify classes for the markup and how to delegate tag processing to them.

Exercise Setup

Common exercise setup tasks should have been performed so that the `com.ibm.lab.extensionpoint` project is available for this exercise.

Exercise Instructions

Part 1: Create a new Extension Point

Let's imagine that we are an ISV, developing a couple of different tools as plug-ins for the Workbench. We want to provide a common pulldown item on the application toolbar where all of our tool commands may execute. We decide that this requires a simple framework that can be easily implemented using an extension point. So, we want to create a plug-in with an extension point at the base of this framework. As we build additional tool plug-ins, we use the extension point mechanism to add tool command menu items.

To save you time, the following is already supplied to you:

- A template for `plugin.xml` `in project` `com.ibm.lab.extensionpoint`.

  This template offers the body of the `plugin.xml`.
- The class `com.ibm.lab.extensionpoint.ToolActionsPulldownDelegate`.

  This class is our entrypoint for new commands that we want to customize with the new extension point.
- A utility class `com.ibm.lab.extensionpoint.ToolActionParameters`.

  This class encapsulates the parameters specified on our extension point tag.

Declaring an extension point

1. First have a look at the `extension-point` DTD[1]:

```
<!ELEMENT extension-point EMPTY>
<!ATTLIST extension-point
  name                CDATA #REQUIRED
  id                  CDATA #REQUIRED
  schema              CDATA #IMPLIED
>
```

The `extension-point` tag has no child tags. The attribute `name` is just a descriptive name for the `extension-point` which is displayed by the various Plug-in Development Environment (PDE) dialogs. The `id` attribute is important because it is the key part of the extension point identifier. Extenders use this key to declare the points their plug-ins will extend. It must be unique inside the `plugin.xml` and be a simple token (characters a-z, A-Z, 0-9). The attribute `schema` is not used in workbench runtime but by the PDE; we'll discuss this in more detail later.

2. Declare the new extension point.

Open the plug-in manifest editor by double-clicking `plugin.xml`, then turn to the **Extension Points** page. Define a new extension point by selecting **Add…** and filling out the dialog:

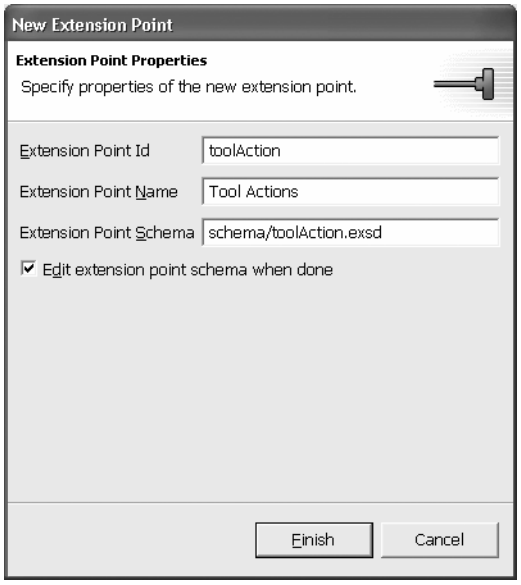


Figure 12-1
New Extension Point Wizard [extpt_01.tif]

[1] See the *Platform Plug-in Developer Guide*, under *Reference* > *Plug-in Manifest* for the complete DTD.

Choose the extension point name carefully, since it will be displayed by PDE dialogs that reference your extension point. Notice that a schema for your new extension is automatically created and stored in the `schema` subdirectory of the project.

Why define a Schema?

Extensions cannot be arbitrarily created. They are declared using a clear specification defined by an extension point, but until now we haven't discussed how to create a specification that can validate that extensions will conform to the extension point's expectations.

Whether specified programmatically via a schema or implicitly via reference documentation, each extension point defines attributes and expected values that must be declared by an extension. In the most rudimentary form, an extension point declaration is very simple. It defines the `id` and `name` of the extension point.

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

By convention, new schemas have the same name as the extension point id with a `.exsd` file extension[2]. They are placed in `schema` directory in your plug-in directory tree. When a new extension point is created in the PDE, the initial schema file will also be created and the schema editor will be opened for editing. The PDE schema editor is based on the same concepts as the plug-in manifest editor. It has two form pages and one source page. Since XML schema is verbose and can be hard to read in its source form, we will use the form pages for most of the editing. The source page is useful for reading the resulting source XML.

3. Add extension point schema information

Since **Edit extension point schema when done** was checked when the extension point was created, the editor below is displayed:

[2] Extension point schema file extensions were changed to *.exsd in version 2.0 to avoid conflicts with XML Schema Definitions.

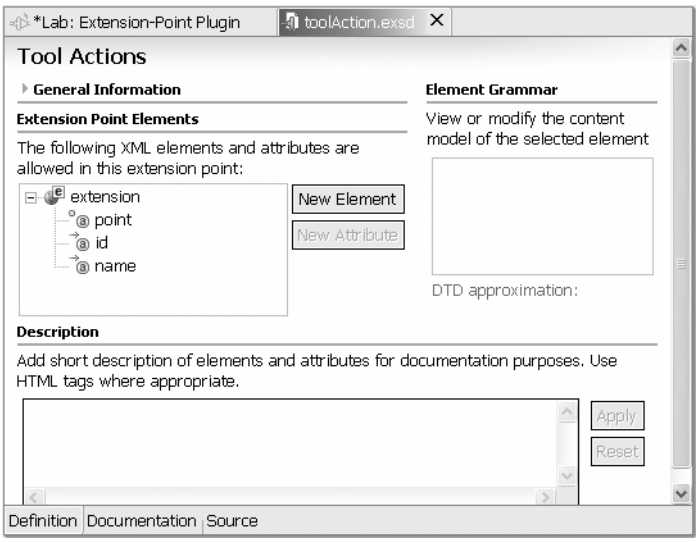


Figure 12-2
Extension Point schema editor [extpt_02.tif]

Here the extension point elements are shown on the left, and their relationship to one another on the right. The top-level `extension` element is shown by default; add the new `tool` element by selecting "New Element" and entering its name in the Properties view. Then define the `tool's` attributes `action` and `label` by selecting `tool` and then the **New Attribute** pushbutton. Specify in the Properties view that the former accepts only those classes that implement the `com.ibm.lab.extension.IToolAction` interface by changing the `Kind` to `java` and entering the interface name as the `basedOn` value of the `action` attribute (it is not a problem that we have not yet defined this interface); change the `Use` to `required`. The PDE will use this information to ensure that only compatible classes or interfaces can be entered for this attribute when defining an extension of the `tool` extension point. Specify in the Properties view that the `label` attribute will accept any string, and the attribute is required.

Turn to the Source page to see what the PDE schema editor has generated. Below is a partial extract:

```
<?xml version='1.0' encoding='UTF-8'?>
<!-- Schema file written by PDE -->
<schema targetNamespace="com.ibm.lab.extensionpoint">
<annotation>
   <appInfo>
      <meta.schema plugin="com.ibm.lab.extensionpoint" id="toolAction" name="Tools actions"/>
   </appInfo>
   <documentation>
      [Enter description of this extension point]
   </documentation>
</annotation>

   <element name="extension">
      <complexType>
         <sequence>
         </sequence>
```



```
         <attribute name="point" type="string" use="required">
            <annotation>
               <documentation>

               </documentation>
            </annotation>
         </attribute>
         <attribute name="id" type="string">
            <annotation>
               <documentation>

               </documentation>
            </annotation>
         </attribute>
         <attribute name="name" type="string">
            <annotation>
               <documentation>

               </documentation>
            </annotation>
         </attribute>
      </complexType>
   </element>
... not shown ...

</schema>
```

Whew! Some plug-in developers might enter source directly into the PDE manifest editor, but here it is clear that the schema editor is saving you a lot of typing!

We've finished defining the elements `extension` (which was provided by default) and its child tag, `tool`.

4. Finish defining the relationship between `extension` and `tool`.

The `extension` element already has a sequence compositor defined; let's use it to finish defining the relationship between `extension` and `menuItem`.

Select the extension element grammar entry, select the Sequence pop-up menu, then **New** > **Reference** > **tool**:

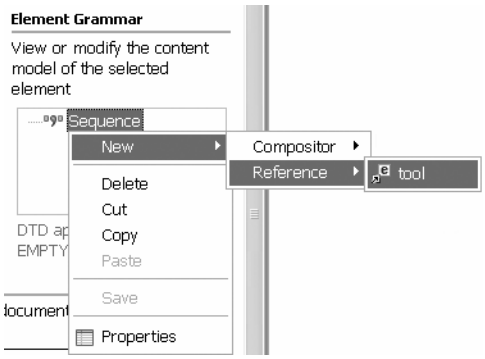


Figure 12-3
Sequence Context Menu [extpt_03.tif]

Then update the cardinality in the `tool` property view within the element grammar:

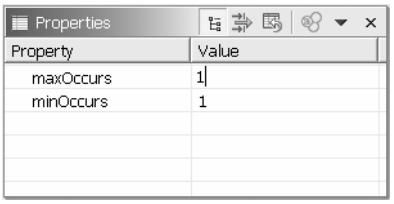


Figure 12-4
Grammar Properties View [extpt_04.tif]

This will update the Element grammar display for the `extension` element as shown below:

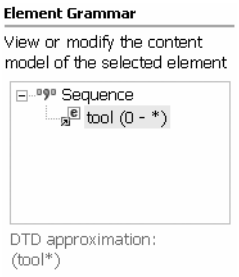


Figure 12-5
Element Grammar Display [extpt_05.tif]

When a PDE developer creates an extension of this extension point, the PDE manifest editor will only allow the definition of `tool` subtags and will prompt for those classes/interface that fulfill the `IToolAction` interface for the `action` attribute.

You can also add in general documentation for the extension point by switching to the Documentation page. Information for individual elements/attributes is entered in the Description text area on the Definition page.

Once we have finished with the documentation, you can take a look at the reference documentation, `toolAction.html`, in the project's "doc" folder. It is built by a PDE builder registered to react to changes in extension point schema files.

Helpful Hint: You can easily reopen the schema editor by double-clicking the .exsd file in your project's schema directory or selecting "Go to File" from the extension point's Properties view:

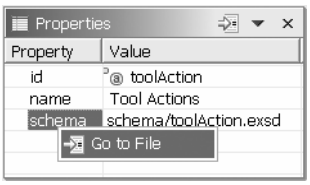


Figure 12-6
Context Menu to display Schema Editor [extpt_06.tif]

Declaring the extension point's code

5. To define our JAR file, which includes the Java code we'll write later on, go back to the Runtime page of the Manifest Editor, select **Add** and enter `extensionpoint.jar`. The export statement to the parent library tag (see bold text below) will make it visible (public) to extenders; this corresponds to the "Export the entire library" choice in the PDE manifest editor:

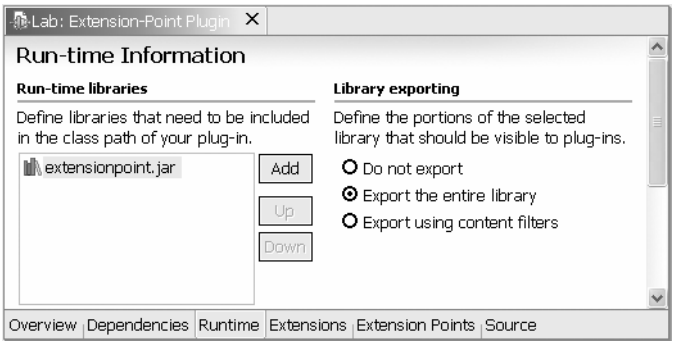


Figure 12-7
Run-time Editor pane [extpt_07.tif]

And here's the corresponding XML that will be generated:

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

Now is a good time to save the plug-in file.

Processing Extension Tags

After declaring the new extension point, we need to process the data tags coming from the extension. When the Workbench is started, all markup tag data is stored in key/value

pairs in the Workbench Plug-in Registry (`Platform.getPluginRegistry()`). So we must access this data and process it.

6. Open the class called `ExtensionProcessor`. Note the following constants:

```
private static final String PLUGIN_ID = "com.ibm.lab.extensionpoint";
private static final String EXTENSION_POINT_ID = "toolAction";
private static final String EXTENSION_TAG = "tool";
private static final String EXTENSION_TAG_ACTION_ATTRIB = "action";
private static final String EXTENSION_TAG_LABEL_ATTRIB = "label";
```

These define the IDs and attributes that will be specified in the plugin.xml.

Note: The reason why we use constants (`private static final`) is that you can easily reuse the `ExtensionProcessor` class as a template in your own code by simply changing the values of the constants.

To start processing of the extensions, update the code that gets the plug-in registry (the run-time composite of the plugin.xml files that were parsed during Workbench startup) and retrieves the `tools` extensions (if any) and save them in instance variables (in bold):

```
private IPluginRegistry pluginRegistry = Platform.getPluginRegistry();
private IExtensionPoint point =
    pluginRegistry.getExtensionPoint(PLUGIN_ID, EXTENSION_POINT_ID);
```

Refer back to the generated HTML documentation of our extension point schema:

```
<!ELEMENT tool>
  <!ATTLIST
     action       CDATA #REQUIRED
     label      CDATA #REQUIRED
   >
```

We have a tag called “tool” and this tag has two attributes: `action` and `label`. The action will hold a fully-qualified name to a class we want to execute when the menu item is chosen. The attribute “label” is just the label of the menu item.

7. Add the following logic to the `load` method that loops through the markup data in the Plug-in Registry, picking up our tags:

```
IExtension[] extensions = point.getExtensions();

for (int i = 0; i < extensions.length; i++) {
```



```
  IExtension currentExtension = extensions[i];

  IConfigurationElement[] tags =
      currentExtension.getConfigurationElements();

  for (int j = 0; j < tags.length; j++) {
    IConfigurationElement currentTag = tags[j];

    // just load the tag we want, all others are ignored
    if (currentTag.getName().equals(EXTENSION_TAG))
      addMenuItem(currentTag);
    }
  }
}
```

You can add more logic to this method, limit the number of tags for each plug-in, exclude plug-ins, add detailed error handling, etc., it's up to you.

Create an Interface that Extenders Must Implement

8. Open the interface, `IToolAction`, that defines the behavior which the extenders must implement. Finish the interface by adding the method definition for `run()`:

```
public interface IToolAction extends IExecutableExtension
{
  public void run();
}
```

The interface `IToolAction` extends `IExecutableExtension` because we need to handle data initialization. We will discuss this in more detail shortly.

You could also add methods to pass variables from the menu pulldown, `ToolActionsPulldownDelegate`, to the classes that handle the actual menu item action (implementors of `IToolAction`). `ToolActionsPulldownDelegate` implements the `IWorkbenchWindowPulldownDelegate` interface, and therefore has access to variables such as `IWorkbenchWindow`, `ISelection` or `IAction`. It depends on the extent of the framework that you are building. In this example, we just need a plain `run()` method.

Design Note: If we wanted to create a more involved example, we would likely extend `IToolAction` with different interface subclasses and default implementations. For example, let's imagine the example extension markup below:

```
<extension point="com.ibm.lab.extensionpoint.toolAction">
```



```
  <tool
    type="external"                          default="internal"
    executable="notepad.exe"                 executable
    cwd="project"                            project|workspace|"other"
    parameters="{selectedResource}"/>        keywords as desired
</extension>
```

In our fictitious example, the "type" dictates the 'action processor.' We have already defined the interface of such a processor for internal commands, namely `IActionTool`. We would define another interface and implementation, `IExternalActionTool` and `ExternalActionTool`, which allows the plug-in developer to define an external command, define its starting directory (`cwd` attribute above), parameters, etc. The default implementation would likely need additional parameters in its `run(…)` method that would be passed to it by `ToolActionsPulldownDelegate`.

We won't implement this particular enhancement, but it does give you a better idea of possibilities that extension points offer.

Finish the ExtensionProcessor Class

9. Complete the `addMenuItem` method of the ExtensionProcessor class. Start by retrieving the tag's attributes inside the commented `try…catch` statement as shown below (in bold):

```
IToolAction toolAction =
    (IToolAction) configElement.createExecutableExtension
    (EXTENSION_TAG_ACTION_ATTRIB);
String label = configElement.getAttribute(EXTENSION_TAG_LABEL_ATTRIB);
```

An interesting point here is the `createExecutableExtension()` method. It looks for the specified tag attribute, the value being a fully-qualified class, and creates an instance of that class. If the class implements `IExecutableExtension`, its `setInitializationData(…)` method is called, passing the `IConfigurationElement` tag that resulted in the new class instance creation. This gives the receiving class the opportunity to process attributes or child tags itself. You can use this approach to simplify and distribute extension tag processing, that is, move the processing out of a central class as we have done in this exercise and into a class specified in an attribute of the tag (`action`, in our case; other Eclipse extension points often call it `class`).

You can add more logic, tags (e.g. better error handling, unknown attributes, etc). For now we will keep it simple and press on.

Connect the ExtensionProcessor class to ToolActionsPulldownDelegate

Now we are able to select the right tags and load the attribute parameters into the `toolActionsParameters` list of the `ExtensionProcessor` class.

10. Go to the `ToolActionsPulldownDelegate` class and add this field:

```
private ExtensionProcessor extensionProcessor =
    new ExtensionProcessor();
```

11. Go to the `init` method and call the `load()` method of the `extensionProcessor` instance (bold text):

```
public void init(IWorkbenchWindow window)
{
  extensionProcessor.load();
}
```

The `init` method is called when the `ToolActionsPulldownDelegate` action delegate is created. This occurs when the user first selects the pulldown menu; prior to that moment, an action proxy handles the initialization and enablement of the menu based on the `<action ...>` tag and its child tags (e.g., `<selection ...>` tag).

12. Take a look at the `getMenu()` method. There we iterate over the `ExtensionProcessor`'s tool actions list to fill in the menu items. Add the following code to the `getMenu()` method to insert a menu choice for each action:

```
for (int i = 0;
    i < extensionProcessor.getToolActionsParameters().size(); i++) {

  ToolActionParameters extensionParms =
    (ToolActionParameters)
      extensionProcessor.getToolActionsParameters().get(i);

  MenuItem menuitem = new MenuItem(menu, SWT.NONE);
  menuitem.setText(extensionParms.getActionLabel());

  // save the action in the menu item reference for handling selection
  menuitem.setData(extensionParms.getAction());

  // set and create selection listener with inner class
  menuitem.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent e) {
```



```
        IToolAction action = (IToolAction) e.widget.getData();
        action.run();
      }
    });
}
```

13. Start the run-time workbench. Go to the Perspective menu and choose Customize. Expand the **Other** menu item and activate our **Toolbar Actions** action set.

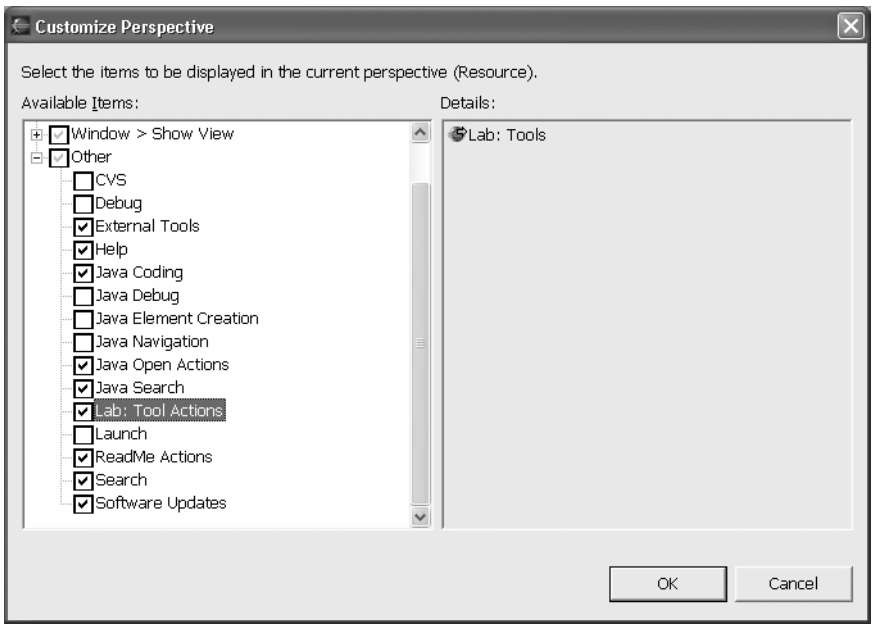


Figure 12-8
Customize Perspective Panel [extpt_08.tif]

14. Find the new Button with a small arrow on the right side. Click on the arrow, no menu items are displayed because no plug-in loaded in the workbench is contributing to our extension point.

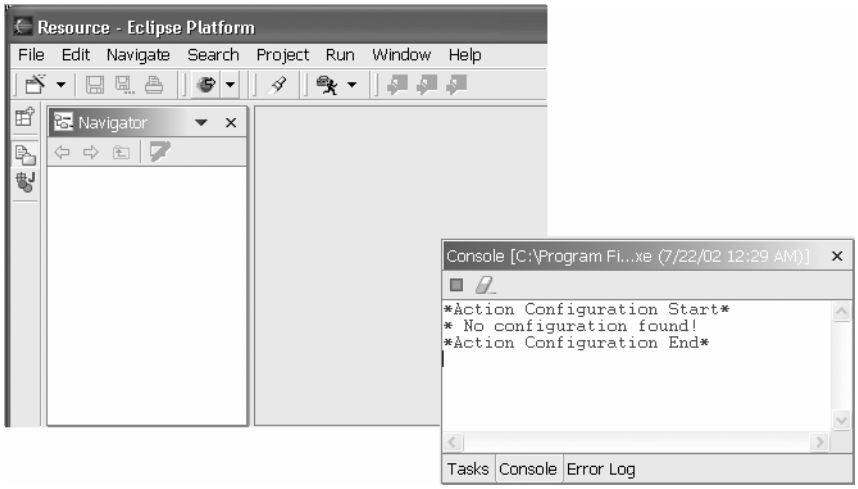


Figure 12-9
Run-time Menu Option and Console Output [extpt_09.tif]

Note: The menu is displayed in the run-time instance and its `System.out` messages appear back in the console of the development Workbench.

15.Close the test instance of the Workbench.

Part 2: Use the Extension Point

Now we want to use our new extension point. This procedure is similar to other exercises. Can you still remember `FirstPlugin` extending actionSets?

Test the new Extension Point

Let's have a short look at the extension tag DTD[3]:

```
<!ELEMENT extension ANY>
<!ATTLIST extension
  point    CDATA #REQUIRED
  id       CDATA #IMPLIED
  name     CDATA #IMPLIED
>
```

According to the above definition, this tag can hold any child tags – it doesn't matter what they are named as long as they are well-formed XML. However, you'll see that by using a extension point schema definition, the PDE manifest editor only permits valid tags and attributes.

The `point` attribute is important because it defines which extension point we are going to use. The key or identifier is built out of the extension point id and the accompanying plug-in id.

So our extension point identifier looks like this:

[3] See the *Platform Plug-in Developer Guide*, under *Reference* > *Plug-in Manifest* for the complete DTD.


```
com.ibm.lab.extensionpoint.toolAction
```

The first part is the plug-in id and the last part is the extension point id, separated by a period.

The `id` attribute of the extension tag is optional and used to make a extension unique. It isn't used very often, but you might find it necessary if your extension point processing code needed to identify a specific tag. If omitted, the workbench provides generic ids. The `name` attribute of the extension tag is optional again and used to give the current extension a user-readable name.

Now to specify our tool action extension point – here's where we see the benefit of an extension point schema.

1. Go back to the plug-in manifest editor, and begin by selecting **Add** from the Extensions page to create an extension. Selecting our schema-based extension, `toolAction`, will guide the rest of our choices.

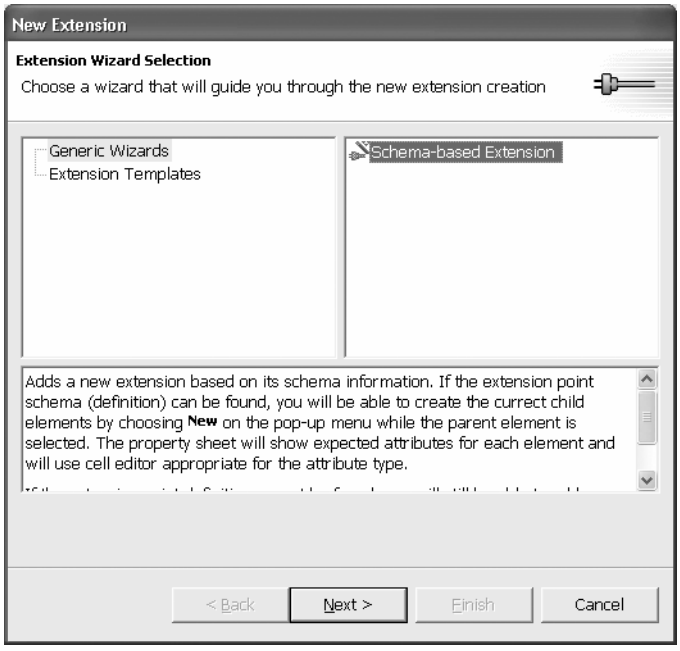


Figure 12-10
New Extension Wizard [extpt_10.tif]

The `toolAction` extension point is shown by its ID, “com.ibm.lab.extensionpoint.toolAction” in the Extension Wizard. The id and name of the extension can be left blank in this case.

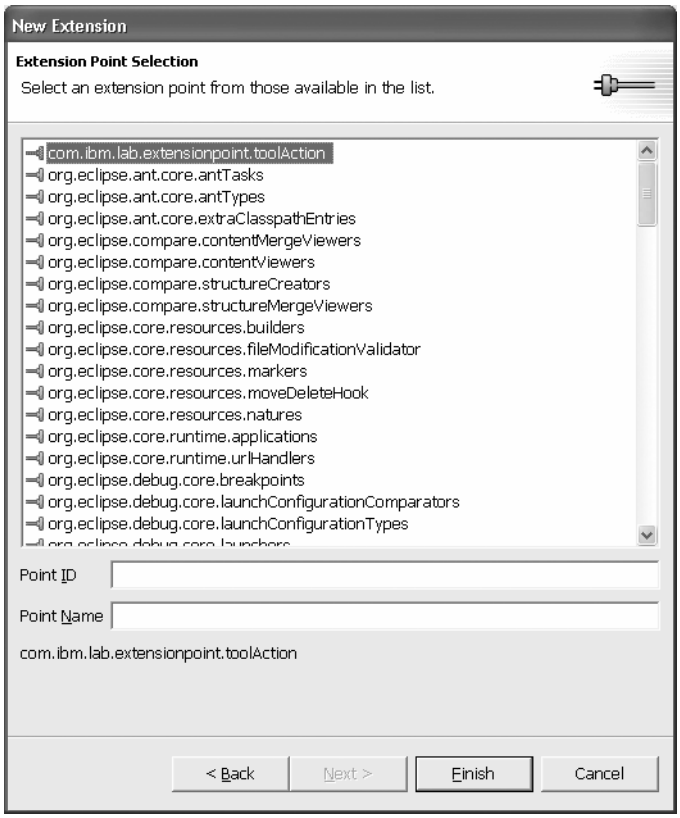


Figure 12-11
Extension Point Selection Pane [extpt_11.tif]

Rather than seeing the generic menu choices when creating subtags, we only see those that apply:

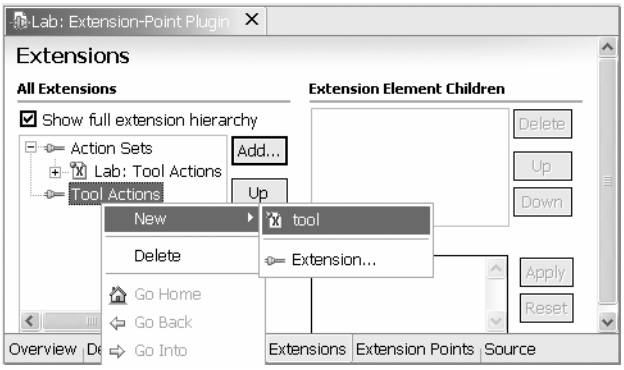


Figure 12-12
Extension Context Menu [extpt_12.tif]

Moreover, the attributes of properties can be selected from prompt dialogs, such as the `action` attribute of the `tool` element, rather then being entered manually. Since in this case we specified an interface rather than a superclass for the `baseOn` value of the `tool`'s `action` attribute, we need to create the target class implementing this interface so it will be accepted:

Figure 12-13
New Java Class Wizard [extpt_13.tif]

Checking "Inherited abstract methods" will generate templates for the `run()` and `setInitializationData(IConfigurationElement, String, Object)` methods.

Now select the newly created class for the `action` attribute of the extension. After entering "My First Command" as the descriptive label, selecting Save, and turning to the Source page, you should see something like this:

```
<extension point="com.ibm.lab.extensionpoint.toolAction">
  <tool
    action="com.ibm.lab.extensionpoint.TestToolAction"
    label="My First Command"/>
```



```
</extension>
```

2. Go back to the PDE and finish the implementation of the class we defined above:

```
public class TestToolAction implements IToolAction
{
  public void run(){}

  public void setInitializationData(
      IConfigurationElement config,
      String propertyName,
      Object data)
      throws org.eclipse.core.runtime.CoreException
  {}
}
```

You should get these two methods above automatically. First we will work with the `run()` method and later on with the `setInitializationData()` method.

3. In the `run()` method, add the following code (bold text):

```
public void run()
{
  MessageDialog.openInformation(null,
      "Run Example",
      "Test successful!");
}
```

Run the Workbench and open our pulldown menu on the toolbar. You should see a menu item called "`My First Command`". When you choose it, the message dialog in the `run()` method is displayed.

4. Close the run-time instance of the Workbench.

Using the extension point from a different plug-in

5. Create a new project, `com.ibm.lab.extensionpoint.test`.

Use the **File>New>Project** wizard, select "Plug-In Development" and then choose "Plug-In Project". Press **Next**.

Enter `com.ibm.lab.extensionpoint.test` for the name and press **Next** and then **Next** again. Check the “Create a plug-in project using a code generation wizard” and choose the “Default Plug-In Structure” and press **Next**.

Enter “Extension Point Test” for the Plug-in name and uncheck “Generate code for the class.” Also remember to blank out the "Class name" field. Press **Finish**.

Remember that the Workbench reads all the plug-in manifests during start-up. It isn’t until the user actually invokes a given plug-in’s functionality that its code is read into memory. Thus the manifest must define the necessary run-time environment, such as prerequisite plug-ins, just as Java source code must list prerequisite packages in its `import` statements.

6. Define the new project’s prerequisites by pressing the **Add…** button and checking the appropriate plug-ins on the Dependencies page of the new class’s plugin.xml file in the PDE manifest editor:

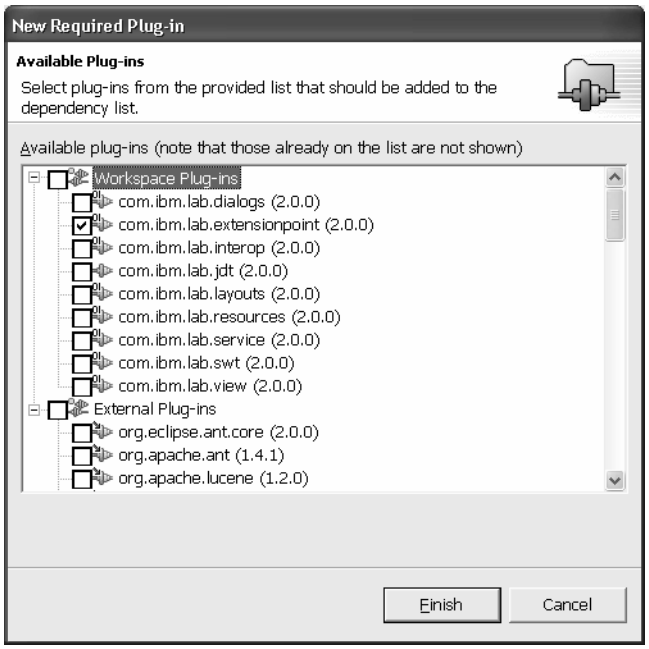


Figure 12-14
Plug-In Dependencies Page [extpt_15.tif]

Adding ”com.ibm.lab.extensionpoint” as a required plug-in to the Dependencies page of the PDE manifest editor will generate the statements below on the Source page:

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



```
  <import plugin="org.eclipse.core.runtime"/>
  <import plugin="com.ibm.lab.extensionpoint"/>
</requires>
```

The corresponding code dependency is documented in the Java source with `import` statements. Modifying and then saving the plugin.xml file will automatically update the project's classpath, assuming the option is set in the PDE preferences:

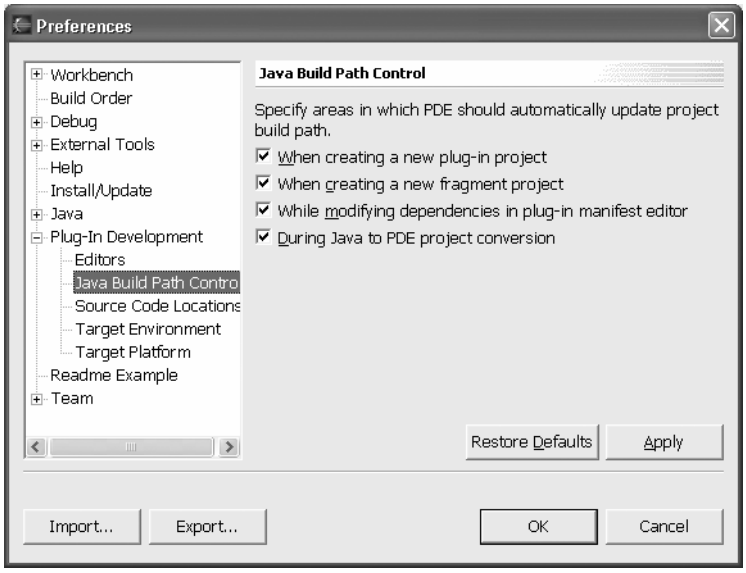


Figure 12-15
Java Build Path Preference Pane [extpt_16.tif]

7. Create a `com.ibm.lab.extensionpoint.test` package and copy the `TestToolAction` class to it. Its `package` statement will automatically be updated; and add the import statement below:

```
import com.ibm.lab.extensionpoint.IToolAction;
```

8. Create an extension with the help of the schema-based generator as before, this time adding two tool actions with labels "`New Command 1`" and "`New Command 2`". In the action field, click the browse button and choose `TestToolAction` from the `com.ibm.lab.extensionpoint.test` group.

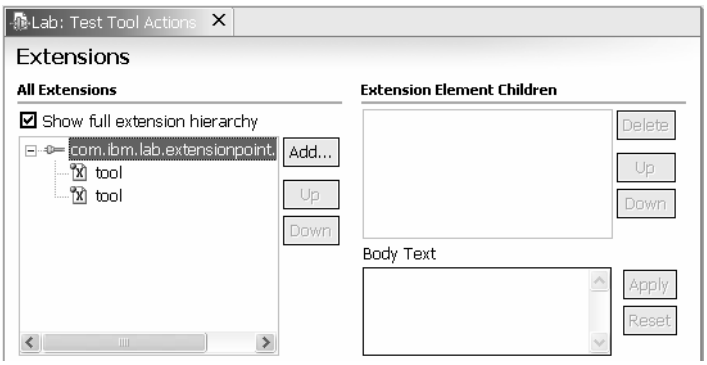


Figure 12-16
Extension Page [extpt_17.tif]

After saving, the Source view should show something like this (bold text):

```
<?xml version="1.0"?>
<plugin
  name        = "Extension-Point Test"
  id          = "com.ibm.lab.extensionpoint.test"
  version     = "1.0.0"
  vendor-name = "IBM">

  <requires>
    <import plugin="org.eclipse.ui"/>
    <import plugin="org.eclipse.core.runtime"/>
    <import plugin="org.eclipse.core.resources"/>
    <import plugin="com.ibm.lab.extensionpoint"/>
  </requires>

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

  <extension point="com.ibm.lab.extensionpoint.toolAction">
    <tool
      action="com.ibm.lab.extensionpoint.test.TestToolAction"
      label="New Command 1"/>
    <tool
      action="com.ibm.lab.extensionpoint.test.TestToolAction"
      label="New Command 2"/>
  </extension>
</plugin>
```

Don't forget to **import** the **`com.ibm.lab.extensionpoint`** package! Otherwise you get an error that your class could not be loaded. Remember from Part 1: We added an `<export name="*">` child tag to the `<library>` tag to make the pulldown menu package with our extension point visible [public] for extenders. This mechanism, to export first and to import later could be called a "handshake". If one of these parts is missing, it will not work.

Note: To keep it simple we are just using the same Java class for both menu commands.

We are not finished yet!

9. Modify the `TestToolAction` class in the `com.ibm.lab.extensionpoint.test` package. Add the following fields (bold text):

```
public class TestToolAction implements IToolAction
{
  private Object stringData;
  private IConfigurationElement tag;
}
```

10. Go to the `setInitializationData` method and add the following (see bold text):

```
public void setInitializationData(
  org.eclipse.core.runtime.IConfigurationElement config,
  java.lang.String propertyName,
  java.lang.Object data)
  throws org.eclipse.core.runtime.CoreException
{
  stringData = data;
  tag = config;
}
```

Now we can receive data from this method and are able to store it so we can reference it later. Let's look a little more closely into how `setInitializationData(…)` is called.

When the `IConfigurationElement.createExecutableExtension()` is called from `ExtensionProcessor's addMenuItem(IConfigurationElement)` method, the Eclipse Platform is looking for the `setInitializationData()` method in the target class. If the `setInitializationData()` method is defined, it is called using the following objects as initializing data: the `IConfigurationElement config`, the `Object data` and the `String propertyName`. The `config` parameter is the current tag in the XML data, in our case the tag `tool`. The `data` parameter is actually a String (more about it in a moment). The `propertyName` parameter is the name of the attribute that is passed to the `createExecutableExtension()` method, in our case "action".

First let's start with the `Object stringData`.

11.Go to the `run()` method and change the content to the following (bold text):

```
public void run()
{
  MessageDialog.openInformation(null,
    "Run Example",
    "Parameters hard coded into the xml, " +
    "on the class attribute and after the colon: " +
    stringData);
}
```

You'll see how this format is handled by the `createExecutableExtension` method. Of course, we could have created another subtag to pass this parameter; we chose `createExecutableExtension's` ":" technique since it was only a simple string.

12.Go to the plugin.xml and make the following changes (bold text, should be one line):

```
<extension point="com.ibm.lab.extensionpoint.toolAction">
  <tool
    action="com.ibm.lab.extensionpoint.test.TestToolAction:Greetings
from New Command 1"
    label="New Command 1"/>
  <tool
    action="com.ibm.lab.textensionpoint.test.TestToolAction:Greetings
from New Cammand 2"
    label="New Command 2"/>
</extension>
```

The text after the semicolon in the class attribute is our `Object stringData`. The value of the attribute, which is made executable (in our case, the attribute `action`), is divided into two sub-strings: the first part up to the semicolon is seen as a full qualified class name and the second from the semicolon on is an Object that simply holds a String. If you leave off the semicolon and the rest, the `Object stringData` is null.

In this exercise, we used this Object to customize a dialog message so that we can reuse one class (`TestToolAction`) for any number of menu commands.

13.Save the `plugin.xml` and start the Workbench.

Now the menu items appear because there is a plug-in in place to extend the tool action extension point.

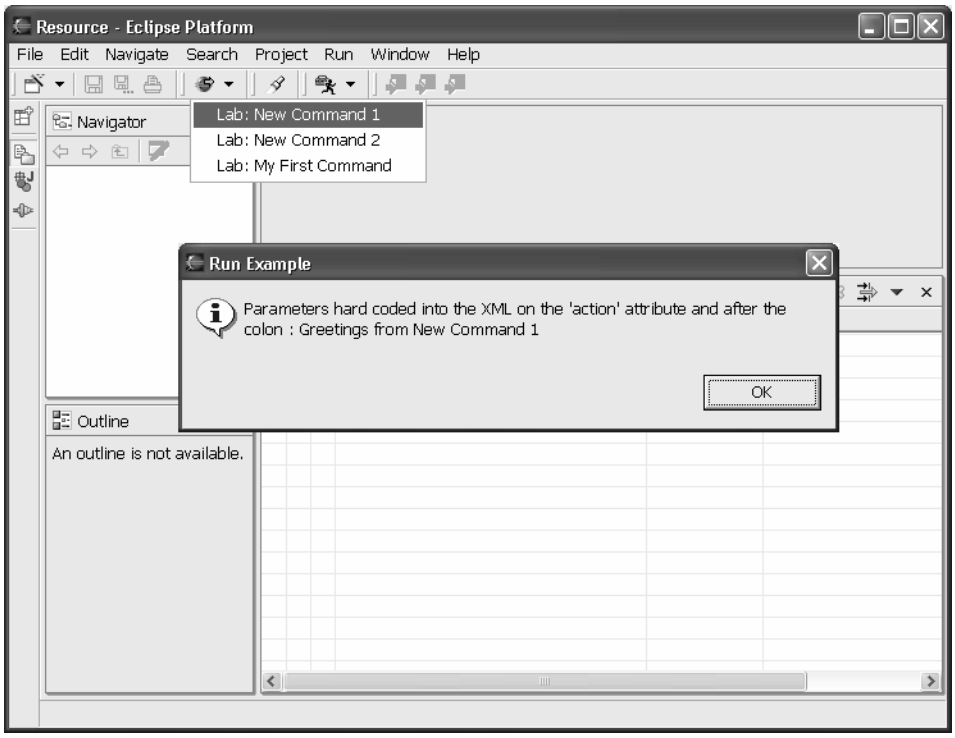


Figure 12-17
Final Result [extpt_18.tif]

Exercise Activity Review

In this exercise, you became familiar with extension point creation and usage.

Exercise 13 Feature Development and Deployment

This exercise takes you through the process of developing a feature and then deploying the feature using three different approaches for installation. This exercise is designed to help you understand how features are created and how you can use the PDE to assist in the packaging steps required for plug-ins and features. Once packaged, you then use all the available techniques that support delivery of your features as a product, or as part of a product, that is based on the Eclipse platform.

Introduction

Plug-ins are the fundamental building blocks that a tool developer creates to add new capabilities or augment the existing capabilities of the Eclipse platform. However, the Eclipse end user's point of view is different: they are interested in using and adding to the capabilities of an Eclipse platform-based product. Eclipse features represent this viewpoint by offering the end user a means to add additional tool capabilities, without having to be concerned with the underlying plug-ins and associated artifacts that make up its implementation.

Exercise Concepts

The Update Manager provides the user built-in support for installing and managing features. It falls on you, the plug-in developer, to define features that organize your plug-ins and its dependent features so users can integrate them as part of an Eclipse-based product. The feature definition also allows you to supply branding for the function you want to add.

A user of an existing product can then add your features, either configured as an extension using your install routine or installed as part of an existing product using the Update Manager. Once the features are installed, they use Update Manager to manage the features.

The final scenario covered in this exercise is for tool builders who want to bundle their features with Eclipse to create their own branded product. To do this, you identify one of your features, which supplies the product and feature branding, as the primary feature.

Skill Development Goals

This exercise looks at the definition and use of features from three viewpoints.

- Section 1 is from the viewpoint of plug-in developers who want to deliver their plug-ins by means of features—that is, the role of a *feature creator*. In this section you'll learn how to define and package features and plug-ins so that they can be used to add function to Eclipse through an install process.
- Section 2 is from the viewpoint of end users, and focuses on their role as a *feature consumer*, as they use an existing Eclipse installation to install and configure features from an update site.
- Section 3 is from the viewpoint of the *product developer*, who will bundle one or more features and their associated plug-ins, as part of a branded Eclipse-based product.

The steps defined in each section are continuous. That is, they must be done in sequence; you cannot jump straight to Section 2 or 3.

The role of the individual who performs each activity is different; the sequence followed is more part of the exercise structure than the expected steps for any one individual. A feature developer defines and packages features; a tool user installs features as an extension or using Update Manager, and a product developer builds the configuration of Eclipse and the packaged features as a customized product.

Exercise Setup

Setup tasks must be complete before you begin the exercise. These exercise setup tasks are required before you can begin this exercise:

1. Install a second copy of Eclipse.

   This task is actually optional. You may wish to work with an alternate install if you are not comfortable with manipulating the Eclipse installation you are using for the exercises during the feature installation section of the install exercise.

   If you do not use a second copy of the workbench, you will need to reverse the final product configuration steps in Part 3, as these steps will change the configuration of your original Workbench installation.

2. Import the `com.ibm.lab.tool.pkg` plug-in project.

   If not done when preparing for other exercises, import the `com.ibm.lab.tool.pkg` plug-in project from the `\PluginLabTemplatesCore` directory as described in Appendix A, "Installing Exercise Templates". This project provides feature and product branding content, as well as other files you will copy to your feature project and the file system during the exercise.

3. Choose the plug-ins for which you want to create features.

   You may choose plug-ins from the exercise solutions, your previously completed exercises, or your own plug-ins that you wish to deliver as features for this install/update packaging exercise.

Exercise Instructions

Note: Each part must be performed in sequence. The definitions in Part 1 are used in Part 2; the result of Part 2 sets up the system for Part 3.

Part 1: Tasks of an Eclipse Feature Developer

In Part 1 you will perform the tasks required to prepare a plug-in, and define a feature, to support their installation as part of an Eclipse configuration. This includes the following activities:

- Creating runtime components for the plug-ins that will be contained in the feature
- Organizing the function provided by your plug-ins as a feature
- Defining product and feature branding content for your feature
- Creating feature and plug-in archives that will be used by the Update/Manager to install or service features.
- Implementing an update site that Update Manager can browse for features to install or use to service features in the current configuration.

Step 1. Generate JAR files for selected plug-ins

1. Identify at least one, but no more than three plug-ins that you will package as a feature during this exercise. You could choose to package them all, but selecting more will increase the time it takes to perform the plug-in setup and Ant build tasks.

   They can be from the set of plug-in solutions provided to you or the plug-ins you have developed following other exercises (or on your own). Suggested plug-in projects:

   ```
   com.ibm.lab.soln.dialogs
   com.ibm.lab.soln.view
   com.ibm.lab.soln.firstplugin
   ```

2. Create a `build.properties` file if one does not already exist.

   The PDE can generate the runtime JAR file. This process requires that you have a properly configured build.properties file for the plug-ins you want to prepare for runtime.

   If the build.properties file does not exist, you can cause one to be created by editing the plugin.xml file. Add the appropriate source folder (src-pluginname) folder to the runtime

specification page as shown in Figure 13-1 and the PDE will create a build.properties file.

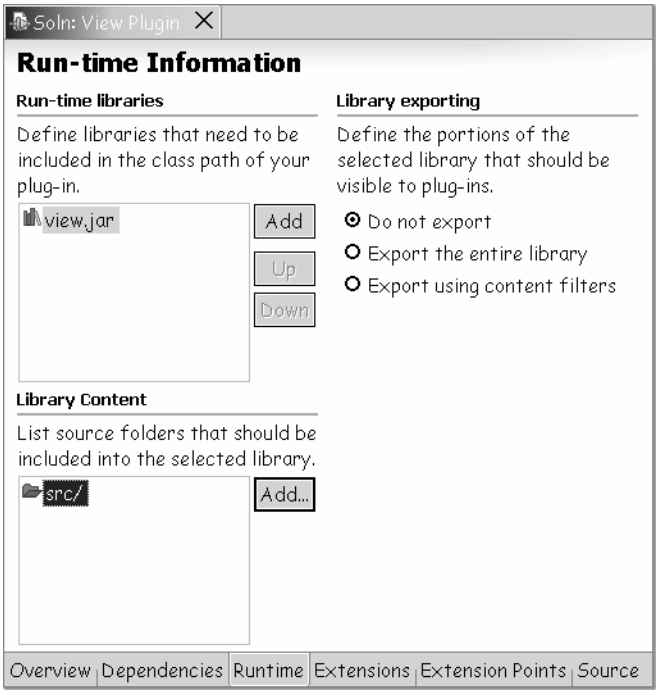


Figure 13-1
Updating plug-in JAR Definition to Generate build.properties File

Note: The source folders for projects you have imported will have names such as src-view/ because of how the source was created from an zip file in the imported plug-in.

You can even just edit the `plugin.xml`; any changes made on the Runtime page will trigger the creation of a partially configured `build.properties` file. The generated `build.properties` file has one entry; you will add others as you proceed through the exercise.

3. Customize the `build.properties` file to support plug-in install JAR creation processing.

   The PDE can generate the install JAR file to support runtime deployment. This build process requires that you have a properly configured `build.properties` file for each plug-in you want to prepare for a runtime deployment. First we will review the process and requirements for creating a properly configured `build.properties` file and then you will customize one for each plug-in that you have chosen to include in your feature.

   The `build.properties` file is associated with a specialized editor, this is shown in Figure 13-2.

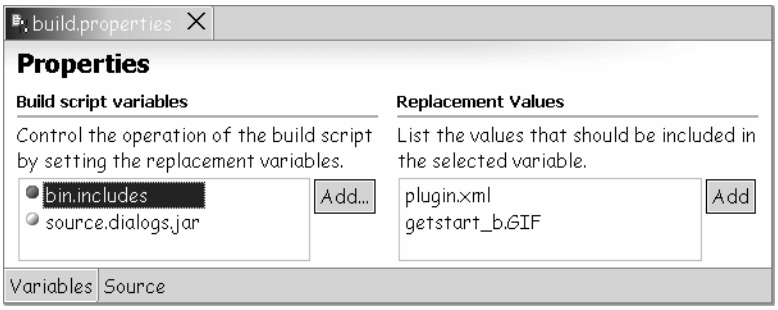


Figure 13-2
build.properties editor

In Figure 13-2 you see two variables:

- source.identifier.jar – where identifier.jar is the name of the runtime jar file identified in the plug-in specification. The variable value identifies the source tree that determines the contents of the JAR file.
- `bin.includes` – identifies the files that need to be included when preparing a plug-in install JAR or the ZIP file that supports distribution of the feature and contained plug-ins. A `bin.includes` is also used in a Feature project to define the files that should be included in the feature install JAR file or distribution ZIP file.

To complete this task, for each plug-in that you have chosen to include in your feature, ensure that the associated `build.properties` file:

- Includes an appropriate source.identifier.jar entry and value to correctly identify the source directory for the runtime JAR file you want to build
- Includes a `bin.includes` entry that lists all the files and directories that should be included in the runtime distribution for the plug-in (you do not need to identify the runtime JAR file in this list).

Files such as the `plugin.xml`, image files, or other files used in the plug-in must be specified in the `bin.includes` variable of the `build.properties` file if you want them to be available at runtime.

Note: You have to add the `plugin.xml` file to the `bin.includes` entry. This mandatory file is not automatically added by the PDE when creating the `build.properties` file.

Turning to the Source page of the `build.properties` editor, you should see something like this (the example below is for the `com.ibm.lab.dialog` plug-in):

```
bin.includes = plugin.xml,\
               getstart_b.GIF
source.dialogs.jar = src/
```

The entry above will include any Java packages in the `src` folder in a JAR named `dialogs.jar`; the files `plugin.xml` and `getstart_b.GIF` will be included in the runtime distribution (\ is a continuation character).

About now, you might recognize the wisdom of placing all image files in a directory such as `icons`. This allows you to simply add `icons/` to the `bin.includes` variable in the `build.properties` file in order to include all your images. That is, identifying a directory indicates that all files and subdirectories contained within the specified directory should be included in the runtime distribution.

Notes:

- If you modify or add values to the `build.properties` file, be sure to regenerate the build.xml file. This is done by selecting the `plugin.xml` file and choosing the context menu option **Create Plug-in JARs**. This choice regenerates the `build.xml` files and then opens a dialog that will allow you to run Ant using the build.xml for the selected plug-in. This is what you will do in the next task.
- If you select the `build.xml` file and select the **Run Ant...** menu choice, any changes made to the `build.properties` file will not be reflected in the `build.xml` file. That is, the Ant processing will either fail or omit required files from the runtime or install JAR files.

You will use these options to create the `build.xml` and build the plug-in runtime JAR file in the next step.

4. Generate the runtime JAR files for a plug-in using the PDE Create Plug-in JARs option.

   For one plug-in project, select the `plugin.xml` file and choose the context menu option **Create Ant Build File**. This creates a `build.xml` file based on the current content of the `build.properites` file.

5. Select the `build.xml` file and choose the **Run Ant...** context menu option to start the Ant launcher. Only the default target, **build.jars**, should be selected on the **Targets** page. Select **Run** to start the Ant processing.

   The Console view displays a log of the processing performed to create the runtime JAR file (see Figure 13-3).

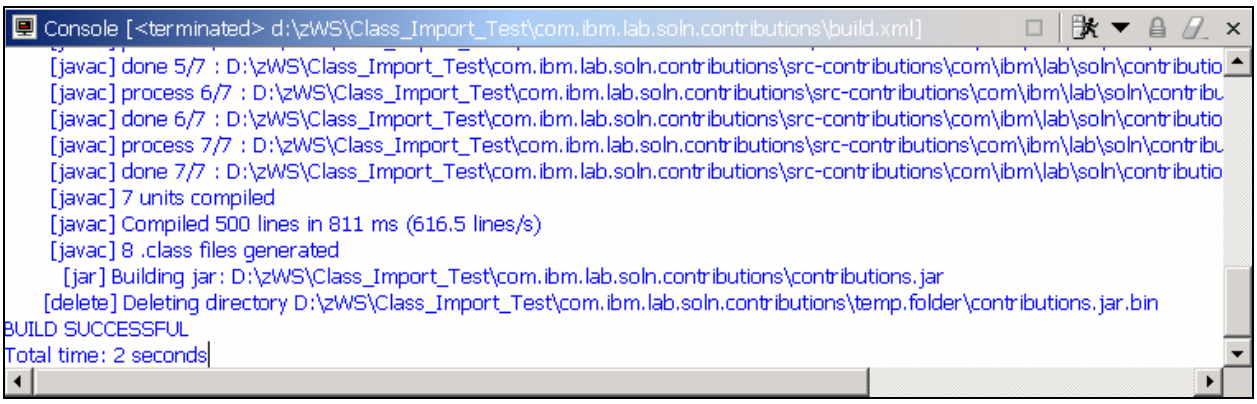


Figure 13-3
Runtime plug-in build log

6. Select the project and choose the **Refresh** context menu option. This will make sure you can see all the files created by the JAR creation process. Review the files added to the project.

Note: If you are using CVS this would be a good time to add a `*.jar` entry to the `.cvsignore` file in plug-in and feature projects to prevent sending the runtime and install JAR files to the repository.

You can add the folder to the `.cvsignore` file by selecting the temp.folder, and using the **Team > Add to .cvsignore** context menu option. If you decide not to send any JAR files to CVS, use the same process and then select the Custom pattern option, enter *.jar, and select **OK**.

Only run the **Create Ant Build File** and **Run Ant…** process and for one plug-in. We did this so that you would know how this works in case you need to generate a runtime JAR for a single plug-in. The runtime JARs for the other plug-ins will be automatically generated when the feature that includes the plug-ins is processed. This is part of the next step.

Step 2. Package the function provided by your plug-ins as a feature

1. Create a feature project using the PDE wizard.

   Create a feature project using File > New > Project; select the Plug-in Development category and Feature Project wizard. Enter the following as attributes on the wizard pages:

   - Project Name: com.ibm.lab.tool.pkg-feature
   - Feature Id: com.ibm.lab.tool.pkg
   - Feature Name: Demonstration Tools Package
   - Feature Version: 2.0.0
   - Feature Provider: IBM

2. Select the plug-ins that you have chosen to include in your tool feature package.

   Shown below is an example of the feature.xml source that is created after you select the desired plug-ins:

```
<?xml version="1.0" encoding="UTF-8"?>
<feature
      id="com.ibm.lab.tool.pkg"
      label="Demonstration Tools Package"
      version="2.0.0"
      provider-name="IBM">

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

   <plugin
         id="com.ibm.lab.soln.dialogs"
         download-size="0" install-size="0"
         version="2.0.0">
   </plugin>

   <plugin
         id="com.ibm.lab.soln.firstplugin"
         download-size="0" install-size="0"
```



```
      version="1.0.0">
   </plugin>

   <plugin
      id="com.ibm.lab.soln.resources"
      download-size="0" install-size="0"
      version="2.0.0">
   </plugin>

   <plugin
      id="com.ibm.lab.soln.view"
      download-size="0" install-size="0"
      version="2.0.0">
   </plugin>

</feature>
```

Note: It is very important that the version reference in a `feature.xml` file match the version defined in the target plug-in's `plugin.xml` file. If these do not match, the feature manifest editor will identify that there is a mismatch on the Content page; errors will also occur when you attempt to package the feature install JAR.

If you select a plug-in entry on the Content page of the feature manifest editor, you can use the **Synchronize Versions…** context menu option to open the Feature Versions dialog (see Figure 13-4). The Feature Versions dialog helps coordinate the version defined for the feature with the versions identified in the feature for the contained plug-ins and the version defined in the manifest file for each included plug-ins. The Feature Versions dialog options allow you to:

- Set the plug-ins' version number(s) to the version specified for the feature.
- Copy the version numbers from the plug-in manifest(s) in the workspace to the matching plug-in entries in the `feature.xml` file
- Update the version numbers in the plug-in manifest(s) files in the workspace with the version values from the matching plug-in entries in the `feature.xml`

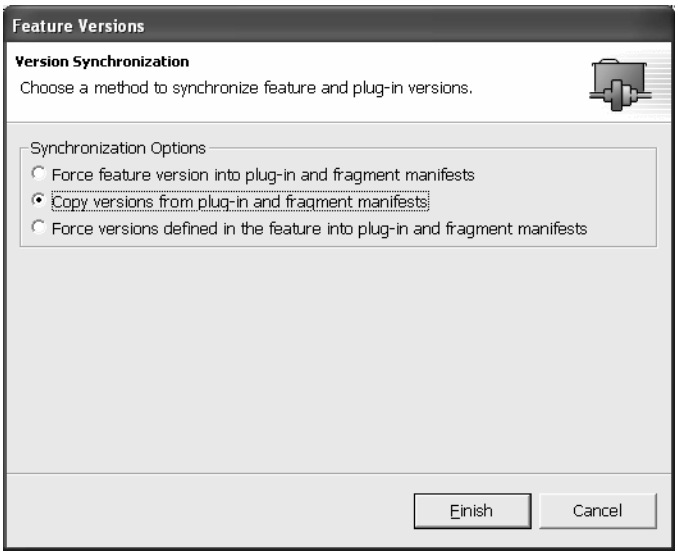


Figure 13-4
Coordination of version ids between feature and plug-ins

3. Add license text to the feature definition. You must enter some text.

   The license text is added using the Information page of the `feature.xml` manifest editor. Choose the License Agreement entry from the section pull down and enter the text you would like shown on the Feature License page of the feature install wizard.

   The user must accept the feature license displayed before they will be able to install the package using Update Manager.

   When the HTML version of the license is identified in the `feature.xml` it can be opened from the About *product* Features dialog. (Help > About *product* > Feature Details > More Info).

   Note: If you do not have any text in your license, the install button is not visible. A license is required.

4. New in 2.1, the PDE automatically creates a `build.properties` file for the feature. The `bin.includes` entry identifies the files that should be included in the runtime distribution for the feature. The PDE created `build.properties` file contains this entry:

   ```
   bin.includes = feature.xml
   ```

   If your feature includes a `feature.properties` file, an HTML version of the license, or a banner graphic, these should also be included in the `bin.includes` entry.

   Note: A `feature.properties` file can be used to provide translatable strings that are substituted for values in the `feature.xml` file. In this exercise, we have kept the feature definition simple, but you may wish to add additional files and definitions to suit your needs.

5. To package the feature:

   - Select the feature.xml file and choose the context menu option **Create Ant Build File**
   - Select the `build.xml` file and choose the **Run Ant...** context menu option.

   The **Create Ant Build File** will regenerate the `build.xml` based on the current `feature.xml` and `build.properties` definition.

   Note: If you just invoke the `build.xml`'s **Run Ant...** option, any changes to the `build.xml` file that are required because of `feature.xml` or `build.properties` file updates will not be applied to the `build.xml` file before it is processed. The other feature packaging options will generate a new `build.xml` file to reflect any changes in the content of the `build.properties` or `feature.xml` files. The

   The **Run Ant...** option will open the Ant launcher. Only the `build.update.jar` target should be selected in the Targets page. If not, adjust the selections until only the `build.update.jar` target is selected, and select **Run**. The **Create Ant Build File** option will also create the build.xml files required for the plug-ins referenced by the feature.

   You can deselect/select the available targets in the Ant launcher dialog to select which targets will be processed and control the order. The order that they will be invoked is

displayed as part of the dialog. The targets chosen will first run their dependent targets. You may want to experiment with alternative target selection later. For the duration of this exercise, to ensure you get to expected results, stick with the target selection guidance provided.

Selecting **Run** to invoke Ant for only the `build.update.jar` target will:

- Process the build.xml for each plug-in referenced in the feature to:
- Generate the runtime JAR(s) for the plug-in
- Generate the plug-in install JAR
- Generate the feature install JAR for the feature project.

Note: You may need to refresh the plug-in and feature projects before you can see the JAR files that are created.

These JAR files would be created for the `com.ibm.lab.soln.dialogs` plug-in project if it was included in your feature:

- dialog.jar
- com.ibm.lab.soln.dialogs_2.0.0.jar

The first file, `dialog.jar`, is the plug-in runtime JAR file. If the feature is composed of more than one plug-in, runtime JAR files are created for each plug-in included in the feature. The second file, `com.ibm.lab.soln.dialogs_2.0.0.jar`, is the plug-in install JAR file.

The file `com.ibm.lab.tool.pkg_2.0.0.jar` is created in the `com.ibm.lab.tool.pkg-feature` project. This is the feature install JAR.

The plug-in install JAR and feature install JAR file names are based on the respective plug-in or feature id, not the project name. During an install, the Update Manager will read the `feature.xml` file from the feature install JAR, and using the plug-in id references, find the plug-in install JAR files that should be processed when an installation is requested.

You may choose other target options in the Ant launcher dialog, but be sure you understand the overall processing flow before you add to the list of targets to be processed. The different target options in the `build.xml` file sometimes include `depends="…"` definitions along with directed activity. The `depends="…"` definitions are targets that are performed first before the directed activity, which represents tasks or other targets that are then invoked. You do not need to select targets that will be run anyway; you may want to review the `build.xml` source to become familiar with the processing for each target.

The `build.update.jar` target triggers processing which includes:

- init processing – a dependency for the build.update.jar target
- `all.children` – a directed target that runs the `build.xml` script for each plug-in in the feature
- `gather.bin.parts` – a directed target customized from the `bin.includes` entry in the `build.properties` file. This target processing gathers all the files required in the feature install JAR into a temporary directory.

- Ant `jarfile` processing to create the feature install JAR file from the contents of the temporary directory

This processing flow is visible (key sections are in bold) when you review the Ant `build.update.jar` target definition from the feature `build.xml` file:

```
  <target name="build.update.jar" depends="init" description="Build the
feature jar of: com.ibm.lab.tool.pkg for an update site.">
    <antcall target="all.children">
      <param name="target" value="build.update.jar"/>
    </antcall>
    <property name="feature.base" value="${feature.temp.folder}"/>
    <delete dir="${feature.temp.folder}"/>
    <mkdir dir="${feature.temp.folder}"/>
    <antcall target="gather.bin.parts" inheritAll="false">
      <param name="feature.base" value="${feature.temp.folder}"/>
    </antcall>
    <jar jarfile="${feature.destination}/${feature.full.name}.jar"
basedir="${feature.temp.folder}/features/${feature.full.name}"/>
    <delete dir="${feature.temp.folder}"/>
  </target>
```

6. With the `build.xml` file selected, start the Ant launcher dialog again, and select the `refresh` target as the second target to be processed. Select **Run** to start the Ant processing. This will refresh the plug-in and feature projects so that you can see the JAR files created during the Ant build.

   Remember, if you change the feature definition, you would need to regenerate the `build.xml` file.

Step 3. Add product and feature branding to the feature

Features support the definition of brand information, in the form of graphics, descriptive text, and license content. This allows the feature to uniquely identify itself in the workbench through entries in the About *product* Features dialog and Install/Update perspective.

One feature is identified as the primary feature, which means it contributes product branding. The primary feature provides the startup splash image, Workbench title, and the content found in the Help > About *product* dialog.

By default, almost all of the branding content is kept in a plug-in that has the same id as the feature; the feature only provides the image and license information that can be seen in the Install/Update perspective. The feature license can also be seen when you select the More Info button (Help > About *product* > Feature Details > More Info). In this step, you will add a plug-in that will supply branding content for the feature you just defined.

1. Create a plug-in with the same id as your feature

   A plug-in with the required branding content has been provided for your use. If required, import the plug-in `com.ibm.lab.tool.pkg` into your workspace from the exercise template location.

   @since 2.1 – You can now identify the plug-in that will supply branding content as part of the `feature.xml` definition. The attribute `plugin="com.qrs.brand.plugin"` can be defined as part of the `<feature>` element in the `feature.xml` file.

2. Add a banner image to the feature. Move the `FeatureUpdate.jpg` file from the `icons` directory of the `com.ibm.lab.tool.pkg` project to the feature project. The `feature.xml` definition can identify a feature graphic as an attribute of the `<feature>` tag.
3. Update the `feature.xml` file to identify the banner image (`image="FeatureUpdate.jpg"`). Enter `FeatureUpdate.jpg` in the banner image field on the Overview page of the feature.xml manifest editor.

   The Install/Update perspective displays the feature graphic when the feature is selected.
4. Add the `com.ibm.lab.tool.pkg` plug-in to the `feature.xml` definition.

   Use the Feature plug-ins and fragments Add… button on the Content page of the `feature.xml` manifest editor to select the `com.ibm.lab.tool.pkg` plug-in. Save the `feature.xml` file when done.
5. Add the `FeatureUpdate.jpg` file to the `bin.includes` list in the `build.properties` file for the feature project. This will ensure that packaging includes the image as part of the feature install JAR and ZIP distribution. Save the `build.properties` file.

Remember, these updates require that you recreate the build.xml file for the feature.

Step 4. Repackage the Feature

Multiple techniques are available when you want to prepare a runtime distribution. You can create either a:

- Distribution ZIP file with a directory structure for the feature, its plug-ins, and their files.
- Feature install JAR for the feature and a plug-in install JAR for each plug-in included in the feature. These JARs are added to an update site and can then be used to install function using the Update Manager.

In this exercise step, you will create both distribution formats.

1. Package the feature by either selecting the `feature.xml` file and choosing the **Create Ant Build File** context menu option
2. Select the `build.xml` file and choose the **Run Ant…** context menu option to open the Ant launcher dialog.
3. In the Ant launcher dialog, make sure you have selected the three targets with the processing sequence shown in Figure 13-5.

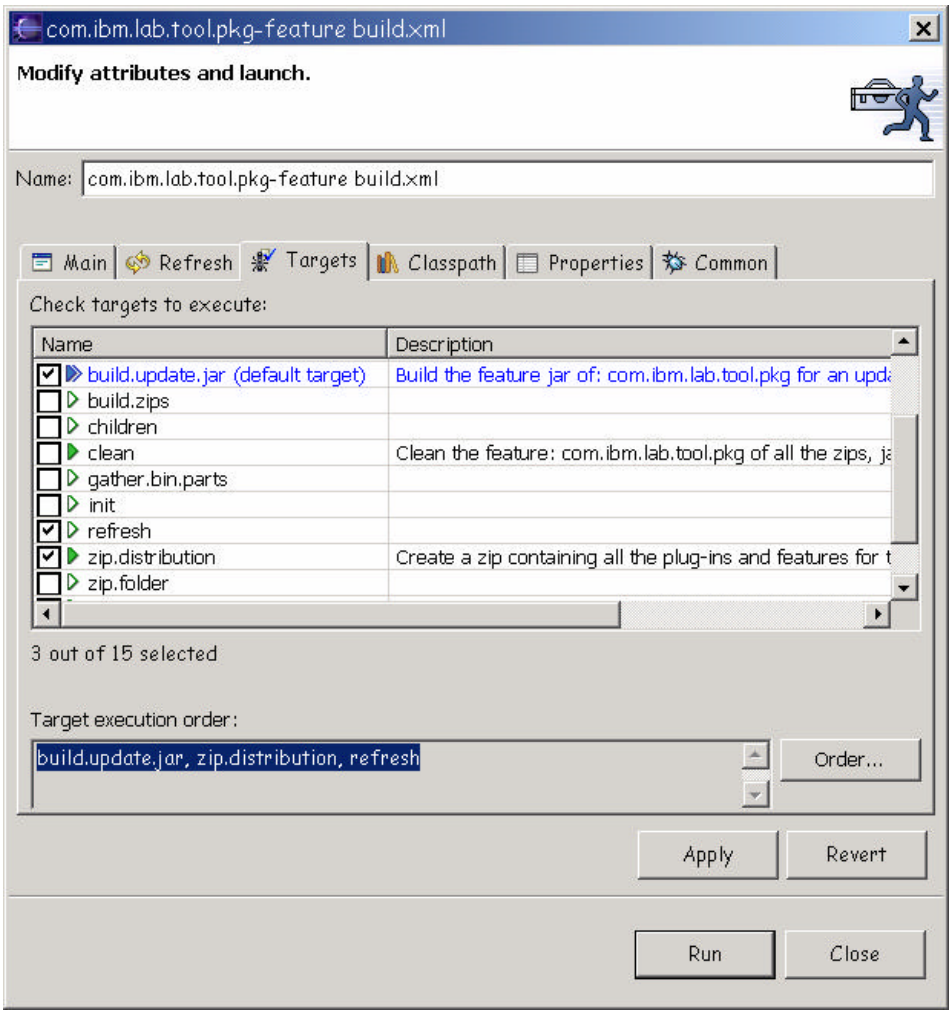


Figure 13-5
Target Selection for Feature Packaging

To get the target invocation order correct you can deselect/select the targets or use the **Order…** button.

4. Selecting **Run** will generate runtime JAR files and then install JAR files for the plug-ins contained in the feature and then a feature install JAR file for the feature itself. A distribution ZIP file of the feature and plug-ins will also be created. The content of both the packaged JAR files and the zip file and is determined by the `bin.includes` setting in the appropriate `build.properties` file for each plug-in and feature project.

   The distribution ZIP file can be used to support installing the feature by either unzipping the contents over an existing Eclipse platform install or unzipping the contents to a new location and adding the function to an existing Eclipse platform install as an extension. You will use the packaged JAR files to implement an update site in Step 5. Step 6 and 7 will cover the extension and update site installation approaches.

@since 2.1 – The feature manifest editor includes support for the direct export of the update JARs, or a distribution zip file, for the feature and it referenced plug-ins. Select **Export…** on the Overview page of the feature manifest editor to use this function.

You should now have a set of projects with a runtime JAR and plug-in install JAR for each plug-in and a feature install JAR and distribution ZIP file for the feature project. This will look similar (depending on your choice of plug-ins) to the files shown in the Package Explorer view in Figure 13-6. Make sure you refresh the feature project so that the JAR and ZIP files are visible.

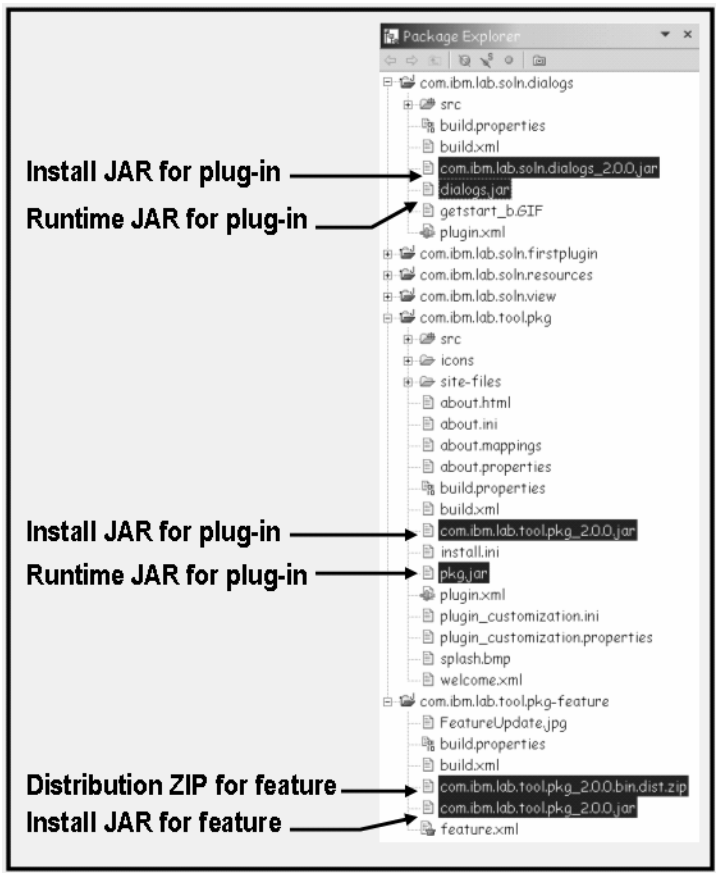


Figure 13-6
Project content after packaging feature

Step 5. Extract installable feature from Workspace and implement an update site
The feature has been built and packaged. The next step is to integrate these files as part of an update site, which can be used to modify an existing Workbench configuration. This requires that the feature install JAR and plug-in install JARs be copied from the development workspace to the local file system along with a `site.xml` file which defines the structure and content of the update site.

1. Create target directory structure.

   Create a directory tree (for example, `d:\labsite\features` and `d:\labsite\plugins`) to use as a site location target.

2. Copy files from the workspace to the appropriate location in the site directory tree (`d:\labsite`). The site files, feature install JAR, and plug-in install JARs need to be copied to the site directory tree. You can drag and drop the files directly from the Navigator to the Windows Explorer.

   The site files located in the site-files directory of the branding plug-in provided to you (com.ibm.lab.tool.pkg) need to be copied to the site directory tree:

   `d:\labsite\site.xml`

   `d:\labsite\siteinfo.html`

   @since 2.1 – The PDE now supports the development of an install site, with its associated `site.xml` file, as a new type of project. Use the New wizard to create an Update Site Project if you want to learn more about `site.xml` development.

   The feature install JAR file located in the feature project needs to be copied to the site directory tree:

   `d:\labsite\features\com.ibm.lab.tool.pkg_2.0.0.jar`

   The plug-in install JARs located in each plug-in project need to be copied to the site directory tree:

   `d:\labsite\plugins\com.ibm.lab.tool.pkg_2.0.0.jar`

   `d:\labsite\plugins\com.ibm.lab.soln.dialogs_2.0.0.jar`

   … and so on, for as many plug-ins as you included in the feature, which includes the branding plug-in that was provided to you.

Part 2: Tasks of an Eclipse User

In Part 2, you will play the role of a user of the Workbench and add function to the current Workbench installation. This includes:

- Installing a new feature as an extension to an existing Workbench installation
- Adding a feature to an existing Workbench configuration using Update Manager

The result of these two techniques is the same with respect to a given workspace; the feature is added to the current configuration. What can differ is whether the feature will be accessible when you open a new workspace.

The answer is yes when extending a Workbench installation, but if a default configuration exists, you must accept the features in the extension as part of the configuration when opening a new workspace.

When using an Update Site to add features, the features will not be accessible when opening a new workspace if the features are added to a new install location. The location of the new install location is only recorded in the current workspace; another workspace would have no visibility. To have the new features accessible when opening a new workspace you must add the features to the same directory tree as the current installation of Eclipse. If a default configuration exists, you must accept the features in the extension as part of the configuration when opening a new workspace.

Step 6. Install a new feature as an extension to an existing product

Using the zip file created in Part 1, you can integrate its contents with an existing Workbench installation. Unzipping the feature and plug-in content emulates how you, as tool provider, would use a product installation routine to allow others to install and use your tool.

As a tool provider, you may decide to use installer technology, such as InstallShield, to extend an existing Eclipse platform installation. Your installer would add your features and plug-ins to the file system and then add a link file to identify the new install site to the existing Eclipse platform-based product you want to extend. The new site is processed during the next startup of the Eclipse platform, which adds your features and plug-ins to the existing Workbench-based product.

1. Unzip the feature and plug-ins distribution file to the file system.

   First create a directory tree (for example, `d:\labpkg\eclipse`) to use as a target and then unzip the `com.ibm.lab.tool.pkg_2.0.0.bin.dist.zip` file into this directory.

   The target of a link file is a directory that contains an `eclipse` directory; the zip file only contains `features` and `plugins` directory trees.

2. Start the Eclipse with a new workspace using one of these techniques:

   - Open a command prompt, change to the <install-dir>\eclipse directory (for example, c:\eclipse2.1\eclipse) for the Workbench instance that you are using for this exercise, and start the Workbench by entering eclipse –data altworkspace on the command line
   - Create and then run a shortcut for the eclipse.exe with these parameters:

   ```
   eclipse –data altworkspace
   ```

   The workbench will display the splash image shown in Figure 13-7 as it initializes the configuration for the new workspace. If this image is not displayed you are using a workspace that already exists.

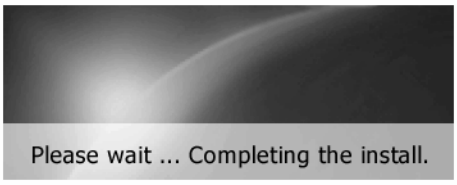


Figure 13-7
Eclipse initialization

After Eclipse has started, you should have a new workspace directory (`altworkspace`) in the same directory as the `eclipse.exe` program.

Important: Close Eclipse before continuing to the next step.

3. Add a link file to the Eclipse platform install to identify the new feature location. This task would normally be done by the install routine for an extension. A link file connects an Eclipse install with the extension install site; an install site is a directory that contains an eclipse directory tree with features and plug-ins.

   Create a directory named `links` in the `<install-dir>\eclipse` directory (for example, `c:\eclipse2.1\eclipse\links`) and add a file named `labpkg.link` with this content:

   ```
   path=d:\\labpkg
   ```

   The entry is treated as a properties file entry, so an escape sequence is necessary for the backslash (i.e., "\ \"). Since the entry is a URL, it could be entered as:

   ```
   path=d:/labpkg
   ```


This will instruct the Eclipse platform to look for additional features and their associated plug-ins in the `d:\labpkg` directory during startup. The link file content identifies where you have unzipped the feature and plug-ins that you previously packaged. You can use any file name you want for the link file. However, you should use a file name that will be unique when adding to a product that may be extended by others. You may want to consider using your feature id as the file name (`com.ibm.lab.tool.pkg.link`).

Note: Be sure there are no trailing blanks in the entry. Trailing blanks, in V2.0 of the Eclipse platform, will result in the link entry being ignored (see `http://dev.eclipse.org/bugs/show_bug.cgi?id=22993`).

4. Start the Eclipse using the `altworkspace`.

   If you watch carefully, you will see that the Eclipse platform knows you made a change. On the first invocation, Eclipse completes the install as was shown by the "Please wait… Completing install" information image shown in Figure 13-7. When changes are detected during subsequent invocations, the Workbench processes the changes and prepares for a possible acceptance of the new feature(s). This is visible at startup time because the splash screen will appear twice.

5. Accept the new feature as part of the current configuration.

   Once Eclipse has started, you are prompted about new updates. Accept the option to open up the Update Manager Configuration Changes dialog. Eclipse prompts when it has discovered that the existing configuration (stored in the `platform.cfg` file in the `workspace\.metadata\.config` directory) might need to change because of new features that exist. The features might have been added to the configuration by extension (link file) or directly added to the existing `features` and `plugins` directories; either way, you will be prompted to accept the configuration modification.

   Note: If you just add a plug-in without a corresponding feature, you do not get to decide if you want to add the new function to the active configuration. Eclipse will detect the change, and the plug-ins are unconditionally added to the configuration. Plug-ins that are not referenced by a feature are *unmanaged plug-ins* and are not recognized by the Update Manager user interface.

   Use the Configuration Changes dialog to select the change and add it to the current configuration as shown in Figure 13-8.

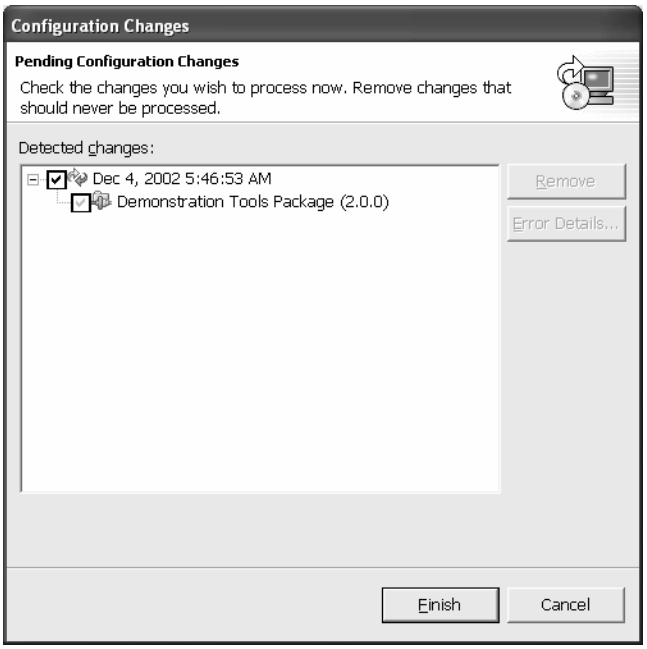


Figure 13-8
Installing Exercise Feature

Once you have selected the root entry and clicked on **Finish**, you will be prompted to restart Eclipse. Allow Eclipse to shut down and then restart. This activates the new configuration.

6. Validate that the new feature functions are available.

   The configuration change should have added the packaged feature and associated plug-ins to Eclipse. Use one or more of the following to confirm that this is true:

   - Open the About product dialog (or Help > About product name…) and look for the Tools Package feature icon. You should see icons for the active feature families, one of which is for the Tools Package (see Figure 13-9).

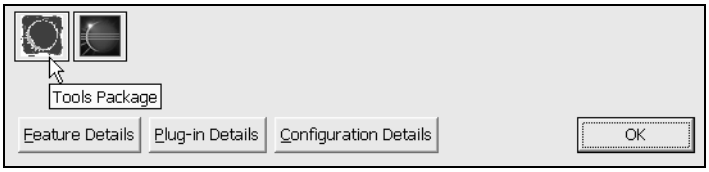


Figure 13-9
About Product Dialog Feature Icon

A family of features is represented by one image; features are in the same family when their images are identical. The About *product* dialog logic will compare the images for the available features, only unique images are shown.

- Select the Feature Details button in the About product dialog, the *Demonstration Tools Package* feature will be in the table of active features.

- Open the Install/Update perspective (Help > Software Updates > Update Manager) and expand the current configuration to find the *Demonstration Tools Package* feature, as shown below in Figure 13-10:

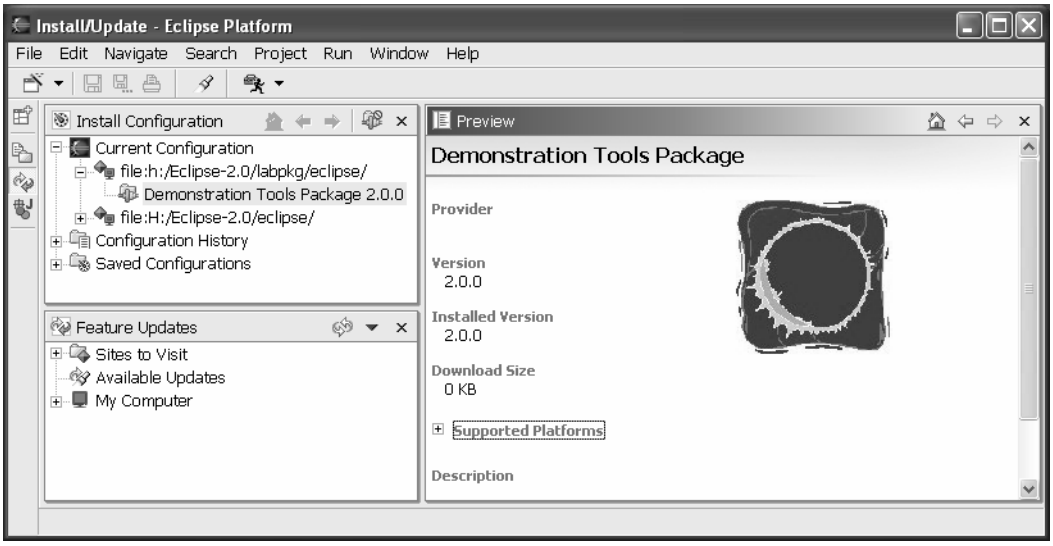


Figure 13-10
Installed Feature as shown in Install/Update Perspective

- Open the New Wizard (File > New > Other… or Ctrl+N) and find any wizards added as part of the Dialogs exercise (the *Lab: My New Wizards* or *Soln: My New Wizards* category will exist).
- Open the My First View created as part of the View exercise.
- Run the First Plug-in action in the current perspective; you may need to reset or customize the current perspective in order to add the action.

Any of the above should prove that the feature has been added and is functional in the new Workbench configuration. The function provided by the plug-ins you referenced in the `feature.xml` fill would be available.

Step 7. Add a feature to an existing product configuration from an update site
This approach is different from the previous "link file" technique. The Install/Update perspective of Eclipse will be used to pull a feature into the current configuration. In this exercise the feature site is local (on the file system), but a site could just as easily be found using a `http://` reference if the site were loaded onto a Web server.

1. Remove the `eclipse\links\labpkg.link` file from the Workbench installation you used during the previous install test (you may want to just move it to another directory, such as `eclipse\link-out`).

   If you have not done so already, close the Eclipse instance that was used in the previous install test.
2. Start Eclipse using the same (alternative) workspace used during the previous install test.
3. Use the Install/Update perspective to locate the Tools Package site on the file system.

Open the Install/Update perspective (**Help > Software Updates > Update Manager**) and then use the Feature Updates view to navigate to the `\labsite\site.xml` location in the file system.

4. Install the Tools Package feature. Once selected in the Feature Updates view, the Tools Package feature can be installed.

   Press the Install button, accept the license, use the default install location, and click on **Finish,** and then **Install,** to add the feature to the current configuration.

   You will be prompted to restart the Workbench. Select yes. Eclipse shuts down and then restarts. This activates the modification to the configuration.

5. Validate that the new feature functions are available.

   The configuration change should have added the packaged feature and associated plug-ins to Eclipse. Use one of the techniques described earlier to validate that the function is available as expected.

Part 3: Tasks of a Product Developer

In Part 3 you will create your own branded version of the runtime platform.

In the previous step, you used the Update Manager to add the Tools Package feature and associated plug-ins to the existing Eclipse platform installation directory. We are now going to alter the configuration so that the directory contains an installed and branded Eclipse platform-based product.

Important: As part of this process, you will delete the configuration information that tells the Eclipse platform that the Tools Package feature was installed. Delete the existing workspace to reset the installation. In the previous step, the `com.ibm.lab.tool.pkg` feature was installed in the same directory tree as the Eclipse platform. The Update Manager install registered the feature in the `workspace/.metadata/.config/platform.cfg` file. By deleting the workspace, the feature will be viewed as part of the Eclipse platform install on subsequent invocations.

Step 8. Implement a branded product

You will identify the Tools Package feature as the primary feature, the branding plug-in will then contribute additional information and graphics to Eclipse startup and user interface.

1. Modify the `install.ini` file in the `eclipse` directory.

   The install.ini file identifies the primary feature. This value is read by the `eclipse.exe` during startup processing.

   Edit the `install.ini` file and change the `feature.default.id` setting to `feature.default.id=com.ibm.lab.tool.pkg`

2. Identify the target feature as a primary feature. Edit the `feature.xml` file in the `com.ibm.lab.tool.pkg` project and add the primary="true" attribute setting to the feature entry (see highlighted text):


```
<?xml version="1.0" encoding="UTF-8"?>
<feature
      id="com.ibm.lab.tool.pkg"
      label="Demonstration Tools Package"
      version="2.0.0"
      provider-name=""
      image="FeatureUpdate.jpg"
      primary="true">
```

Step 9. Launch and review a branded product installation

This will allow you to see how the Tools Package feature, now defined as the primary feature, has added branding content to the Workbench user interface.

1. Close, if required, and then start the Workbench with a new workspace, or delete the previous workspace (`altworkspace`) to use the short cut or command line invocation.

   You should see a new splash screen during startup processing.

2. Review the product branding information (About *product* dialog, Install/Update perspective content).

   There are many indicators of the active primary feature in use by a Workbench instance. The identified primary feature controls the Workbench product branding. Content from the branding plug-in is used to modify the system icon, default welcome page, About *product* dialog, configuration shown in the Install/Update perspective, and window title.

   Many of these changes are visible in the Workbench image shown below:

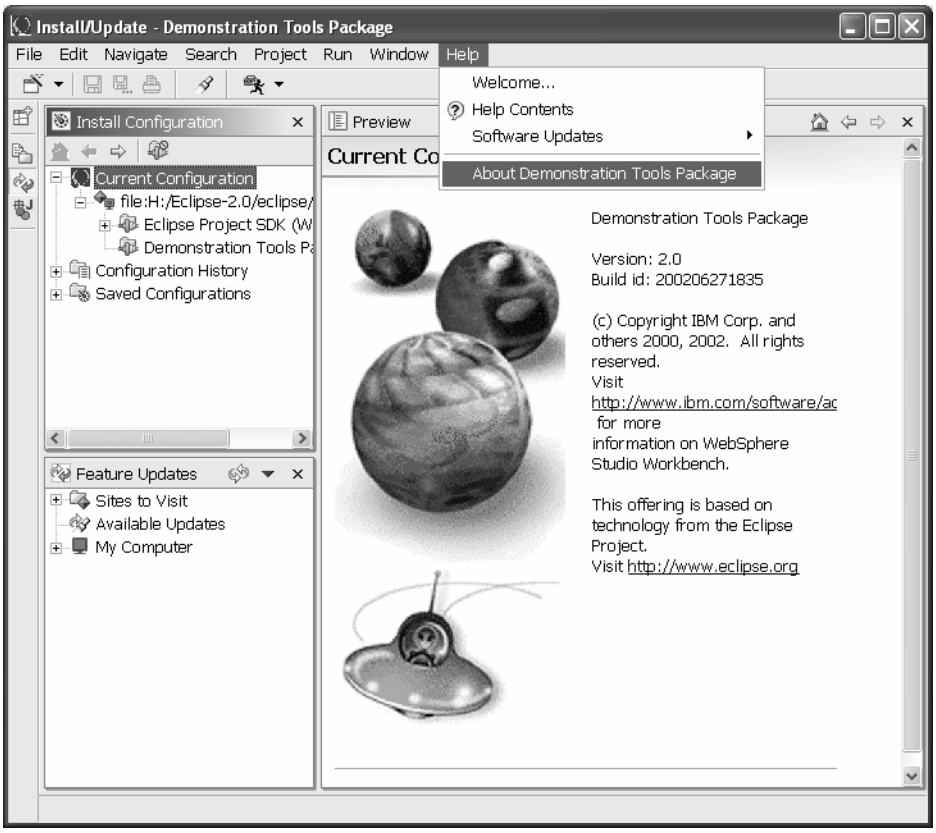


Figure 13-11
Workbench with customized product branding

Exercise Activity Review

What you did in this exercise:

- Learned how to use the PDE and Ant processing to automate the creation of runtime JAR files for a plug-in
- Identified the content required in a `build.properties` file to instruct the Ant processing on what should be included in a runtime JAR, plug-in install JAR, and feature install JAR.
- Defined features to organize plug-ins and provide feature and product branding.
- Packaged the feature and plug-ins to create runtime JAR files and plug-in install JAR files for each plug-in and a feature install JAR for the feature project.
- Learned how to installed a feature using a variety of techniques (extension using a link file, installation using the Update Manager to pull code into the Workbench configuration)

Configured a branded product by adding an alternative primary feature to an Eclipse platform installation.

Exercise 14: SWT Layouts

Introduction

Exercise Concepts

This exercise will show you how to use all the major SWT layout managers. At the completion of the lab all major layouts: `FillLayout`, `RowLayout`, `GridLayout`, and `FormLayout` can be viewed side by side for comparison purposes.

Skill Development Goals

At the end of this lab, you should be able to use basic elements of all the major layout managers and understand their differences.

Exercise Setup:

A PDE project has been setup for you named `com.ibm.lab.layouts`. As a container for our layouts, we will use a view. You may not be familiar with views, so a plug-in containing an empty view, and the class defining the view, has been defined for you. Load the lab project `com.ibm.lab.layouts` into your workspace. A file named `LayoutViewScrapbook.jpage` is available to assist you.

Exercise Instructions

Part 1: Add SWT button controls to the view

1. A view class has been defined in the project named `LayoutView`. It is an empty view (you can actually display it if you test the PDE project). The view is named **Lab: Layout Lab**.
2. Define the following button fields. They will be incorporated in several layouts:

```
Button b1;
Button b2;
Button b3;
Button b4;
Button b5;
```

To clear up any compile problems use the editor context menu called **Organize Imports**. You will need to use this feature frequently in the lab. Of course, if you typed "Button" followed by Ctrl-Space to activate content assist, the `import` statement would be generated as well.

3. Define method named `setButtons` to create and initialize these controls.

```
void setButtons(Composite parent) {
  b1 = new Button(parent, SWT.PUSH);
  b2 = new Button(parent, SWT.PUSH);
  b3 = new Button(parent, SWT.PUSH);
  b4 = new Button(parent, SWT.PUSH);
  b5 = new Button(parent, SWT.PUSH);

  b1.setText("Button 1");
  b2.setText("Button 2");
  b3.setText("Button 3");
  b4.setText("Button 4");
  b5.setText("Button 5");
}
```

4. In method `createPartControl`, add the following code that will define a vertically oriented fill layout that will contain a set of `Group` controls, one for each layout we will define.

```
FillLayout fillLayout = new FillLayout();
fillLayout.type = SWT.VERTICAL;
parent.setLayout(fillLayout);
```

Part 2: Add FillLayout Manager

1. Create a new method named `setFillLayout`. This is identical to what we just did, but we are going to populate it with the buttons we created earlier.

```
void setFillLayout(Composite parent) {
  setButtons(parent);
  FillLayout fillLayout = new FillLayout();
  fillLayout.type = SWT.VERTICAL;
  parent.setLayout(fillLayout);
}
```

2. In method `createPartControl`, define a Group control that will contain our fill layout of buttons in the previous step.

```
Group groupFill = new Group(parent, SWT.NONE);
groupFill.setText("Fill Layout");
```

This group will participate in the fill layout we previously defined. Call `setFillLayout` to imbed the button set in this new group.

```
setFillLayout(groupFill);
```

We will repeat this pattern as we define additional layouts.

3. Test your plug-in. In the test instance of the workbench, open the **Lab: Layouts Lab** view. It will be listed under **Window > Show View > Other**. Open view **Lab: Layout > Lab: Layouts Lab.** It should look like Figure (when the view is fully expanded).

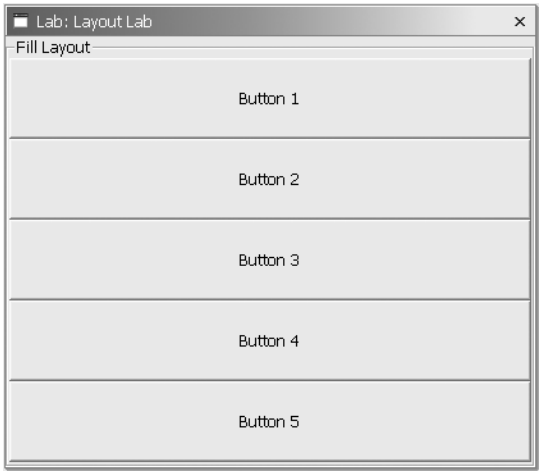


Figure 14-1
Simple FillLayout [layout_01.tif]

4. Not very interesting, is it! If you were to change the style bit on the buttons to `SWT.CHECK` you would get a set of vertical check boxes. Figure might be a more useful application of a fill layout.

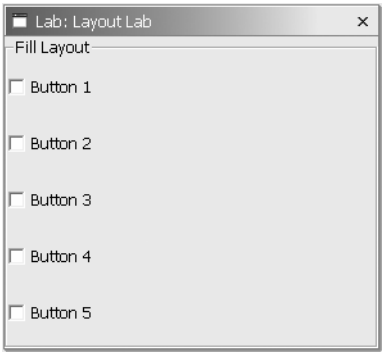


Figure 14-2
Simple FillLayout using checkboxes [layout_02.tif]

Part 3: Add a RowLayout Manager

1. Let's create a `RowLayout`. Create the following method.

```
void setRowLayout(Composite parent) {
  setButtons(parent);
  RowLayout rowLayout = new RowLayout();
  rowLayout.wrap = true;
  rowLayout.pack = false;
  rowLayout.justify = true;
  rowLayout.marginLeft = 5;
  rowLayout.marginTop = 5;
  rowLayout.marginRight = 5;
  rowLayout.marginBottom = 5;
  rowLayout.spacing = 10;
  parent.setLayout(rowLayout);
}
```

We have defined the layout to have margins of five pixels. Controls will wrap to the next row, if required, and the size of the buttons will not change, if the containing window is resized.

Add this code to the `createPartControl` method to define a `Group` and populate it with our `RowLayout`.

```
Group groupRow = new Group(parent, SWT.NONE);
groupRow.setText("Row Layout");
setRowLayout(groupRow);
```

Note that we are defining the same buttons, `b1` through `b5`, using the `setButtons` method. Obviously, we are doing this for illustration and convenience. Normally, widgets would not be used in this way. However, are we creating issues with respect to widget disposal by specifying button `b1` in `RowLayout` and `FillLayout` (and shortly in `GridLayout` as well)? When `b1` is disposed, which one will be disposed? Does the second and third use of button `b1` result in losing track of the previous uses of `b1`? You need not worry. First, we are creating separate instances of the object each time. Second, each instance has its own `Composite` parent; and it is the job of the parent to dispose of its children. Moreover, each instance that `b1` once referred to can have its own listener. These are, effectively, separate widgets that can have their own behavior. If you were to add the following code to the `setButtons` method, you could observe each instance of `b1` in operation. Note that `ourParent` is defined as `final` (it never changes) so we that we are permitted to reference it in the `SelectionAdapter` inner class.

```
final Composite ourParent = parent;
b1.addSelectionListener(new SelectionAdapter() {
  public void widgetSelected(SelectionEvent e) {
    System.out.println(
      "Button b1 pressed. Parent is " + ourParent.toString());
  }
});
```

This code displays the following on the console.

```
Button b1 pressed. Parent is Group {Fill Layout}
Button b1 pressed. Parent is Group {Row Layout}
```

2. Retest the plug-in and you should see a view that looks like Figure .

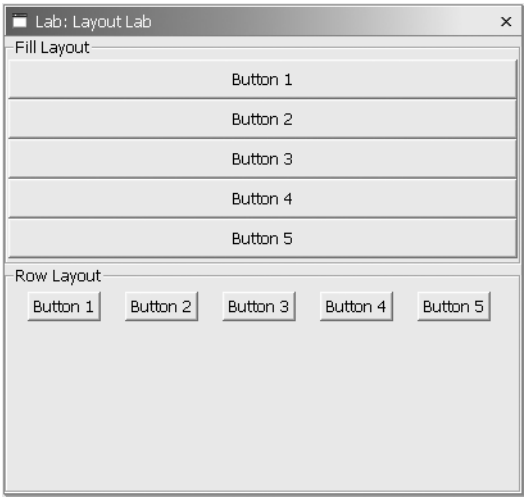


Figure 14-3
Addition of a simple RowLayout [layout_03.tif]

By adjusting the window size, you can see how the `wrap` attribute behaves.

3. You can adjust other layout attributes like `wrap`, `pack`, and `justify` and observe the changes.

Part 4: Add a GridLayout Manager

Let's add a simple grid layout to our collection of layouts. Actually, we will add two. We will reuse our buttons in one grid layout and then create another using Labels and Text fields. This latter one will provide a transition to our final layout, a form layout.

1. Create a method named `setGridLayoutButtons`. In this method, we will create a grid two columns wide. Button 3 will be wider than the others due to its label length. Button 5 will be centered within its cell of the grid by defining a `GridData` object for it.

```
void setGridLayoutButtons(Composite parent) {
  GridLayout gridLayout = new GridLayout();
  gridLayout.numColumns = 2;
  setButtons(parent);
  b3.setText("Wider Button 3");
  GridData b5GridData = new GridData();
  b5GridData.horizontalAlignment = GridData.CENTER;
  b5.setLayoutData(b5GridData);
  parent.setLayout(gridLayout);
}
```

2. In method `createPartControl`, define a Group control that will contain our grid layout. This is just as we did for the row layout earlier.

```
Group groupGridButtons = new Group(parent, SWT.NONE);
groupGridButtons.setText("Grid Layout: Buttons");
setGridLayoutButtons(groupGridButtons);
```

3. Test the plug-in and you should see a view that looks like Figure .

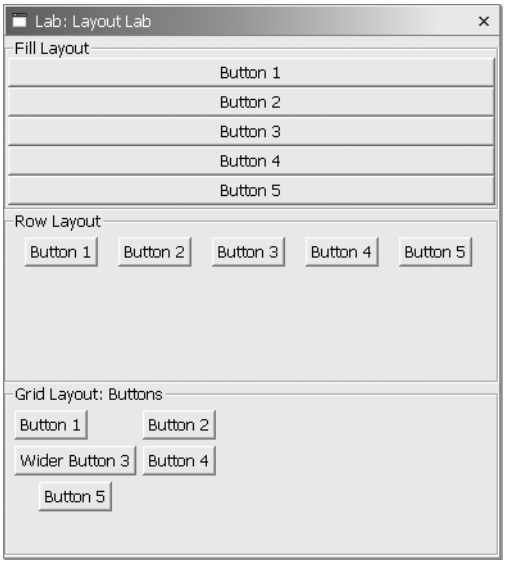


Figure 14-4
Addition of a simple GridLayout [layout_04.tif]

4. Observe that, unlike the row layout, shrinking this view will not reposition the buttons.

Part 5: Create Another Grid Layout

1. Let's repeat the process again with a simple name and address layout.
2. Add the following label and text fields to the class.

```
Label firstNameLbl;
Text firstName;
Label lastNameLbl;
Text lastName;
Label streetLbl;
Text street;
Label cityLbl;
Text city;
Label stateLbl;
Combo state;
```

3. Create a new method named `setFields` as described below to define your widgets and initialize them. Observe the use of multiple style bits in the constructor for several of the fields.

```
void setFields(Composite group) {
  firstNameLbl = new Label(group, SWT.SINGLE);
  firstName = new Text(group, SWT.SINGLE | SWT.BORDER);
  lastNameLbl = new Label(group, SWT.SINGLE);
  lastName = new Text(group, SWT.SINGLE | SWT.BORDER);
  streetLbl = new Label(group, SWT.SINGLE);
  street = new Text(group, SWT.SINGLE | SWT.BORDER);
  cityLbl = new Label(group, SWT.SINGLE);
  city = new Text(group, SWT.SINGLE | SWT.BORDER);
  stateLbl = new Label(group, SWT.SINGLE);
  state = new Combo(group, SWT.DROP_DOWN | SWT.BORDER | SWT.SIMPLE);

  firstNameLbl.setText("First Name: ");
  lastNameLbl.setText("Last Name: ");
  streetLbl.setText("Street: ");
  cityLbl.setText("City: ");
  stateLbl.setText("State/Province: ");
  state.setItems(new String[] { "AL", "AK", "AZ", "..." });
}
```

4. With this in hand, create a method called `setGridLayoutAddress` as described below. This will be defined as a four-column grid. We will layout the fields in a label-text field pattern.

```
void setGridLayoutAddress(Composite parent) {
  GridLayout gridLayout = new GridLayout();
  gridLayout.numColumns = 4;
  // Layout name and address fields into the grid
  setFields(parent);
  parent.setLayout(gridLayout);
}
```

5. In method `createPartControl`, define a `Group` control that will contain our grid layout. This is just as we did for the row layout earlier.

```
Group groupGridAddress = new Group(parent, SWT.NONE);
groupGridAddress.setText("Grid Layout: Name and Address Form");
setGridLayoutAddress(groupGridAddress);
```

6. Test your plug-in. It should look like Figure .

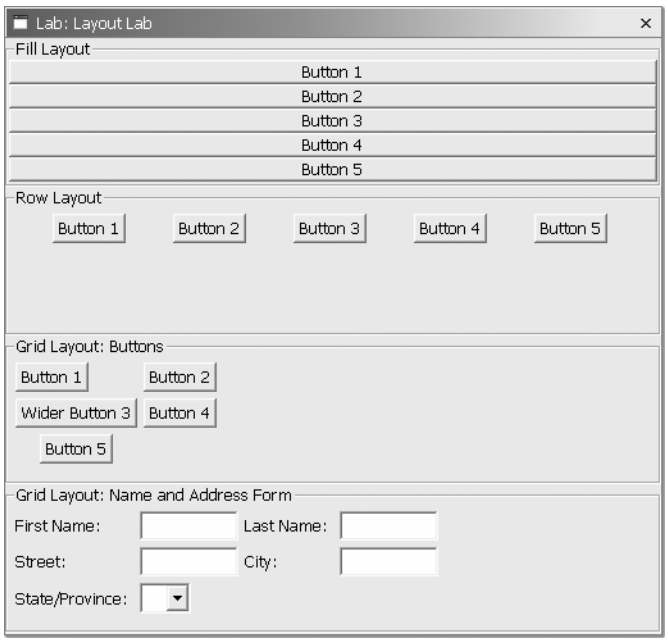


Figure 14-5
Addition of a set of name and address fields using GridLayout [layout_06.tif]

Part 6: Add a FormLayout Manager

1. Lastly, let's build a name and address layout using `FormLayout`. We want to exert a little more control over the appearance. When completed it will look like Figure .

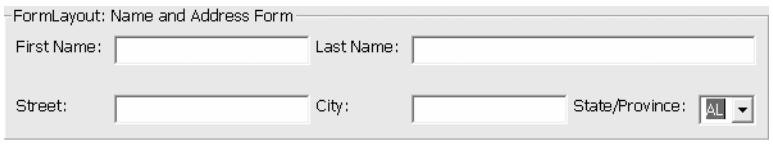


Figure 14-6
Name and address using FormLayout [layout_05.tif]

2. Create a new method named `setFormLayoutAddress` as described below.

```
void setFormLayoutAddress(Composite parent) {
  // Simple name and address input form
  FormLayout addressLayout = new FormLayout();
  addressLayout.marginWidth = 3;
  addressLayout.marginHeight = 3;
  parent.setLayout(addressLayout);
  setFields(parent);
}
```

In this method, we have defined a `FormLayout`, defined margins of 3 pixels and linked it to the parent. Finally, we have initialized the fields as did before. We have not yet laid out the fields onto the form.

3. In method `createPartControl`, define a `Group` control that will contain our form layout.

```
Group groupFormAddress = new Group(parent, SWT.NONE);
groupFormAddress.setText("Form Layout: Name and Address Form");
setFormLayoutAddress(groupFormAddress);
```

4. In the `setFormLayoutAddress` method, let's add a text field for first name and precede it with a label. These two fields will be located inside the group box. The label will be placed 5 pixels from the left and 5 pixels from the top of the group box. The text field will be placed 5 pixels to the right of the label. Its right edge will be at 40% of the group box width.

   Add the following code to the end of the method.

```
// FirstName. 5 pixels from top and left edge of group
FormData firstNameLblFD = new FormData();
firstNameLblFD.left = new FormAttachment(0, 5);
firstNameLblFD.top = new FormAttachment(0, 5);
firstNameLbl.setLayoutData(firstNameLblFD);

FormData firstNameFD = new FormData();
// place first name adjacent its label plux 5 pixels
firstNameFD.left = new FormAttachment(firstNameLbl, 5);
// place right edge of text field at 40% of parent width
firstNameFD.right = new FormAttachment(40, 0);
firstNameFD.top = new FormAttachment(0, 5);
firstName.setLayoutData(firstNameFD);
```

We could try testing at this point, but recall that in the `setFields` method we defined all the fields for the form. They will appear on top of each other until they are laid out. An alternative approach would be to refactor the `setFields` method so that the fields are defined in this method.

5. Now let's add the last name label and text field. That will complete the first line of the form. The last name label will be adjacent the first name text field plus five pixels. The last name text field will be adjacent its label and the right edge will extend to the right edge of the parent group control less a five pixel margin. Add this code to the end of the method.

```
// Last Name
FormData lastNameLblFD = new FormData();
lastNameLblFD.left = new FormAttachment(firstName, 5);
lastNameLblFD.top = new FormAttachment(0, 5);
lastNameLbl.setLayoutData(lastNameLblFD);
FormData lastNameFD = new FormData();
lastNameFD.left = new FormAttachment(lastNameLbl, 5);
// set right edge at right edge of
// parent less 5 pixels
lastNameFD.right = new FormAttachment(100,-5);
lastNameFD.top = new FormAttachment(0,5);
lastName.setLayoutData(lastNameFD);
```

6. Let's complete the second line of the form. Add the following constant to the class. We will use this as an offset value in pixels to position the second line contents.

```
static final int line2offset = 30;
```

7. When we add the street label, we want to position the top relative to first line plus thirty pixels, and we want it to line up vertically with the first name label.

```
// Street
FormData streetLblFD = new FormData();
streetLblFD.left = new FormAttachment(0, 5);
//locate street label vertically relative
//to previous line
streetLblFD.top = new FormAttachment(firstNameLbl, line2offset);
streetLbl.setLayoutData(streetLblFD);
```

8. Let's move along and add the street text field, city label, city text field.

The street text field lines up vertically with the first name text field. Its right edge lines up with the last name field less five pixels.

The city label's left edge lines up with the last name label and its right edge lines up with the last name text field.

The City text field lines up with the last name text field. The right edge of the City text field is positioned at 75% of the parent group width less five pixels. This leaves reasonable room for the State/Province field. How was this determined? Trial and error!

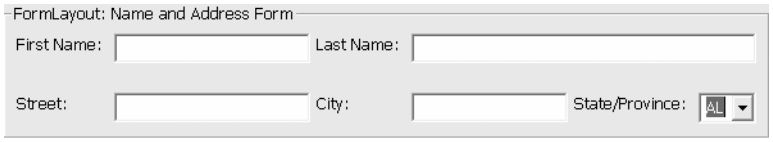


Figure 14-7
Name and address FormLayout [layout_05.tif]

```
FormData streetFD = new FormData();
// line up vertically with firstname
streetFD.left = new FormAttachment(firstNameLbl, 5);
streetFD.right = new FormAttachment(lastNameLbl, -5);
streetFD.top = FormAttachment(firstNameLbl, line2offset);
street.setLayoutData(streetFD);
// City
FormData cityLblFD = new FormData();
// line up vertically with last name
cityLblFD.left = new FormAttachment(firstName, 5);
// locate street label vertically relative
// to previous line
cityLblFD.top = new FormAttachment(firstNameLbl, line2offset);
cityLbl.setLayoutData(cityLblFD);
FormData cityFD = new FormData();
cityFD.left = new FormAttachment(lastNameLbl, 5);
cityFD.right = new FormAttachment(75, -5);
cityFD.top = new FormAttachment(firstNameLbl, line2offset);
city.setLayoutData(cityFD);
```

9. The state label and combo field completes the layout. This should be routine by now!

```
// State
FormData stateLblFD = new FormData();
stateLblFD.left = new FormAttachment(city, 5);
// locate street label vertically
//relative to previous line
stateLblFD.top = new FormAttachment(firstNameLbl, line2offset);
stateLbl.setLayoutData(stateLblFD);
FormData stateFD = new FormData();
stateFD.left = new FormAttachment(stateLbl, 5);
stateFD.right = new FormAttachment(100, -5);
stateFD.top = new FormAttachment(firstNameLbl, line2offset);
state.setLayoutData(stateFD);
```

10.Test your plug-in. It should look like Figure .

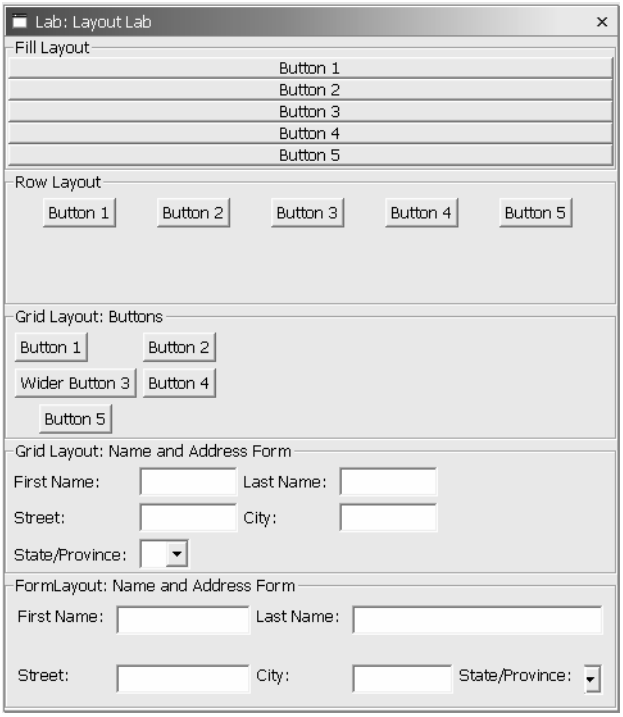


Figure 14-8
Completed lab using all four layout managers [layout_08.tif]

If you resize the workbench window small enough, you will notice that the contents get clipped. Typically, this is a fixed size dialog so this is less of a problem. Also, notice that the fill layout that was used to lay out our groups results in all groups being the same size even though the contents vary. This choice of layout manager, which is convenient for this exercise, might not be the choice you would make in a real situation. If you have time, experiment with using `RowLayout` instead of `FillLayout` at the beginning of the `createPartControl` method. Changing this line of code should do the trick:

```
FillLayout fillLayout = new FillLayout();
```

Replace with:

```
RowLayout fillLayout = new RowLayout();
```

Additional Investigation of Layouts

Try using the SWT Layouts Example that is part of the workbench examples (assuming you installed the workbench examples). Open view **SWT Examples >SWT Layouts**. You can lay out a variety of widgets in any of the four layout managers. It will even generate the code for you.

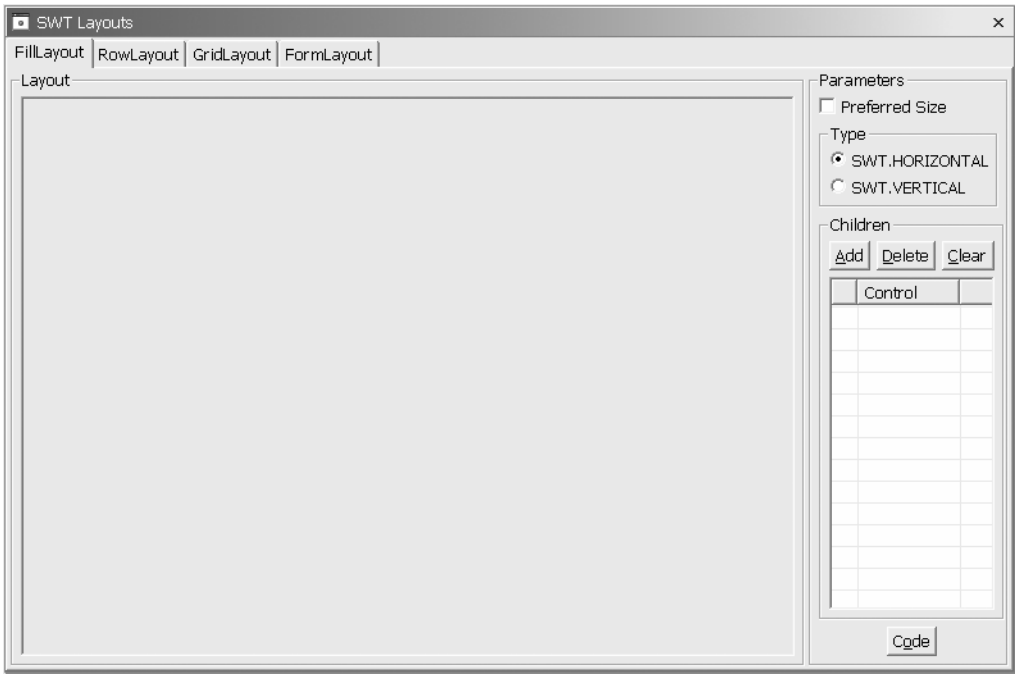


Figure 14-9
SWT Layouts Example Tool [layout_07.tif]

Exercise Activity Review

- Examined all the major SWT layout managers.

Exercise 15: Extending the Java Development Tools

Introduction

This exercise will familiarize you with the keys classes and interfaces of the Java Development Tools plug-in. You will create several code examples that create and process Java Abstract Syntax Trees (AST).

Exercise Concepts

During this exercise, you will implement one or more of the following:

- A simple "Hello AST" class that displays the structure of an AST created from a hardcoded string representing a simple Java program and a user-provided Java source file example.
- A Java Metrics view that shows the number of methods, fields, and literals within the selected

*.java file (`ICompilationUnit`). The view automatically updates when modifications are saved or new methods / fields are added.

Skill Development Goals

At the end of this exercise, you should be able to:

- Create and analyze an AST using subclasses of ASTVisitor.
- Create an "Add Trace Statements" menu choice for the Java editor. This choice will analyze the source and insert trace statements that output the name and parameters of each method.
- Integrate a new view with the JDT model, and extend the Java editor.

Exercise Setup

Common lab setup tasks should have been performed such that a PDE project named `com.ibm.lab.jdt` should be available in Workbench PDE perspective. The JDT lab template provides you with a plug-in template and a base set of code, but the implementation of some code components is incomplete. You will add new methods and references to other methods not yet invoked by the base code.

The `com.ibm.lab.jdt` project contains the following files:

- `plugin.xml` – the plug-in manifest file. The focus of this exercise will be the JDT and its framework, so the provided manifest files is already complete.
- `metrics.gif` and `addtrace.gif` - graphic files that are referenced by the plug-in manifest file.
- Partially coded classes that will be completed during the lab.
- A set of Java scrapbook pages, organized by class, which contain code fragments referenced in this lab. You can use these to minimize the need to retype code defined in this chapter.
- Other files, such as `.classpath`, which we are not working with in this exercise.

Exercise Instructions

Part I: Traversing a Simple AST with HelloAST

This first section of the exercise has you create a new class that will print the basic structure of a given AST to `System.out`. The idea is to get a quick tour of the `ASTVisitor` class and see how it helps you easily analyze Java code. As a bonus, this class is both instructive and helpful when debugging more complex ASTs.

As the name "`HelloAST`" suggests, this code will be primitive in its choice of input and output: `System.in` and `System.out`. This demonstrates the fact that the JDT model in the `org.eclipse.jdt.core` package, which defines the various Java elements, has no dependency on the JDT user interface package `org.eclipse.jdt.ui`.

Step 1: Create ASTVisitor subclass

The `ASTVisitor` subclass is responsible for processing instances of the desired subtypes of `ASTNode`'s. In this first exercise, we want to print a hierarchical list of all the nodes.

1. Create a subclass of `ASTVisitor` called `ASTNode2StringVisitor`.
2. Take a look at its superclass' implementation using the Hierarchy view. You will see that there are several categories of public methods:
   - a method for each node type that a visitor can visit, for example `visit(Block)`, `visit(Javadoc)`, etc. The `visit(XXX)` method's return value, true or false, indicates whether the visitor wishes to continue by visiting the current node's children, or visiting the next sibling node.
   - a `endVisit(…)` method for each node type, called after the node's children are visited.
   - and two arbitrary node-type methods, `preVisit(ASTNode)` and `postVisit(ASTNode)`, that are invoked for all types before and after the type-specific `visit` methods.
3. In this particular case, we want to visit all nodes, including their children. We do not have any specific code for a given node type, so override the `preVisit(ASTNode)` method. You can type the method signature yourself based on the superclass, or select the class in the Outline view and use the **Override Methods…** pop-up menu choice to help you insert the `preVisit(…)` method body.
4. All that's left to do is print the node. The ASTNode class provides a `toString()` method, but it is a bit too sophisticated for our needs here, so instead, only print the class name itself. Enter:

```
System.out.println(node.getClass())
```

in the method body of `preVisit(..)` and save. You might wonder if it is necessary to call `super.preVisit(…)`. There certainly would be no harm in doing so, but we know in this case that the ASTVisitor class only provides a default implementation. In particular, `ASTVisitor.preVisit(ASTNode)` is an empty method body.

Step 2: Complete HelloASTAction class

We are ready to use the `ASTNode2StringVisitor` to help us display an AST. Recall that to further simplify this example, we're only using `System.in` and `System.out` for our I/O.

1. Open the class fragment, `HelloASTAction`. Note it is a `IWorkbenchWindowActionDelegate` subclass, so we can invoke its `run` method by selecting **JDT Exercise > Run HelloAST** from the workbench menu when we're ready to test.
2. The AST class provides two instance methods for parsing. The first accepts a character array (Java source), the second accepts an instance of `ICompilationUnit`. We'll use the first to test our node visitor. To save you typing, the `helloWorldExample()` method returning a sample Java source string is already coded in `HelloAST`, you only need to call it in the `run` method:

```
String javaCode = me.helloWorldExample();
System.out.println(javaCode);
```


Use an AST to analyze the "Hello World" snippet of code as follows:

```
CompilationUnit cu =
    AST.parseCompilationUnit(javaCode.toCharArray());
```

Note: The `CompilationUnit` in this case is a subclass of `ASTNode`, not an instance of `ICompilationUnit`. The latter is the in-memory representation of the *.java file itself and is part of the JDT model (e.g., it knows it is contained in a project). The former is an AST-specific class that only represents Java language elements. It knows nothing about projects, since they are specific to the Eclipse Workbench. The public AST classes are found in the `org.eclipse.jdt.core.dom` package. Implementations of the JDT public interfaces, like `ICompilationUnit`, are located in the `org.eclipse.internal.jdt.core` package. You will want to import the `org.eclipse.jdt.core.dom.CompilationUnit` for the steps above.

3. Now that we have an AST, let's ask our visitor to visit it:

```
    ASTVisitor visitor = new ASTNode2StringVisitor();
    cu.accept(visitor);
```

Step 3: Test HelloAST

Save your code, and verify there are no compilation errors. You may need to add `import` statements. The **Organize Imports…** pop-up menu choice is very handy to detect and add the necessary imports, or you can select the class you wish to import and use **Source > Add Import** (Ctrl+Shift+M). Now we're ready to test.

1. Select the `HelloASTAction.java` file, then select the **Run As > Run-time Workbench** menu choice. Once the run-time instance has started, select the menu choice **Lab: JDT Exercise** and then select the **Lab: Run Hello AST** menu action (you may need to activate this action set to add it to your current perspective). You should see the result below in the Console:

```
package example;
public class HelloWorld {
public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
class org.eclipse.jdt.core.dom.CompilationUnit
class org.eclipse.jdt.core.dom.PackageDeclaration
class org.eclipse.jdt.core.dom.SimpleName
class org.eclipse.jdt.core.dom.TypeDeclaration
class org.eclipse.jdt.core.dom.SimpleName
class org.eclipse.jdt.core.dom.MethodDeclaration
class org.eclipse.jdt.core.dom.PrimitiveType
class org.eclipse.jdt.core.dom.SimpleName
class org.eclipse.jdt.core.dom.SingleVariableDeclaration
class org.eclipse.jdt.core.dom.ArrayType
class org.eclipse.jdt.core.dom.SimpleType
class org.eclipse.jdt.core.dom.SimpleName
class org.eclipse.jdt.core.dom.SimpleName
class org.eclipse.jdt.core.dom.Block
```



```
class org.eclipse.jdt.core.dom.ExpressionStatement
class org.eclipse.jdt.core.dom.MethodInvocation
class org.eclipse.jdt.core.dom.QualifiedName
class org.eclipse.jdt.core.dom.SimpleName
class org.eclipse.jdt.core.dom.SimpleName
class org.eclipse.jdt.core.dom.SimpleName
class org.eclipse.jdt.core.dom.StringLiteral
```

2. That's not bad, but wouldn't it be much better if it were properly indented? Solution: Walk up the hierarchy to calculate the proper indentation. Add this to the `preVisit` method of `ASTNode2StringVisitor`:

```
ASTNode tempNode = node.getParent();
while (tempNode != null) {
  System.out.print('\t');
  tempNode = tempNode.getParent();
}
```

Re-run the code and you should see the tree structure below:

```
class org.eclipse.jdt.core.dom.CompilationUnit
  class org.eclipse.jdt.core.dom.PackageDeclaration
    class org.eclipse.jdt.core.dom.SimpleName
  class org.eclipse.jdt.core.dom.TypeDeclaration
    class org.eclipse.jdt.core.dom.SimpleName
    class org.eclipse.jdt.core.dom.MethodDeclaration
      class org.eclipse.jdt.core.dom.PrimitiveType
      class org.eclipse.jdt.core.dom.SimpleName
      class org.eclipse.jdt.core.dom.SingleVariableDeclaration
        class org.eclipse.jdt.core.dom.ArrayType
          class org.eclipse.jdt.core.dom.SimpleType
            class org.eclipse.jdt.core.dom.SimpleName
        class org.eclipse.jdt.core.dom.SimpleName
      class org.eclipse.jdt.core.dom.Block
        class org.eclipse.jdt.core.dom.ExpressionStatement
          class org.eclipse.jdt.core.dom.MethodInvocation
            class org.eclipse.jdt.core.dom.QualifiedName
              class org.eclipse.jdt.core.dom.SimpleName
              class org.eclipse.jdt.core.dom.SimpleName
            class org.eclipse.jdt.core.dom.SimpleName
              class org.eclipse.jdt.core.dom.StringLiteral
```

3. We printed the name of the class of each node instead of using their default `toString()` method because they are principally for debugging and include more information than we need for our example. To see the difference, put a breakpoint on `System.out.println(node.getClass())` in `ASTNode2StringVisitor`, then launch the debugger. When it stops at this line of code, select `node` in the Variables pane and show details (see arrow below).

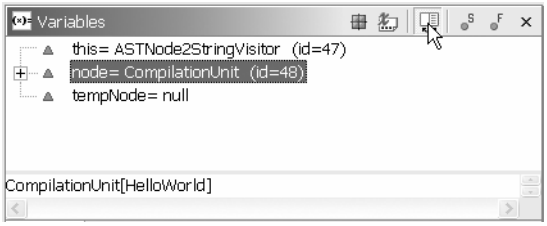


Figure 15-1
Variable Debug View [jdt_1.tif]

Resume several times and reselect the `node` variable. The Details pane shows the `toString()` output of the selected variable. Adding `toString()` methods to your own classes can similarly help you debug without adding `System.out.println(…)` statements.

5. The existing `getJavaExample()` method in `HelloASTAction` will create an AST from an arbitrary *.java file. Add a call to it in the `run` method, directly below our previous "hello world" AST example. The code below will prompt the user for a source file and return its content:

```
javaCode = me.getJavaExample();
System.out.println(javaCode);

cu = AST.parseCompilationUnit(javaCode.toCharArray());
visitor = new ASTNode2StringVisitor();
cu.accept(visitor);
```

Relaunch "Hello AST" and enter the fully qualified file specification of a *.java file in the Console when prompted, "`Enter a fully qualified Java filespec:`".

The size of the AST tree can increase dramatically, so don't choose a very large source file. There is already a HelloWorld.java example in your project. If you installed Eclipse in c:\eclipse2.0, this would be its path (one line, of course):

```
C:\eclipse2.0\eclipse\workspace\
    com.ibm.lab.jdt\src\com\ibm\lab\jdt\HelloWorld.java
```

This is available in the jpage, if you prefer to copy/paste it into the Console. You can try analyzing HelloASTAction.java in the same subdirectory, too.

Part II: Extending the Java Editor

In this exercise, you will reuse the AST skills that you've acquired in the previous part and add a new editor menu choice that adds trace statements to each method in a Java source file. The new concepts in this exercise are principally the workings of the Java editor itself, especially the "working copy" that all JDT editors use to stage modifications before committing them to the in-memory Java model.

Mini-review of JavaUI Working Copies (Read this later if you are pressed for time)

As a reminder, here's a review of what was covered in "Extending the JDT User Interface." We will implement some of these steps in our new editor action.

Java elements support the notion of a "working copy." This is a staging area for modifications separate from the model. This allows, for example, the editor or refactoring code to modify its client's source code and corresponding Java model without committing changes to the "real" Java model until desired.

Below is the sequence of editor events when the user opens, modifies, and saves a Java source file:

1. When the user opens a *.java file, the editor gets a working copy of the Java element, an implementor of `ICompilationUnit`. Again, the working copy is an instance implementing `ICompilationUnit`, but it isn't the Java element itself.

```
IWorkingCopyManager mgr = JavaUI.getWorkingCopyManager
  (compilationUnitEditor.getEditorInput());
ICompilationUnit cu =
  mgr.getWorkingCopy(compilationUnitEditor.getEditorInput());
```

   The working copy manager requires the editor input to create the working copy. This is to assure that the same editor input receives the same working copy. And it allows the editor framework to track change modifications (e.g., the famous dirty marker "*" is added when the working copy is modified).

2. The user makes their desired modifications. Since the modifications are against a working copy, Java element change listeners are notified with a `IJavaElementDelta` that refers to the uncommitted working copy, not the true Java model. This brings us back to the point that was mentioned in Part II regarding working copies and the `JavaMetrics.findJavaElementDelta(…)` method, that it, Java element notifications can refer to either the committed Java model or the working copy if `ElementChangedEvent.POST_RECONCILE` is specified in the call to `JavaCore.addElementChangeListener(IElementChangeListener listener, int event)` or the default event flags are accepted by calling `JavaCore.addElementChangeListener(IElementChangeListener)`, namely `ElementChangedEvent.POST_RECONCILE` and `ElementChangedEvent.POST_CHANGE.`
3. You have probably noticed that the Java editor's Outline view updates dynamically once you stop typing in the editor. The editor uses an internal timer to resynchronize with its helper, the Outline view, by calling `reconcile()` against the working copy. That invocation triggers an `elementChanged(…)` notification. The element delta is based on the original copied contents of the working copy and the changes that have occurred since its inception. In other words, the delta is based on the working copy at two points in time, not the working copy and the original source. This is because the changed contents have not yet been committed to the model
4. The user saves the modifications. During save processing, the working copy's `commit(…)` method is invoked to put its changed contents to the underlying in-memory Java model and it writes the changes to disk.

OK, with the above sequence in mind, let's start coding again.

Step 1: A Quick Look at the Solution

While there isn't much code and only one class, there are a lot of steps in this exercise. It helps to see where we are going with them, so load the solution first, launch the run-time

Workbench, and open an example Java source file. Select the "Soln: Add Trace Statements" pop-up menu choice, or the same action from the toolbar.

It should add trace statements like this to each method:

```
System.out.println("Start methodname" + " <" + parm1 + ">" +
" <" + parm2 + ">");
```

Did it add trace statements as expected?

Step 2: Create AddTraceStatementsEditorActionDelegate class

Previous labs have already covered the necessary extensions to add editor actions, and our case is similar. Here is the relevant extract of plugin.xml:

```
<extension
    point="org.eclipse.ui.editorActions">
  <editorContribution
    targetID="org.eclipse.jdt.ui.CompilationUnitEditor"
    id="com.ibm.lab.jdt.compilationuniteditor">
    <action
      label="Add Trace Statements"
      icon="addtrace.gif"
      tooltip="Add trace statements to each method"
      class="com.ibm.lab.jdt.AddTraceStatementsEditorActionDelegate"
      toolbarPath="additions"
      id="com.ibm.lab.jdt.addtracestatements1">
    </action>
  </editorContribution>
</extension>

<extension
    point="org.eclipse.ui.popupMenus">
  <viewerContribution
       targetID="#CompilationUnitEditorContext"
       id="com.ibm.lab.jdt.compilationunit.editorcontext">
    <action
      label="Add Trace Statements"
      class="com.ibm.lab.jdt.AddTraceStatementsEditorActionDelegate"
      menubarPath="additions"
      id="com.ibm.lab.jdt.addtracestatements2">
    </action>
  </viewerContribution>
</extension>
```

The editor action, `AddTraceStatementsEditorActionDelegate`, is added in two contexts for demonstration purposes only. Realistically, you would likely choose one or the other (toolbar or pop-up menu) as the best place to add it. The `"additions"` location is the default, generally at the end of the toolbar or pop-up menu, and is sufficient for our purposes.

Although the editor action delegate, `AddTraceStatementsEditorActionDelegate`, is referenced in two places (once for the editor toolbar as an editor contribution and a second time for the context menu as a pop-up menu contribution), the Workbench will nonetheless refer to the same instance in both cases. In other words, menu contributions are shared by all like-instances of an editor, indicated by their `targetID`.

Design Note: Sharing editor actions simplifies menu creation (e.g, no need to worry about duplicate menu choices if two editors are open, no need to rebuild menus if focus changes

to an editor with the same ID, etc.), but it requires that the editor action delegate be notified ***which*** editor it is being asked to act upon. We'll return to this point in just a moment.

1. Start by creating the referenced editor action class, `AddTraceStatementsEditorActionDelegate`. It must implement `IEditorActionDelegate`. Also remember to check the "inherited abstract methods" choice when creating the class so you'll have stub methods with which to work. Define an instance variable `cuEditor` of type `IEditorPart` before continuing to the next step.
2. As we discussed in the mini-review above, the editor action is in fact shared among open instances of its associated editor target. Modify the `setActiveEditor(…)` method to store the active editor. Then we'll know to which editor the action applies when `run()` is called to carry out the user's request:

```
public void setActiveEditor
    (IAction action, IEditorPart targetEditor) {
  cuEditor = targetEditor;
}
```

3. Similar to our prior use of `JavaMetricsAccumulator`, we'll use an inner class to define an operation against an AST, this time called `JavaMethodsCollector`. Create this class and include an instance variable, `methodDeclarations`, in which to collect the method declarations, and a getter for it:

```
    private class JavaMethodCollector extends ASTVisitor {
        public List methodDeclarations = new ArrayList();

        public List getMethodDeclarations() {
            return methodDeclarations;
        }
    }
```

4. Now we pick what node(s) we wish to work with – an easy choice, right? Just method declarations, since we plan to insert our trace statements at the beginning of each method. So add the method below to `JavaMethodsCollector`, overriding the superclass' no-op implementation:

```
    public boolean visit(MethodDeclaration node) {
        methodDeclarations.add(0, node);
        return false;
    }
```

Particularly observant readers might notice that the `add(…)` above will insert the node at the head of the list. This is intentional. We want the method declarations in reverse order (bottom to top) and will explain why later.

Design Note: It would be equally legitimate to have the visitor handle all aspects of the operation, not just gathering a list of method declarations. Or even have the editor action subclass from ASTVisitor and avoid an inner class completely. But for instructive purposes, it is clearer to separate these activities.

5. As before, we will use a constructor to start the visit:

```
    public JavaMethodCollector(ICompilationUnit cu) {
```



```
        AST.parseCompilationUnit(cu, false).accept(this);
    }
```

Nothing new here, but it will get more interesting in the next few steps (promised).

6. Now we have a visitor ready to collect all the method declarations; let's actually do something with them! Go to the `run()` method. This is the method that will be invoked when the user selects the toolbar button or pop-up menu choice, "Add Trace Statements." We start by getting a working copy of the editor's input, an implementor of `ICompilationUnit`:

```
IWorkingCopyManager manager = JavaUI.getWorkingCopyManager();
ICompilationUnit cu =
    manager.getWorkingCopy(cuEditor.getEditorInput());
```

7. Then we are ready to actually collect the method declarations.

```
List methodDeclarations = new
    JavaMethodCollector(cu).getMethodDeclarations();
```

Take a moment to review the public methods of `MethodDeclaration`. You'll find that it has everything that one can specify in a method signature: name, return type, parameters, thrown exceptions, etc. In particular, we are interested in where the first line of code for this method starts. That is in another AST node, `Block`, a child accessible directly from `MethodDeclaration`'s `getBody()` method. All `ASTNode`'s have a source starting position and length. For instances of `Block`, that is the locations of the opening and closing brace, respectively.

8. Now something we haven't talked much about until this point. You have the method, its name and parameters, and the location where you want to insert your trace statements. But how do you insert them such that the editor(s) – and user – will see them? This is where the `ICompilationUnit` (and its equivalent working copy) belie the fact that they are based on source text, specifically represented by an instance of `IBuffer`. The working copy's buffer content is what is displayed in an editor. To retrieve it, add the code below:

```
    IBuffer buffer = cu.getBuffer();
```

The buffer interface is similar to `StringBuffer`, e.g., `append(…)`, `replace(…)`, etc. Remember, the `cu` variable above is a working copy, so any changes we make will not be passed to the element change listeners until we specifically request it via `reconcile()`.

9. Iterate through each method declaration and get its opening block:

```
for (Iterator iterator = methodDeclarations.iterator();
    iterator.hasNext();) {
  MethodDeclaration methodDeclaration =
     (MethodDeclaration) iterator.next();
  Block block = methodDeclaration.getBody();
```


The method declaration's block knows precisely where the opening brace of the method is, but it is possible that this method declaration is not the introduction of a method at all, but instead part of an interface declaration. So test to verify that the method declaration has a block, and if so, start to build up the final result in a `StringBuffer`:

```
if (block != null) {
  int i = block.getStartPosition();
  if (i >= 0) {
      StringBuffer sb = new StringBuffer();
    sb.append("\n\t\tSystem.out.println(\"Start ");
    sb.append(methodDeclaration.getName().getIdentifier());
    sb.append("\"");
```

This will output "`Start` *`methodName`*". There will be quite a few quotes-within-quotes, tabs, returns, etc. in this part of the code, and they are easy to miss. You may want to copy portions of this code from the jpage to save some typing and puzzling about which quote is missing.

10. The method name is done, now for the parameters. We won't get fancy, just the default `toString()` method for each parameter:

```
if (!methodDeclaration.parameters().isEmpty()) {
  for (Iterator parmIterator =
      methodDeclaration.parameters().iterator();
          parmIterator.hasNext(); ) {
    sb.append("\n\t\t\t + \" <\" + ");
        SingleVariableDeclaration parm =
        (SingleVariableDeclaration) parmIterator.next();
    sb.append(parm.getName().getIdentifier() + " + \">\"");
  }
}
```

Insert this code, it directly follows that in the prior step.

Finally, close the end of the trace statement:

```
sb.append(");");
```

To then insert it into the buffer, use the `replace()` method.

```
buffer.replace(i+1, 0, sb.toString());
```

Note that we are processing the method declarations in reverse order (bottom to top) to avoid the situation where inserting something into the buffer invalidates all the AST's calculated source positions before the insertion point.

In other words, if we insert something at position 100 of length 10, then all the positions calculated in the AST referring to said buffer beyond that point are now off by -10. We could compensate, but it is easier to work backwards and not worry about it.

Design Note: If you later code more complex modifications, ones that require multiple passes, then clearly this trick won't work. In that case, you could either double-buffer to another `StringBuffer` and copy back to the `IBuffer` when finished, or compensate as you go along. The latter would likely require a modification framework, but that's beyond the scope of this exercise.

11. Close the two `for()` loops. At this point, we have inserted all the trace statements into the buffer and are ready to let the others know. Add the invocation below after the loops:

```
synchronized (cu) {
   cu.reconcile();
}
```

12. Now save your method. Did you see any compiler errors? You should see the compiler flag an uncaught exception, `JavaModelException`, thrown by a few of the methods we invoked. Wrap a catch block around the code, as shown here:

```
try {
  … code written in prior steps …
} catch (JavaModelException e) {
  e.printStackTrace(System.out);
}
```

You can type this yourself, or select the block of code and use the editor's **Surround with try/catch** pop-up menu choice.

Step 3: Test

Now you are ready to save and test. Try the same tests that you did for the solution and verify that you see the same behavior (or better ☺).

Extra Credit:

- Detect if the code has already inserted a trace statement beforehand and skip that method if one is found.
- Detect those trace statements that have been modified and offer the user the choice of replacing them with the default or skipping them.
- Trace statements added to a constructor or method in a subclass that calls is parent class using the super() method results in a compilation error because super() must be the first statement, not the trace statement. Resolve that.

Part III: Generating Code Metrics while Traversing an AST (Optional)

Let's build on what we learned in the previous exercise while producing a more interesting user interface:

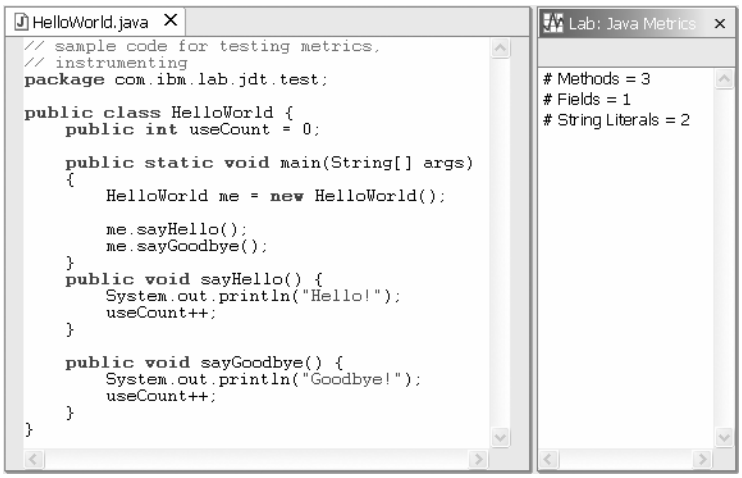


Figure 15-2
Java Metrics Example View [jdt_2.tif]

Here we have a view that displays code metrics for the selected *.java file (an instance of `ICompilationUnit`). The Java Metrics view on the right includes the number of methods, defined fields, and string literals. We could certainly add more metrics (e.g., hierarchy depth, number of imports, etc.), but the focus is on integrating with the JDT user interface and better familiarizing ourselves with ASTs, so let's keep it simple.

This exercise has several classes involved, so it pays to first look at the overall design of the solution we'll be producing. First, the classes and their basic responsibilities:

`JavaMetrics` – encapsulates the Java metrics for a given `ICompilationUnit`. The Java Metrics view creates a new JavaMetrics instance as its model when it opens. The JavaMetrics instance listens for changes to its compilation unit, updates the calculated metrics appropriately, and notifies interested parties.

`JavaMetricsAccumulator` – an inner class of JavaMetrics, this `ASTVisitor` subclass helps drive the metrics collection activity.

`IJavaMetricsListener` – defines the interface between a `JavaMetrics` instance and those, like `JavaMetricsView`, who wish to be notified of its state changes.

`JavaMetricsView` – view that displays the results of the `JavaMetrics` instance in simple text format. Listens for changes in its model and updates its display appropriately. Also listens for changes in the Workbench selection in order to set its model's `ICompilationUnit`.

You'll find that this exercise contains a fair amount of code that you have already seen in prior exercises in one form or another. So the exercise commentary will focus on how to integrate with the JDT model along with the help of an AST to produce a realistic and useful view.

Mini-Review of Workbench Views

Before we begin, a mini-review of some of the key characteristics of a Workbench view.

A perspective will only display one instance of a given view type (Outline, Navigator, Properties, etc.). A view can derive their input from different sources.

We have already seen a number of views that derive their input from "well-known sources." These sources are typically singletons that represent the "root" of a model. The

Plug-in Registry, Navigator, and CVS Repositories views are just a few examples you may have seen so far.

Another kind of view displays additional information about the currently selected object. A Properties view is such an example. Our view will work much the same, displaying the code metrics for the selected `ICompilationUnit`, or a message if the current selection is not a *.java file.

Step 1: Complete JavaMetrics class

Instances of the `JavaMetrics` class are not dependent on the user interface, but only their underlying model, an instance of `ICompilationUnit`. In a sense, this class is a "model of a model," the source model being the instance of `ICompilationUnit`. Keeping this in mind, let's start coding.

1. Note `JavaMetrics`' instance variables `methodDeclarationCount`, `fieldDeclarationCount`, and `stringLiteralCount`, corresponding to our desired code metrics. We have also defined a field for our underlying model, `ICompilationUnit`, and an array for listeners to the class' state changes:

```
private int methodDeclarationCount;
private int fieldDeclarationCount;
private int stringLiteralCount;
private ICompilationUnit cu;
private List listeners = new ArrayList();
```

   Note that while in our example `JavaMetrics` will only have one listener (the Java metrics view), we'll allow for more than one listener by declaring `List listeners` should we later decide to allow for more than one view on the same model.

   Public accessors (`getXXX`) for the three above metrics are already defined, but no setters. Remember, these values are only set when accepting a new compilation unit, or when the compilation unit changes, so there is no need for public setters. You can define private setters if you wish; this is more a matter of coding style in this case. If you choose to do so, the **Refactor > Self Encapsulate Field…** wizard can generate the getters/setters for you.

2. Take a brief look at the `IJavaMetricsListener` interface. The pattern should look rather familiar by now: a notification method that is called against registered listeners, in this case, called `refresh(JavaMetrics)`.
3. `JavaMetricsView` will use listener APIs to register interest with its model, an instance of `JavaMetrics`, so as to receive updates. These methods are fairly standard, except for the two lines of code that you will add to the `JavaMetrics` class, shown in bold below:

```
public void addListener(IJavaMetricsListener listener) {
  listeners.add(listener);
  JavaCore.addElementChangedListener(this);
}

public void removeListener(IJavaMetricsListener listener) {
  listeners.remove(listener);
  JavaCore.removeElementChangedListener(this);
}
```



```
private void notifyListeners() {
  for (Iterator iterator = listeners.iterator();
      iterator.hasNext();) {
    IJavaMetricsListener listener =
      (IJavaMetricsListener) iterator.next();
    listener.refresh(this);
  }
}
```

If you save now, you will see that a few errors were detected. That's because we added a call to register `JavaMetrics` as an element changed listener in `addListener/removeListener(…)` above, but haven't yet defined the `implements IElementChangedListener` clause or the `elementChanged(…)` method to handle it. Add the clause and skeletal method to resolve the errors; we'll complete these methods in a moment. Note that you can use context assist in the class body after adding the implements clause to create a skeletal method corresponding to the new interface or you can select **Override Methods…** from the Hierarchy view, as before.

By registering with `JavaCore's IElementChangedListener` interface, `JavaMetrics` will be notified if its `ICompilationUnit` is modified. It can then in turn notify its listener, `JavaMetricsView`. We're adding these invocations in the same methods as our own model's add/remove listener methods since we will only be interested in Java model changes as long as someone else is interested in our own metrics updates. To put this another way, the `JavaMetrics` model doesn't bother updating itself if there are no views (listeners) displaying its results.

Step 2: Create JavaMetricsAccumulator inner class

To process an AST, you need a subclass of `ASTVisitor`. We could have `JavaMetrics` subclass from `ASTVisitor` and process the desired `ASTNode` subclasses directly. Perhaps that would be expedient, but would clutter our metrics model with a number of `visitXXX(..)` methods that have more to do with the mechanics of AST processing than metrics gathering. Instead, let's create an inner class, `JavaMetricsAccumulator`, to handle the AST processing.

1. Define the `JavaMetricsAccumulator` inner class as a private subclass of `ASTVisitor`. You may need to add the appropriate `import` statements so the superclass is visible. Include a field for the visitor's client, `JavaMetrics`:

```
    private class JavaMetricsAccumulator extends ASTVisitor {
        private JavaMetrics jm;
    }
```

2. Our `ASTVisitor` subclass handles the mechanics of traversing an AST, and our `JavaMetrics` class handles calculating the metrics. To keep this separation of responsibilities clear, our visitor defers the processing to private methods of its client:

```
public boolean visit(StringLiteral node) {
  return jm.processStringLiteral(node);
}

public boolean visit(FieldDeclaration node) {
```



```
    return jm.processFieldDeclaration(node);
}

public boolean visit(MethodDeclaration node) {
  return jm.processMethodDeclaration(node);
}
```

Add these methods to the visitor class, and ignore the compile errors about the undeclared method invocations for the moment.

3. To start a visit, create a constructor method that accepts an instance of our client, `JavaMetrics`, and the compilation unit whose metrics it represents:

```
public JavaMetricsAccumulator
    (JavaMetrics jm, ICompilationUnit cu) {
  this.jm = jm;
  AST.parseCompilationUnit(cu, false).accept(this);
}
```

We have seen a variation of the `AST.parseCompilationUnit(…)` method before in `HelloAST`. In the prior case, it accepted a source code string. Here it accepts an instance of ICompilationUnit (a subtype of `IResource`) for analysis and returns the root `ASTNode`, an instance of `CompilationUnit`.

Remember: This is not an instance of type ***`ICompilationUnit`*** (which is part of the JDT model), rather it is part of the AST parse tree (i.e., models of the Java language elements and nothing more). There is another JDT class named `CompilationUnit` that implements `ICompilationUnit`, but it is in an internal package and we have no need to reference it.

The boolean parameter in `parseCompilationUnit(…)` specifies whether it should generate the map necessary to enable variable identifier-to-type information binding. If this parameter is `true`, then those `ASTNode` subclasses implementing `resolveBinding()` can return a description of the type associated with their identifier as an subtype of `IBinding` (e.g., `ITypeBinding`, `IBinding`, `IPackageBinding`, and `IVariableBinding` corresponding to `ASTNode` subclasses `AnonymousClassDeclaration`, `ImportDeclaration`, `PackageDeclaration`, and `VariableDeclaration` respectively). Calculating this binding takes additional time, and we don't need it in our case. The `CompilationUnit.accept(…)` method starts the visit, passing our visitor instance.

Step 3: Complete JavaMetrics ASTNode processing, notifications

We are almost finished with our metrics model.

1. Our metrics are quite trivial, we only increment a counter as we find the corresponding node type. Recall that the return value of the type-specific `visit(XXX)` methods in `ASTVisitor` indicates if you wish to continue to the child nodes. You can use this to optimize your traversal, e.g., to only look at methods and not the method's implementation (child nodes) within. Our visitor used the type-specific methods, and will defer the "visit child nodes or not" decision to the metrics generation code rather than including it in the visitor code:

```
protected boolean processStringLiteral(StringLiteral node) {
  stringLiteralCount++;
  return false;
```



```
}

protected boolean processFieldDeclaration
    (FieldDeclaration node) {
  fieldDeclarationCount++;
  return false;
}

protected boolean processMethodDeclaration
    (MethodDeclaration node) {
  methodDeclarationCount++;
  return true;
}
```

Add these methods to `JavaMetrics` (**not** its inner class), and then carefully study them for a minute or two. Will all string literals that appear in the sample code be counted? If not, which will be excluded and why? Hint: Where else can string literals appear besides methods?

2. This model has inbound notifications (from `JavaCore`) and outbound notifications (to `JavaMetricsView`). Let's handle the inbound notifications first. The Java element change notification is much the same as a Resource change event notification, that is, it is a delta potentially describing more than one change. We will need to discern whether the change affects our underlying compilation unit. For the moment, let's code up a skeletal method and get to the details later:

```
public void elementChanged(ElementChangedEvent event) {
  if (cu != null) {
    ICompilationUnit cu2 = (ICompilationUnit)
        findJavaElementDelta(event.getDelta(), cu);
    if (cu2 != null)
      reset(cu2);
  }
}
```

The first part of the `if` condition indicates that there is no need to update (reset) if the `JavaMetrics'` model is null. This will be the case when the `JavaMetricsView` is initially created, since the newly created `JavaMetric` instance does not yet have a compilation unit from which to calculate metrics. On the other hand, if the metrics model has a compilation unit, the model must determine if the `IJavaElementDelta` returned by `event.getDelta()` references it. This is necessary because change notifications are sent for all `IJavaElements` in the workspace to all element change listeners, and our metrics model is only interested in its own.

Put a breakpoint on this method for later investigation. We'll use the debugger to see that the `ElementChangedEvent.getDelta()` method returns an array of changed elements, starting with the project, folder, and finally compilation unit.

Views like the Navigator, which display a complete hierarchy, are interested in each element in the tree, but again, the `JavaMetrics` instance is only interested if its compilation unit (or one of its child elments) has changed. This is why the `findJavaElementDelta(…)` method recursively searches for a specific element. If it is among the changed elements, it recalculates the metrics, otherwise the change is ignored.

3. Briefly note how the `findJavaElementDelta(…)` method below recursively searches the provided structure, and that it checks if the java element is a "working copy." We'll return to this point in Part III of this exercise. For the moment, suffice it to say that the JDT defines the notion of a staging area for uncommitted (unsaved) changes to an `ICompilationUnit` called a "working copy." A working copy has the same interface as its original element, but changes to it do not effect the Java model until they are explicitly committed. Working copies are not attached to a resource, hence why the code below must retrieve the original element in order that a meaningful comparison between the changed element and the JavaMetrics' `ICompilationUnit` instance can be done.

   Also note that `IJavaElement` overrides `Object`'s definition of `equals()` in order to qualify it logically rather than by strict object identity. It looks for whether the two elements have the same name, parent, etc.

```
private IJavaElementDelta findJavaElementDelta(
    IJavaElementDelta parentJed,
    IJavaElement je) {

  IJavaElementDelta jed = parentJed;
  IJavaElement je2 = parentJed.getElement();

  if (je2 instanceof IWorkingCopy
      && ((IWorkingCopy) je2).isWorkingCopy())
    je2 = ((IWorkingCopy) je2).getOriginalElement();

  if (je.equals(je2)) {
    return parentJed;
  } else {
    for (int i = 0; i < parentJed.getAffectedChildren().length; i++) {
      jed = findJavaElementDelta
        (parentJed.getAffectedChildren()[i], je);
      if (jed != null)
        return jed;
   }
 }

 return null;
}
```

   Open the `IJavaElementDelta` interface. You can do this by selecting its name in the code above and pressing F3. This interface includes more detail about the nature of the change than we will explore during this lab. But looking at the interface, can you describe the difference between the `getAffectedChildren()` and `getChangedChildren()` methods?

4. Complete the `reset` method by starting a new visit (see code in bold). This method will accept a new compilation unit and update the metrics, or null where it simply resets them to zero. In both cases, the listeners are notified so they can update themselves accordingly.

```
    public void reset(ICompilationUnit cu) {
        this.cu = cu;

        methodDeclarationCount = 0;
        fieldDeclarationCount = 0;
```



```
        stringLiteralCount = 0;

        if (cu != null)
            new JavaMetricsAccumulator(this, cu);

        notifyListeners();
    }
```

Remember to save and correct any compiler errors before continuing.

Step 4: Complete JavaMetricsView

This step will seem quite familiar, since it follows the same pattern as the other view exercises. This view is intentionally unsophisticated in appearance – it only has a single text widget for displaying its results. But it demonstrates how to synchronize the Workbench selection with the currently displayed results, and how to keep displayed results synchronized with the underlying Java model.

1. The `createPartControl(…)` method is partially complete. It begins by creating the text widget to display its results. To get selection notifications when the user moves the focus, add the code below:

```
    getViewSite().
        getWorkbenchWindow().
        getSelectionService().
        addSelectionListener(this);
```

This is only one line of code, but it traverses several object relationships that are worth understanding. Follow the method invocations using the background information below:

- Get the view part's view site. View site is an interface between view parts and the rest of the GUI framework (access to the workbench window, action bar, shell, decorator manager, etc.).
- Ask the Workbench window for its selection service. The selection service tracks the current selection, maintains a list selection listeners, and notifies them when the selection changes among ISelectionProviders.
- Add our view (part) to the list of selection listeners. Our view part will now get notified when the selection changes.

You may remember the selection listener interface from prior exercises. In this case, we want to know if the selection changes to an `ICompilationUnit` within those views in the same Workbench window as our view. A selection can either be "text only" (`ITextSelection`), or "structured" (`IStructuredSelection`, like the entries in a listbox, tree view, etc.). We are only interested in selections from the Navigator, Outline, etc., all of whom are `ISelectionProvider` implementors providing `IStructuredSelection` results.

We've already created the viewer and view part and established the view/viewer relationship. Now let's create our model and store a reference to it, the `JavaMetricsView`:

```
    jm = new JavaMetrics();
    jm.addListener(this);
```

Save the class before continuing.

2. Now let's finish the selection listener method, `selectionChanged(…)`. When this method is called indicating that an `ICompilationUnit` was selected, we want to update the `JavaMetrics` instance and subsequently the view:

```
if (selection instanceof IStructuredSelection) {
  ICompilationUnit cu =
    getCompilationUnit((IStructuredSelection) selection);
  jm.reset(cu);
}
```

The `getCompilationUnit(…)` extracts the `ICompilationUnit` from the selection, if possible, or returns null. We code it in the next step.

3. The `getCompilationUnit(…)` method handles those cases where a sub-element of a compilation unit is selected. For example, if a method is selected in the Outline, Packages, or Hierarchy view, we'll treat it as if its containing compilation unit itself was selected. It also handles the case where the selected element is the source file itself (*.java) from a view like the Navigator, where it will be an instance of `IFile`, not `ICompilationUnit`. The method is already coded, here it is for reference:

```
private ICompilationUnit
    getCompilationUnit(IStructuredSelection ss) {

  if (ss.getFirstElement() instanceof IJavaElement) {
    IJavaElement je = (IJavaElement) ss.getFirstElement();
    return (ICompilationUnit)
        je.getAncestor(IJavaElement.COMPILATION_UNIT);
  }
  if (ss.getFirstElement() instanceof IFile) {
    IFile f = (IFile) ss.getFirstElement();
    if (f.getFileExtension() != null &&
        f.getFileExtension().compareToIgnoreCase("java") == 0)
      return (ICompilationUnit) JavaCore.create(f);
  }

  return null;
}
```

In a nutshell, this method will either return null (because the selection is not a `IJavaElement` or a *.java file), or will walk up the Java model object hierarchy looking for the parent compilation unit, if one exists, starting from the new selection.

4. The last case to consider is when the compilation itself changes. As you recall, the `JavaMetric` instance registers with `JavaCore`'s element change listeners. When the `JavaMetric`'s compilation unit changes, it in turn notifies its listeners by calling the listener's `refresh(JavaMetrics)` method. Our view will respond by setting the value of the text field, or if the `JavaMetrics` is not valid (for example, because it does not yet have a compilation unit or the compilation unit has been deleted), it sets the text to a "no metrics available" message to alert the user. Add this code to the `refresh(…)` method:

```
if (jm.hasValidMetrics())
  message.setText(jm.summaryString());
```



```
else
  message.setText(NO_SELECTION_MESSAGE);
```

Since the notifications can come from a non-UI thread, the above code will have to be enclosed in an `Runnable.run` method in order to execute it in the UI thread using `Display.getDefault().syncExec(Runnable)`. That part of the code isn't shown here, but is in the template code and solution. Once you make this change, you'll see a compiler error about referencing a non-final variable (the `jm` parameter in the `refresh(JavaMetrics jm)` method). Change the method parameter to '`unused`', since we want to refer to the `jm` instance variable anyway.

Now we have finished all the coding, only one more step to go!

Step 5: Review plugin.xml and Test

Let's turn to the plugin.xml file that wires our view into the Workbench. This should seem like old hat by now, so we won't bother you with too many details.

1. Take a look at plugin.xml, here's the relevant extract below:

```
<extension
    point="org.eclipse.ui.views">
  <category
       name="Edu-Sol"
       id="com.ibm.lab.view.category">
  </category>
  <view
    name="Lab: Java Metrics"
    icon="metrics.gif"
    category="com.ibm.lab.view.category"
    class="com.ibm.lab.jdt.JavaMetricsView"
    id="com.ibm.lab.jdt.javametricsview">
  </view>
</extension>
```

The category defined is the same as is used by other exercies. You could also decide to place the view into the existing Java category. To do this you would use a category value of `org.eclipse.jdt.ui.java` in the view definition. If you were actually building a tool that was designed to complement the JDT, choosing to use the existing Java category might give the user interface a stronger feeling of integration.

2. Now we're ready to test. Select the plugin.xml and start the Workbench run-time instance using the debugger. You will need some test data, we've provided you with a HelloWorld.java source file in the `com.ibm.lab.jdt` project that you can either import directly into your run-time workspace, or create a test project, package, and class, then paste in the code. Once you have created this class, open the **Edu > Lab: Java Metrics** view and select the HelloWorld.java source file in the Packages view. Do your metrics update?

   If you followed the steps above precisely, you may remember that you set a breakpoint in `elementChanged(ElementChangedEvent)`. If you save a modification of your test Java source once the Java Metrics view has been displayed, you should stop at this breakpoint. Open the Details pane of the Variable view in the debugger and select the JavaElementDelta. You will see a display similar to this:

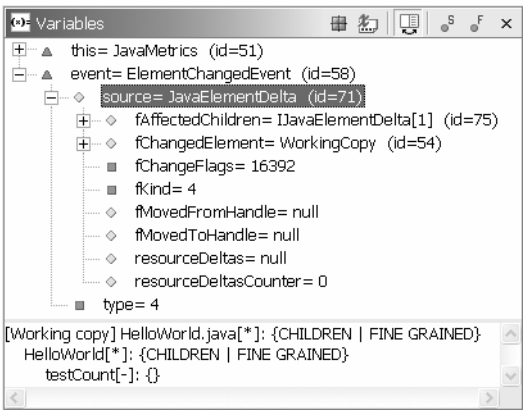


Figure 15-3
Variable Debug View [jdt_3.tif]

Another very helpful example of `toString()` output displayed in the Details pane. In this case, it is recursively displaying the result of `getAffectedChildren()`, giving you a clear view of what Java model elements are affected by the updates.

Exercise Activity Review

In this exercise you should have learned:

- How to analyze Java source code using ASTs and ASTVisitor
- How to extend the JDT UI by adding new actions to an editor
- How to extend the JDT UI by adding a new view

(Optional) Exercise 16 Using Eclipse

Introduction

The objective of this lab is to provide you with a hands-on introduction to using Eclipse, including creating and using resources, and manipulating the user interface.

Note: This material has been tested on 2.1, but screen images still reflect the 2.0 user interface.

Skill Development Goals

At the end of this lab, you should be able to:

- Be familiar with the basic structure of Eclipse
- Navigate the Eclipse user interface
- Move and resize views
- Compare and replace projects and editions of resources
- Recover deleted files and projects
- Customize a perspective

The time available during a formal class may not let you complete the entire exercise. Consider this subset first:

- Part 1 – all
- Part 2 – 1 - 10
- Part 3 – 1 – 5
- Part 4 – all

If you finish you can return to do the steps you skipped.

Exercise Instructions

Part 1: Your First Eclipse Project

Let's get started. In this part, we'll explore the Eclipse user interface and create and import projects, files, and folders.

There is no set up for these exercises, nor sample code or files. You just need to have Eclipse installed. We'll create what we need as we go.

1. In your Eclipse directory, invoke the `eclipse.exe`.

Figure 16-1 below shows the Eclipse user interface as it comes, out of the box. Eclipse opens the Resource Perspective.

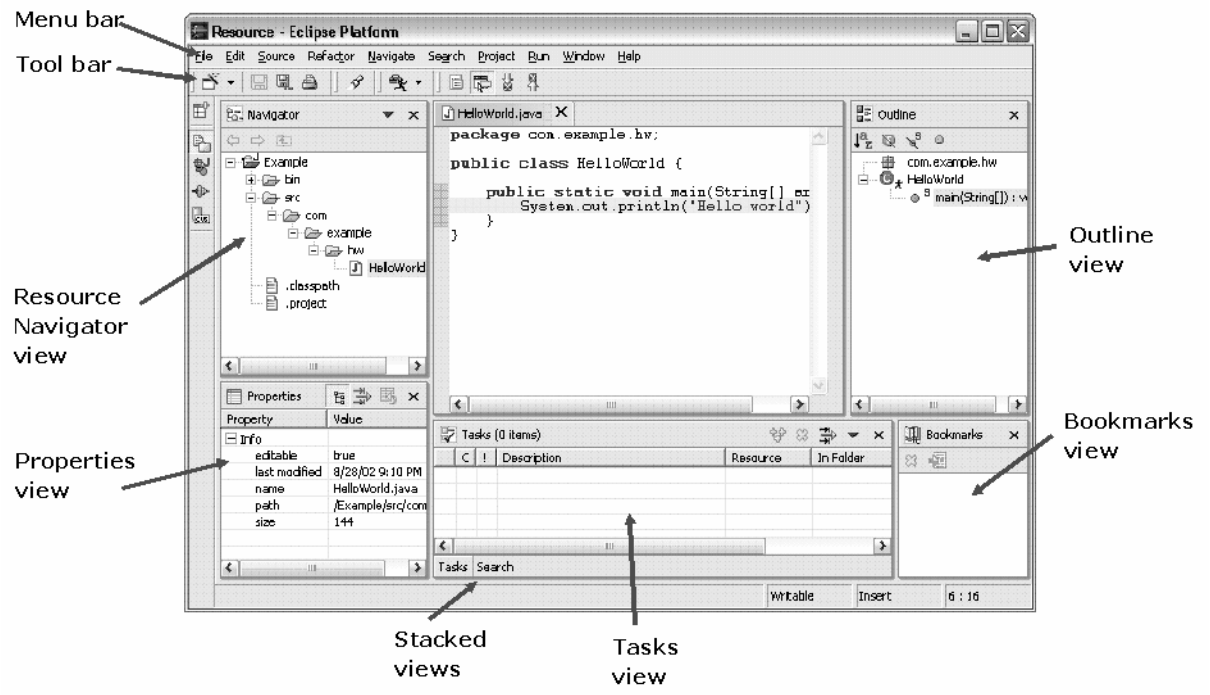


Figure 16-1
Resource Perspective

The Eclipse user interface is presented as an adjustable multi-paned window comprising the following.

Menu bar. All user interface actions are available from menu bar items. If you were an Eclipse 1.0 user, you'll notice a significant reorganization of the menu bar items.

Tool bar. This is a set of icons that represent shortcuts to some Eclipse actions that also appear as menu items.

Editors. Editors are stacked in the middle pane, with tabs for navigation. When Eclipse first comes up with the Resource perspective, the welcome page is

shown. This is an XML document that lists common actions. Welcome pages can contain links to user interface actions along with the text descriptions. Try selecting one of the links. You can resize the editor pane by grabbing and moving one of the interior edges or the pane.

Views. Views are organized by the perspective in the window. The Resource perspective includes the Navigator view for navigating through and operating on resources, the Outline view for navigating within a resource (one that supports an outline view, such as Java source code), and the Task view for listing things to do. In Figure 16-1, we added the Bookmarks and Properties views. Views can be resized and stacked.

Shortcut Area. This is the wide left border of the main window. Initially, only your open perspectives are shown. When you have multiple perspectives open, you can navigate through them by selecting the icons. You can also minimize views to an icon on the shortcut bar to free up desktop real estate. These are called fast views. We'll get to them in a moment.

OK, now that you have a basic understanding of the organization of the user interface, let's get out of first gear. To do this, we need some resources to work on. Let's create these now.

2. Create a project by selecting **File > New > Project…** You can also select the **New Wizard** drop-down and then **Project**. This opens the New Project wizard.

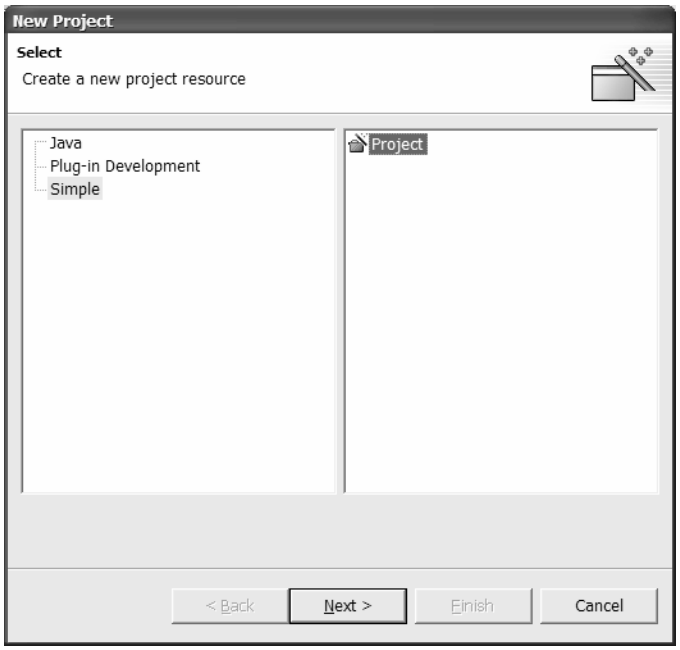


Figure 16-2
Creating a Project

3. Select **Simple** and then **Project.** Press **Next**.

4. Enter `My First Project` as the project name, leave the other defaults, and select **Finish**. You new project appears in the Navigator view. If you expand the project, you'll see that Eclipse has added file `.project`. This is a file Eclipse maintains to keep information about your project.
5. Select **File > New > Folder** or select **Open the New Wizard**  . When you use the New Wizard, if you select the button (verses the pulldown), you'll see the New wizard selection page. This same page can be opened using **Ctrl+N**. Select **Simple** and **Folder**. You can also press on the small down arrow to the right of the image to see a list of choices. Select **Folder**.

6. Select **My First Project**, enter **Text Files** as the folder name, and press **Finish**. In the Navigator view, you see the folder added to your project.

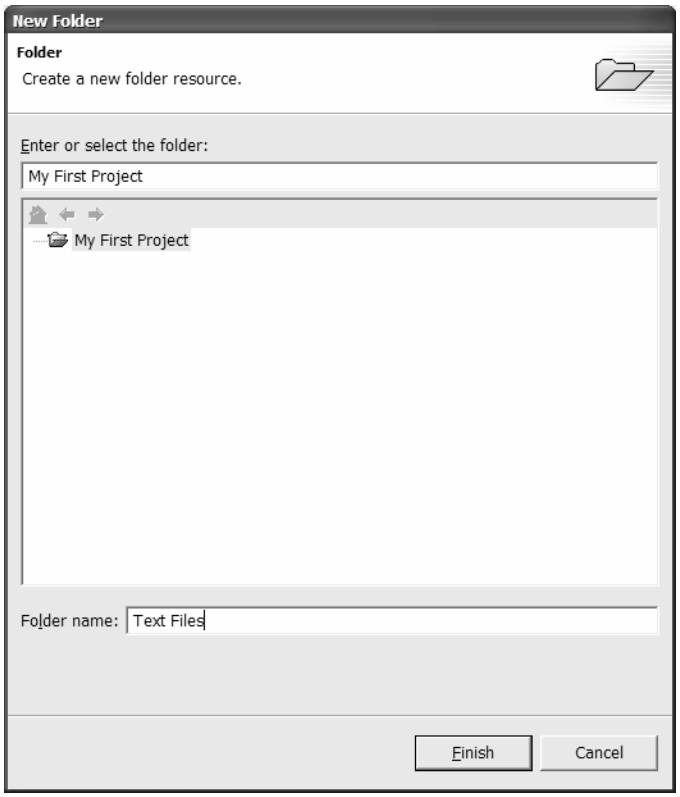


Figure 16-3
Adding a Folder

7. From the menu bar or the tool bar, use the **New** pull-down again to create a new file. Expand **My First Project** and select the **Text Files** folder. (If you select the **Text Files** folder before you open the wizard it will already be selected in the wizard.) Enter "`My First File`"as the file name and press **Finish**. In the Navigator view, observe the file added to your project. The file will also be opened in an editor.

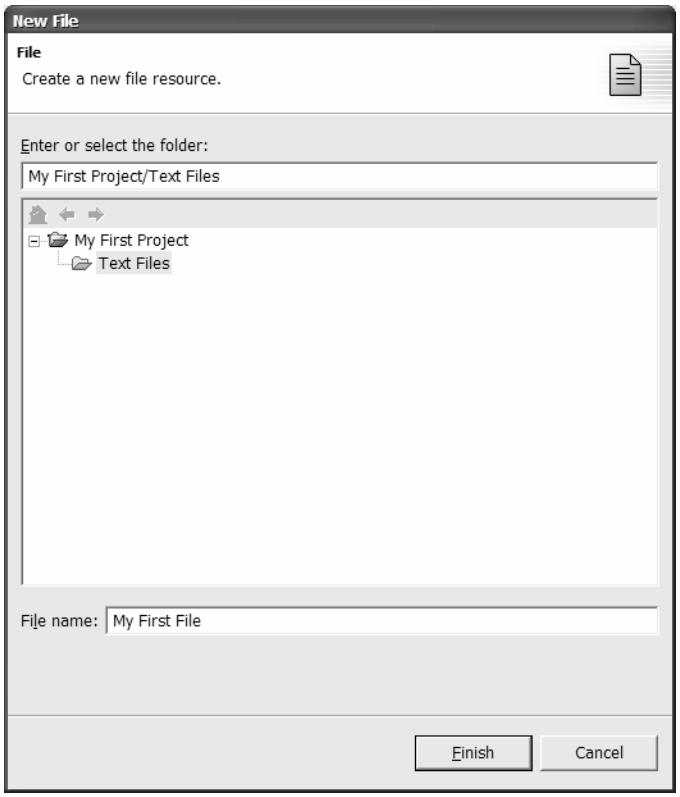


Figure 16-4
Creating a File

@since 2.1 – By default folders and files in the Project root are created in the project directory. With 2.1 you can use the **Advanced >>** button to link a file or folder to another file system location. A Path Variable may also be used to support the link. Try it if you want, add a linked file or folder in **My First Project**.

8. In the editor, add some text to the file. As soon as you start typing, an asterisk, "*", is added as a prefix to the file name on the editor tab. This is an indication that you have changed the contents of the file.
9. Create a second text file in the same folder called `My Second File`. Add some text to this file as well.
10. Create a second simple project called `My Second Project`.
11. The Navigator view should now appear as shown in Figure 16-5.

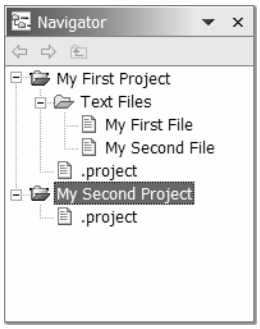


Figure 16-5
Project Organization in the Navigator View

12. We've created files. Now let's add some existing files to one of your projects. We're going to do this by importing files you have on your file system. Select **My Second Project** in the Navigator view and select **Import...** from the context menu. In the Import Wizard, select to import from the **File System** and then select **Next >**.
13. Select **Browse...** to browse your file system for a folder with some files to import (it really doesn't matter which files). In the Browse for Folder dialog, select a folder and then select **OK**.
14. Expand the folder in the left pane to see its sub folders (if the folder you selected previously has subfolders). If you select (check) a folder in the left pane, you will import the folder and all the folders and files it contains. If you give a folder focus you can select individual files in the right pane. If you expand the folder you can select from the subfolders and the files in each subfolder. Select a set of folders and or files. Select **Finish** to import the folders and/or files you selected.

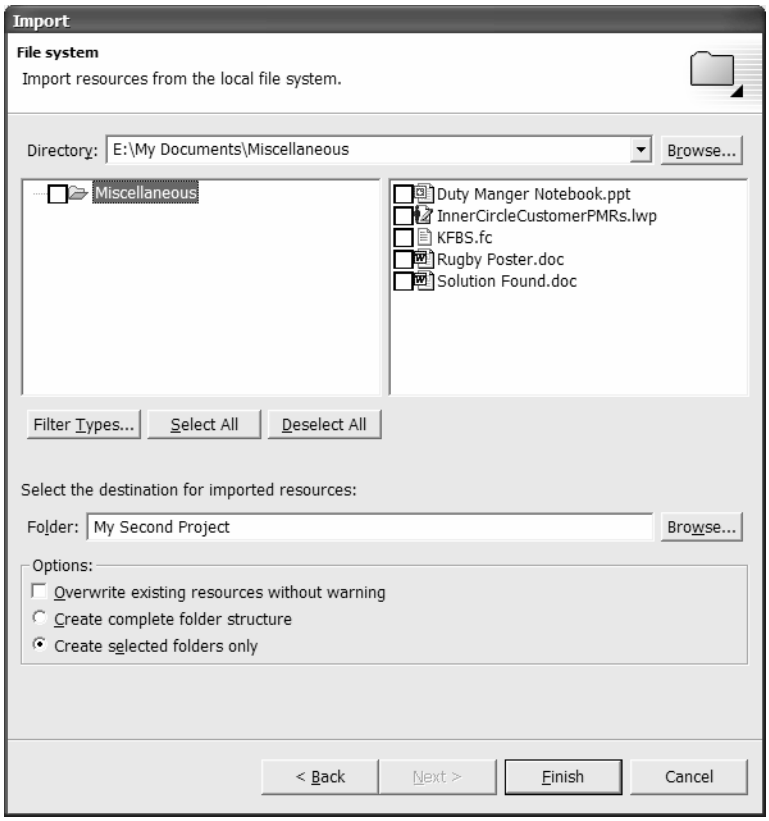


Figure 16-6
Importing Resources

You'll see the files you selected to import show up in the Navigator view

15. You can also drag and drop files to add them to a project. Open a file system dialog, for example Windows Explorer, drag a file from it and drop it on **My Second Project**.
16. If you are using 2.1, you can also add files and folders to the project using a resource link. Select **My Second Project** and then open the New wizard. Select File or Folder, and then select **Advanced >>** to expose the link portion of the new resource dialog.
17. Browse to locate the target File or Folder for the resource link. Note that the wizard will prevent you from creating a linked resource in a location other than the project itself.

    The Navigator will include linked resource decorators for linked resources. This is controlled using **Label Decorations** preference page (Use **Window > Preferences > Workbench > Label Decorations** to open the **Preferences** dialog).

    If you delete a linked resource, you delete the workspace reference to the resource. If you delete a file in a linked folder, you have deleted the file on the file system.

Part 2: Editors and Views

Let's take a look now at using editors and views. We'll see how to edit files, manipulate views and editors, and use tasks and bookmarks.

1. Select the tab on one of the editors you should already have open. On the bottom right margin of the window is the editor message area (see Figure 16-7).

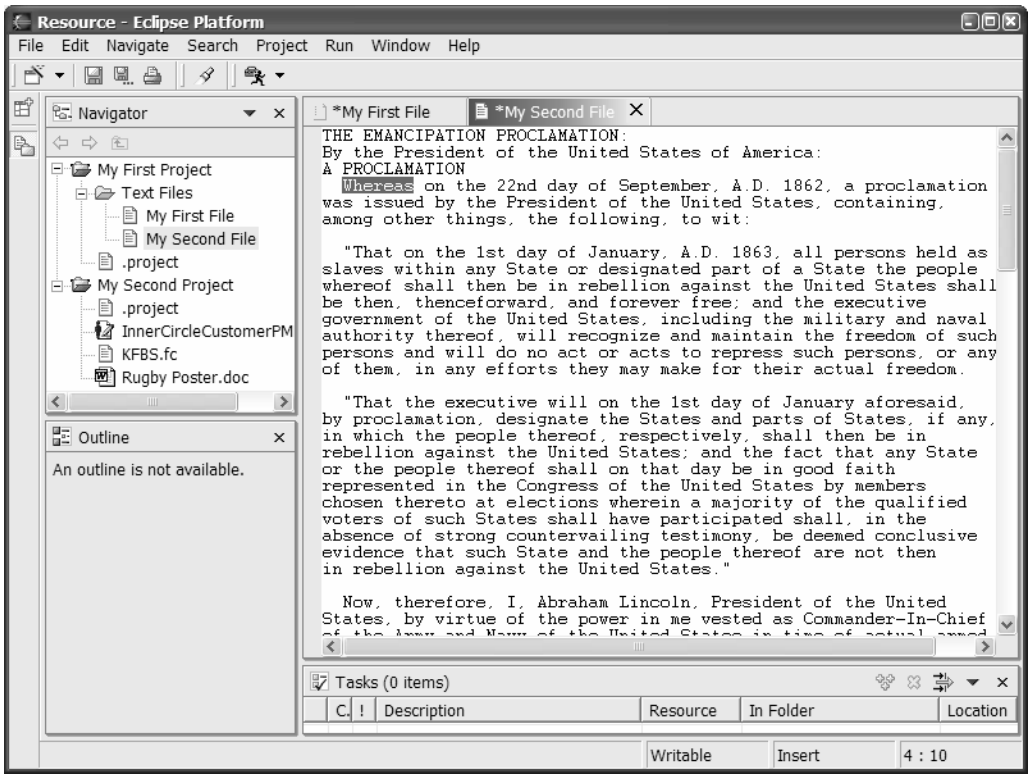


Figure 16-7
Editor Message Area

The editor message area contains three fields, File State, Typing Mode, and Position. These fields provide information about the current editor. The values you should see are **Writable**, **Insert**, and **n:n**. "Writable" means you can change the file. If the file property was read-only, you wold see "Read Only" in this field. "Insert" indicates the state of the "Insert" button on your keyboard. Pressing the Insert button will change this to "Overwrite" meaning your typing will overwrite text instead of inserting the text. The final field indicates the row and column of the cursor position.

2. Close **My Second File** by selecting on the '**X**' on the editor's tab. You can also do this by selecting **Close** from the context menu on the editor tab, or use the menu option **File > Close** (**Ctrl+F4**) . You can even close all open editors at once using **File > Close All** (**Ctrl+Shift+F4**).
3. Reopen **My Second File** by double clicking on it in the Navigator view. You can also select the file and select **Open** or **Open With >** from the Navigator context menu.

When you use the **Open With >** popup menu, the **System Editor** option refers to the operating system associated program for files of this type. You can alter the editor mapping and behavior using the Workbench > File Associations preference page.

4. There are several ways to navigate and manage your editors. Try selecting **My First File** and then **My Second File** in the Navigator view. You see that active editor is synchronized with your selection. The reverse is also true; selecting an open editor will put that file in focus in the Navigator view. In 2.0 this behavior is actually defined by the Workbench preference setting **Link Navigator to active selection**.

   @since 2.1 - In 2.1 the Navigator and Package Explorer views now include a **Link with Editor** toggle to control the synchronization with the active editor.

5. Double click on the blank area to the right of the editor tabs or on one of the file names on the tabs. You'll see that the editor expands to fill the entire window as shown in Figure 16-8. This is useful for serious editing where you need more screen real estate. Double click again to restore the editor to its original size.

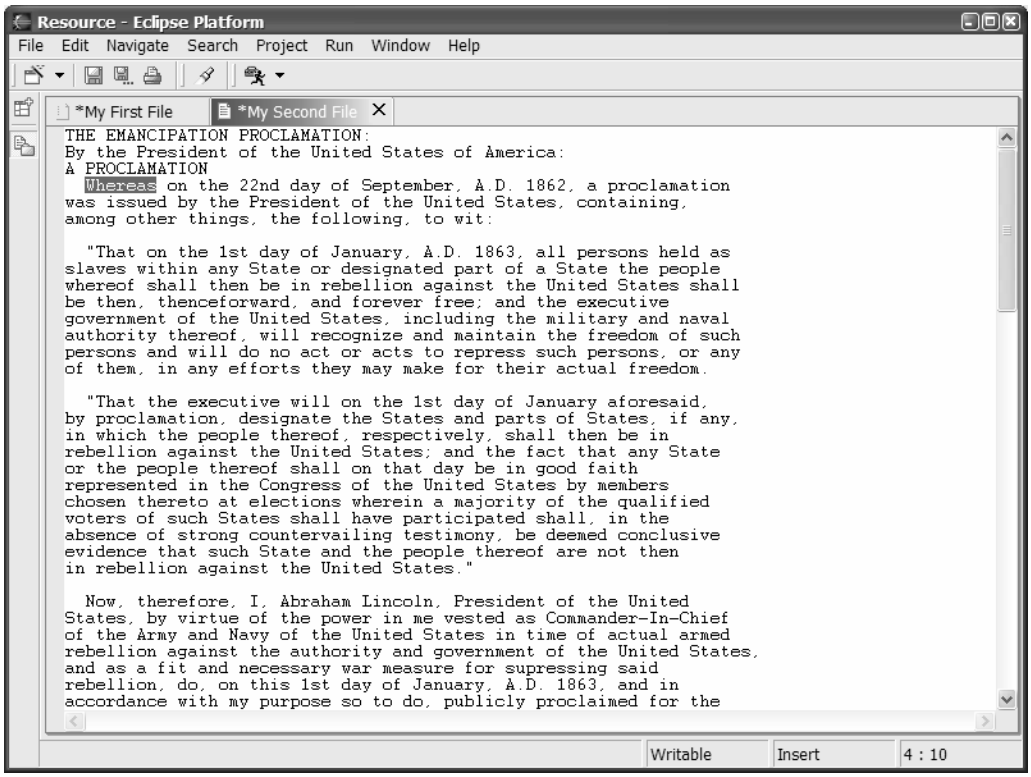


Figure 16-8
Expanding an Editor

6. You can change the ordering of the editors in the editor pane. Select the tab for **My Second File,** drag it in front of the tab for **My First File,** and release it. The order of the editors is changed, as you can see by the order of the tabs.
7. Resize the area the editors occupy by selecting and dragging the bottom or left border of the editor pane. You can also do this by selecting **Size** from the editor tab context menu, choosing a side to move, and using the arrow keys.

8. Editors open stacked one in front of another. You can change this organization by tiling one or more of the editors within the editor area. Create another file called **My Third File**. Select the tab of **My Third File** and drag it to the left border of the editor pane. When the cursor changes to a left arrow, drop the editor. Select the tab of **My Second File,** drag it to the bottom border of the editor pane below **My Third File**, and drop it. The result of this is shown in Figure 16-9.

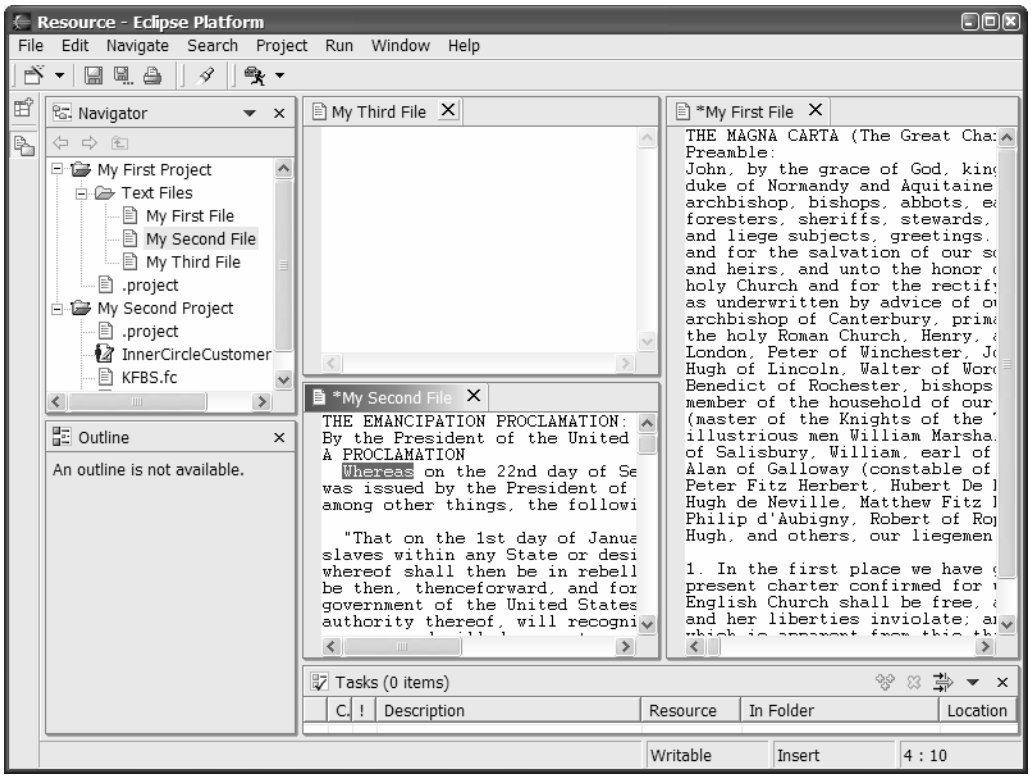


Figure 16-9
Reorganizing Editors

9. Of course, you can also re-stack the editors. Select the tab of **My First File** and drag it on top of the tab of **My Second File** or the area to the right of it. Select the tab of **My Third File** and drag it on top of on of the other two tabs or the area to the right of them. The editors should be stacked again. The order of the tabs may have changed from what it was before, depending on where exactly you dropped the editors.
10. Select anywhere in the Tasks view to make it active. Press **F12**. The editor for the resource you were most recently editing is now active. This is a quick way to get back to your most recent active editor.

Note: This is a good point to move on to the next part if you are just trying to experience a subset of each lab part.

11. You have keyboard shortcuts for navigating between editors. Press **Ctrl+F6** to go to the next editor. Press **Ctrl+Shift+F6** to go to the previous editor. While pressing **Ctrl**, you can press of **F6** repeatedly to move the current selection in the list of open editors.

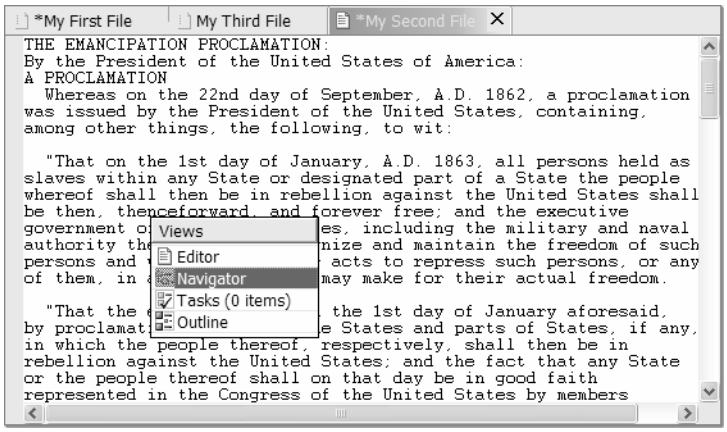


Figure 16-10
Editors List

@since 2.1 – In 2.1, the user interface includes Workbench actions in the form of editor navigation arrows (see Figure 16-11). These can be used to move backwards, forwards, and return to the last file you modified.

Figure 16-11
Editor Navigation

12. You've seen how to open, close, move and resize the editors. You can do the same with views. Select the '**X**' on the top right of the Task view to close the view. You can also select **Close** from the Task view context menu. Reopen it by selecting **Window > Show View > Tasks**. Not all views are listed in this short list, you may have to select **Other…**, a category, and then the view you actually want to open.
13. Like editors, views can be reorganized in the window. Select the title bar of the Navigator view and drag it on the title bar of the Outline view. These views are now stacked (see Figure 16-12). You can navigate between them with the tabs at the bottom.

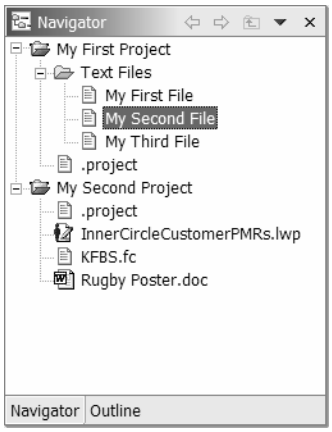


Figure 16-12
Stacked Views

14. Grab a view by its title bar and try dragging it around. Observe how the cursor changes. A folder icon indicates that the view will be stacked. An arrow icon indicates that the view will be placed to that side of the view you're over. You can do this with single views and a group of stacked views.
15. You can also place a view in the shortcut area. This is the wide border on the left side of the window with the icons for the open perspectives. This is a handy way of freeing up screen real estate. Select **Window > Show View > Bookmarks** to open the Bookmarks view. Grab it by its title bar, drag it over to the shortcut area so that you see the stacked image, and drop it there. You will see an icon for the view in the shortcut area under the icon representing the open Resource perspective (the one you're in now). You can also select **Fast View** from the view menu (click on the view icon on the title bar) to add the view to the shortcut area.
16. Select the icon for the Bookmarks view you just placed in the shortcut area. Watch as it slides to the right to become visible (see Figure 16-13).

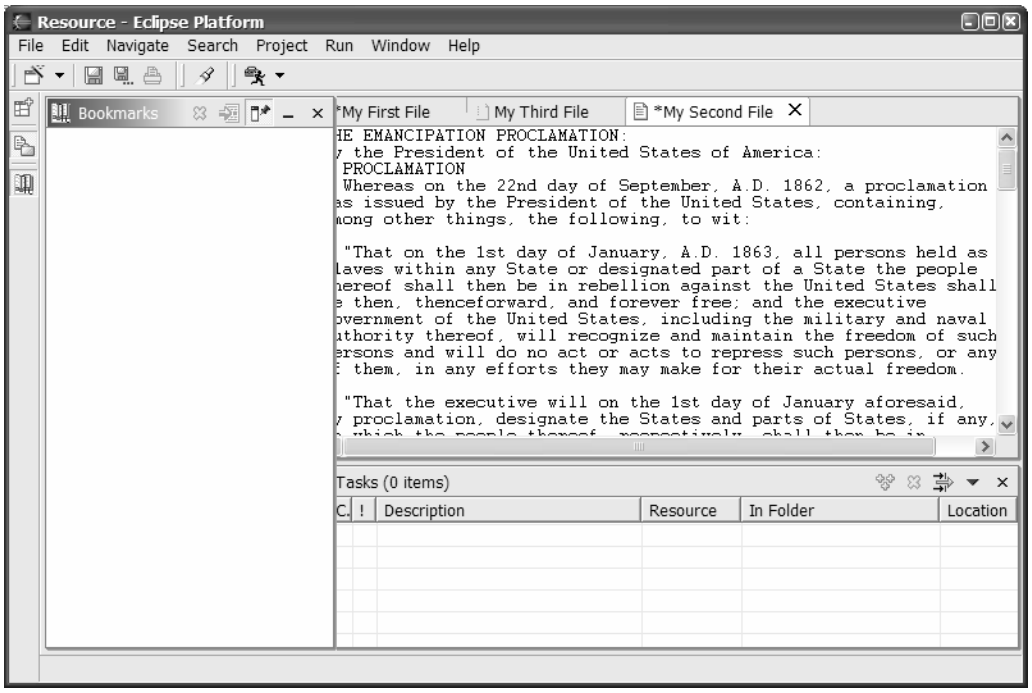


Figure 16-13
Expanding a View from a Shortcut

17. Select the icon again to see it slide back up to the left to become hidden. If you resize the view while it is being displayed, it will return to that size each time you make it visible as a fast view. You can restore the view to its original state by deselecting the **Fast View** toggle in the view.
18. At this point, you're probably wondering how to clean up the mess you've created of your user interface. Select **Window > Reset Perspective** to do so. The user interface returns to its original configuration.

19. As with the editors, you also have keyboard shortcuts for switching between views or the set of editors. The set of open editors is treated a view for purposes of navigation. Press **Ctrl+F7** to switch to the next view and **Ctrl+Shift+F7** to switch to the previous one. The list of views remains visible as long as you hold the keys down. Pressing **F7** repeatedly, while holding **Ctrl**,or **Ctrl+Shift**, will move the view selection.

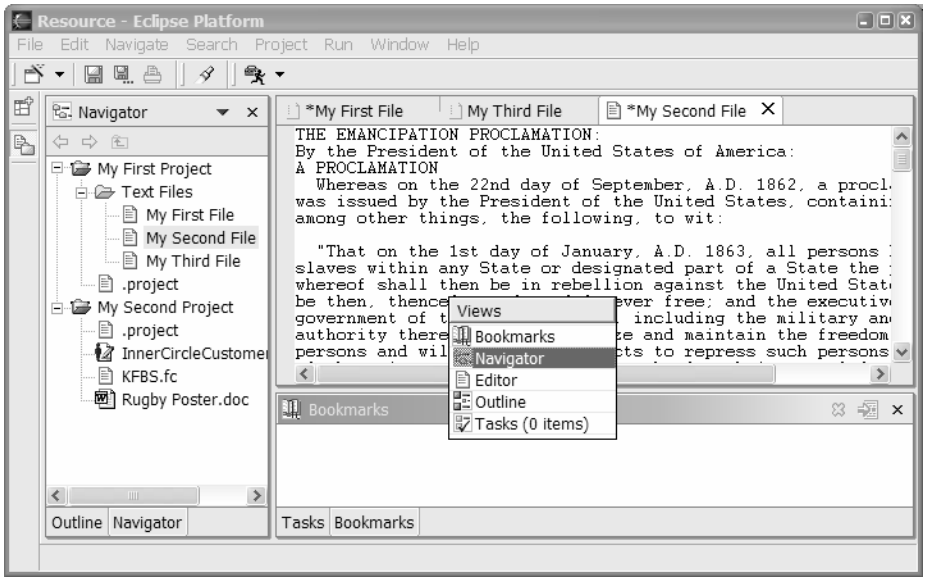


Figure 16-14
Views List

20. Bookmarks are for marking files, or specific locations in a file. Drag the Bookmarks view to stack it on top of the Tasks view. Select the open editor for **My First File**. In the marker area of the editor (the left margin) bring up the context menu and select **Add Bookmark…** Name the bookmark `Start here tomorrow` and select **OK**. You'll see the bookmark added in both the marker area of the editor and the Bookmarks view (see Figure 16-15). You can also add a bookmark to a file using the Navigator context.

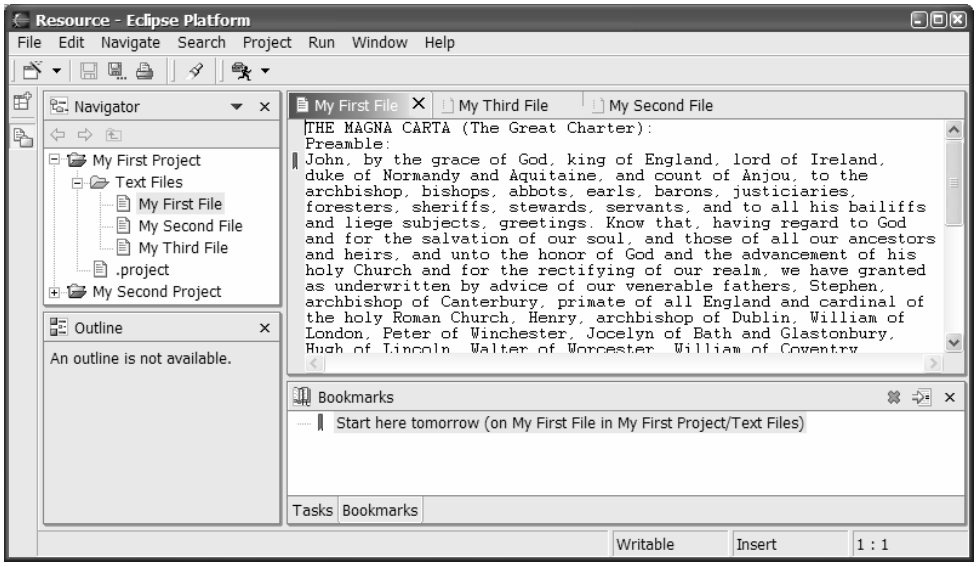


Figure 16-15
Creating a Bookmark

21.Close **My First File** and then double click on the bookmark you just added. The resource associated with the bookmark is opened and the line it refers to is selected. You can delete a bookmark using the context menu of the Bookmarks view or by selecting **Remove Bookmark** from the context menu of the bookmark in the editor marker area.

22.Tasks are for tracking specific work items. Like bookmarks, tasks can be associated with a resource and location in the resource, though they don't have to be. Unlike bookmarks, tasks have state and priority information. You create tasks in the same way you create bookmarks. Select the editor on **My Second File**. In the marker area, bring up the context menu and select **Add Task**… Enter text for the task, select a priority, and then select **OK** to create the task.

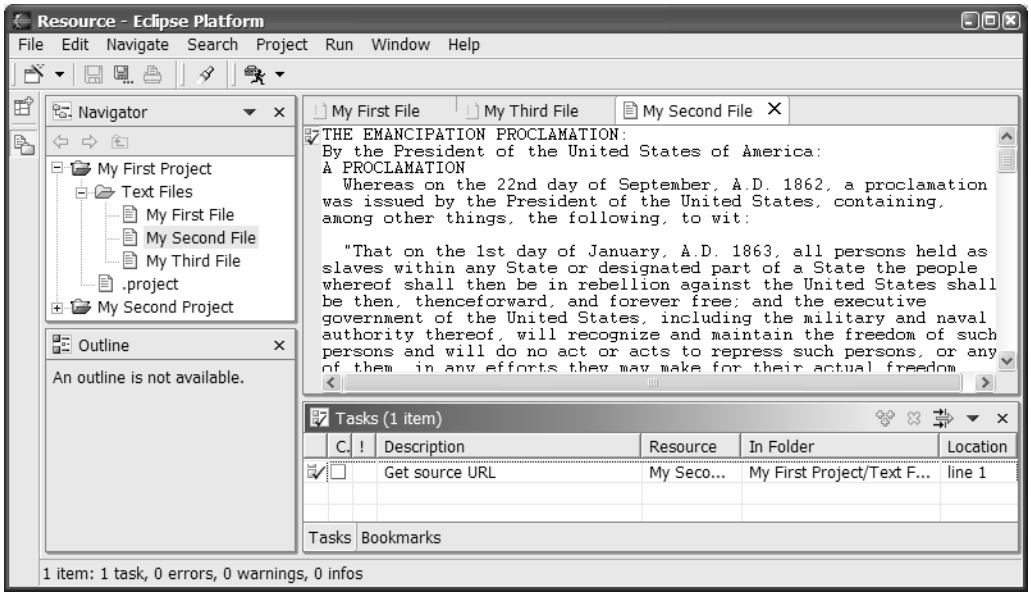


Figure 16-16
Creating a Task

23.Close **My Second File** and double click on the task you just created. As with bookmarks, an editor is opened on the file associated with the task and the line the task references is selected.

To create a task associated with no file, select **New Task** in the Tasks view.

24.Finally, let's take a look at the toolbars. You can reorganize the icons on the toolbar to reorder them or to put them on another toolbar line. Select one of the dividers between the groups of tool bar buttons and try moving the group on the same line or to another line. You can also compress a group of icons into a drop down menu. Select one of the toolbar dividing lines to compress the group of icons. Then select the drop down to see the icons. If you don't want to move the toolbar icons around, you can disable this function by selecting **Lock the Toolbars** from the toolbar context menu.

Part 3: Working with Resources

In Part 3, we're going to look in a bit more detail at projects. We'll also see how to compare and replace resources with other editions and recover resources you've deleted.

1. Previously, when we created projects, we took the defaults, including project location. By default, the resources you define are located in a folder named `workspace` in your main Eclipse folder. Browse this folder to view its contents. You'll see a folder structure that pretty much mirrors the project structure you see in your workspace.
2. When you create a project, you can create it in an alternate location, which is not in the folder `workspace`. Open the wizard to create a new project. Call it `My Alternate Project`. Uncheck **Use Default** for **Project contents** and select **Browse...** to select a location for your project. Choose a folder in your file system with a few files and subfolders. Select **Finish** to create the project.

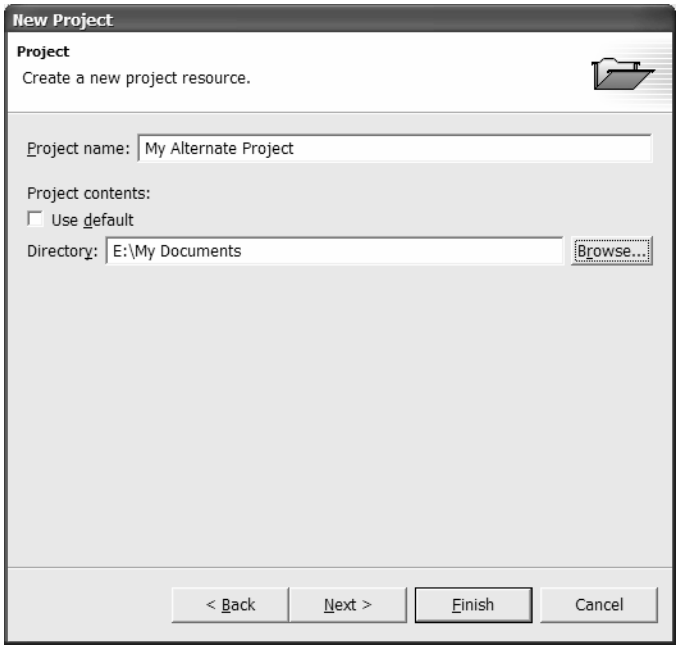


Figure 16-17
Creating a Project at an Alternate Location

In the Navigator view, you'll see the new project. You'll also see that the files and folders at the location you specified are automatically part of the project. A `.project` file is added to the file system to identify the target folder as an Eclipse project.

If you wanted to create a new project in a new location, you could add to the selected folder name or enter the drive and directory specification in the new project wizard.

3. Moving and copying files and folders between projects and other folders is easy. Select a file or folder in the Navigator view and then select **Copy** from the context menu. Select another project or folder and then select **Paste** from the context menu. Your selection is copied.

To move a file or folder, simply drag it and drop it on another folder or project. You can also copy a file with drag and drop. If you want to copy the file to another folder, press **Ctrl** and then drag and drop the file to the target folder. If you want to make a copy of the file in the same folder, press **Ctrl** and then drag and drop it in the target folder. You will get a dialog prompting for the new name.

4. Add a folder to **My Alternate Project**. Add a file to the folder you added. Browse your file system and go to this location. You'll see that Eclipse has created the folder and file at this location.

   Important Note: If you delete a project at an alternate location and specify to have its contents deleted also, Eclipse does just that. If this is the only copy of these resources, they will be deleted. Take care when deleting projects at alternate locations, especially when you mapped the project to and existing folder and file tree with content.

5. Now, let's see how to replace a file with a previous edition of it. Open an editor on **My First File** or switch to this editor if it is already open. Make a few changes to your text and save the file. Make some more changes and save the file again. Select **My First File** in the Navigator view and select **Replace With > Local history...** from the context menu.

   A dialog is displayed with the previous editions of **My First File** in the top pane and a side by side comparison of the current contents with the selected previous edition in the bottom pane.

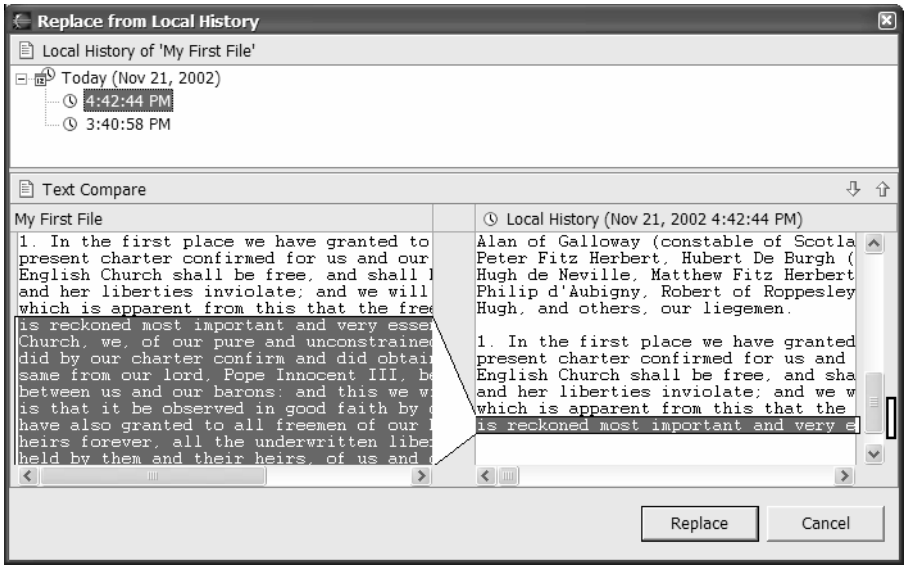


Figure 16-18
Replacing a File from Local History

Note: This is a good point to move on to the next part if you are just trying to experience a subset of each lab part.

6. Navigate the changes with **Select Next Change** and **Select Previous Change** . You can also select the change indicators (rectangles) in the overview ruler on the

right border. Select a previous edition of **My First File** and then select **Replace** to replace its contents in the editor.

Selecting **Compare With > Local History...** is similar to **Replace With**, except that it only shows the differences. It does not allow you to replace the file.

7. You can also select two files in the Navigator view and then select **Compare With > Each Other**. In the Navigator view, select **My First Project** and **My Second Project**. Then select **Compare With > Each Other** from the context menu.

A dialog is displayed showing the files and folders that are different between the two projects. The label decorations in the top pane (small plus, "+", and minus, "-" signs) indicate this. File `.project` has no label decoration. This means it exists in both projects. Double click on this file. The differences are shown in the bottom pane. Close the Compare Editor by clicking on the "**X**" on the tab of the editor.

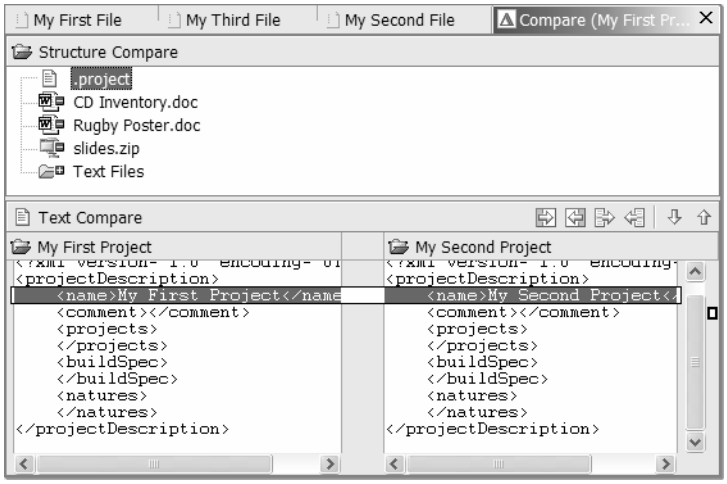


Figure 16-19
Comparing Two Projects

8. Now we'll see how to recover files you've deleted. From the Navigator view, delete **My First File** and **My Second File** from **My First Project**. Select **My First Project** and then select **Restore From Local History...** from the context menu. A list of files you have deleted from the project is displayed.

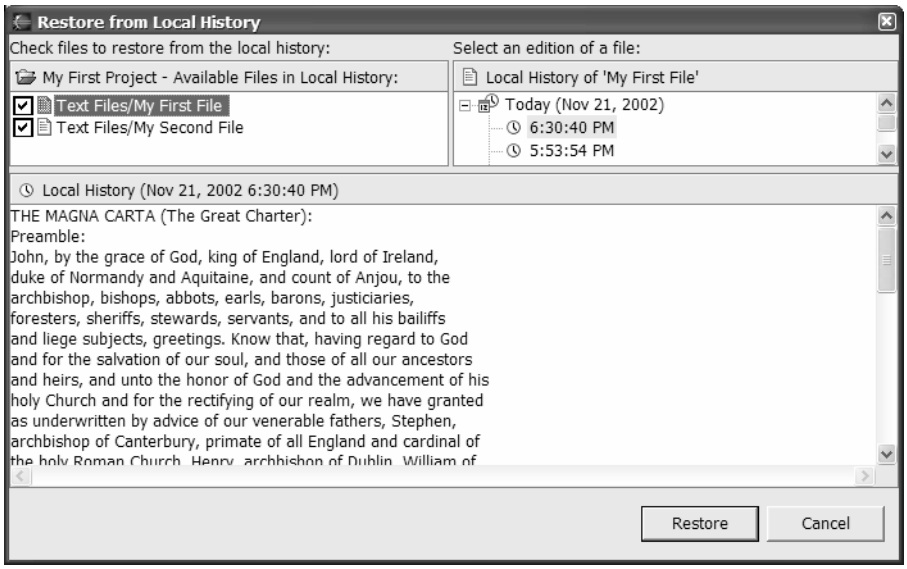


Figure 16-20
Recovering a Deleted File

9. Select a file to see the editions of the file Eclipse is maintaining. Select an edition to see its contents in the bottom pane. To restore a file, check the file in the upper left pane, select an edition, and then select **Restore**. Restore the files you deleted from **My First Project**.

@since 2.1 – Local history and linked resources. There are subtle differences in the use of the local history for the recovery of deleted resources that were linked resources. The local history support for recovery of a deleted linked file does not re-establish a link, nor does it return the most recent version of the linked file. The local history has current-1, current is always available from the file system given that a delete of a linked resource is a link delete, not a resource delete.

10. You can also recover projects you delete, if you do not delete their contents. Select **My First Project** in the Navigator view and then select **Delete** from the context menu. At the prompt, do not select to delete its contents. From the Navigator view, select **Import...** from the context menu. Select **Existing Project into Workspace** and then select **Next >**. Select **Browse...**. Browse your file system to go to folder `workspace` in your main Eclipse folder. In this folder, you'll see folder `My First Project`. This is the contents of the project left when you deleted the definition of the project. Select this folder and then select **OK**. If this folder is recognized as a project (by the presence of file `.project`), **Finish** is enabled. Select **Finish** to recover the project. Verify this in the Navigator view.

Part 4: Perspectives

Let's work a little more with perspectives, in particular, customizing your own perspective.

1. At this point, if you followed this script you have modified the default layout of the Resource perspective by adding the Bookmarks view. Close the Resource perspective by selecting **Close** from the context menu of the Resource perspective icon in the view shortcut area. Select **Window > Open Perspective > Resource**. The Resource perspective opens again, but without the changes you had made.
2. Change the Resource perspective by reorganizing the views and adding or deleting views. Select **Window > Save Perspective As...**, name the new perspective `My First Perspective` (no pun intended), and select **OK** to create your customized perspective.

   Observe the change on the title bar of the window. It now reflects that this is your perspective. Select Window > Open Perspective >. You'll see that your customized perspective is added to this list.
3. Select **Window > Customize Perspective...**. You'll see a dialog listing the menu actions that can be included in your perspective (see Figure 16-21). Expand the list in the left pane. The selected entries will appear in the menus for your perspective. The entries under **File > New**, **Window > Open Perspective**, and **Window > Show View** are the entries that appear on these menus. The other items are available, but you must select **Other...** first and then select them from a list. The items under **Other** are groups of menu items and toolbar buttons. Select an entry here to see the items that would get added. Try customizing **My First Perspective** by adding some items. Select **OK**. Verify the changes to your perspective.

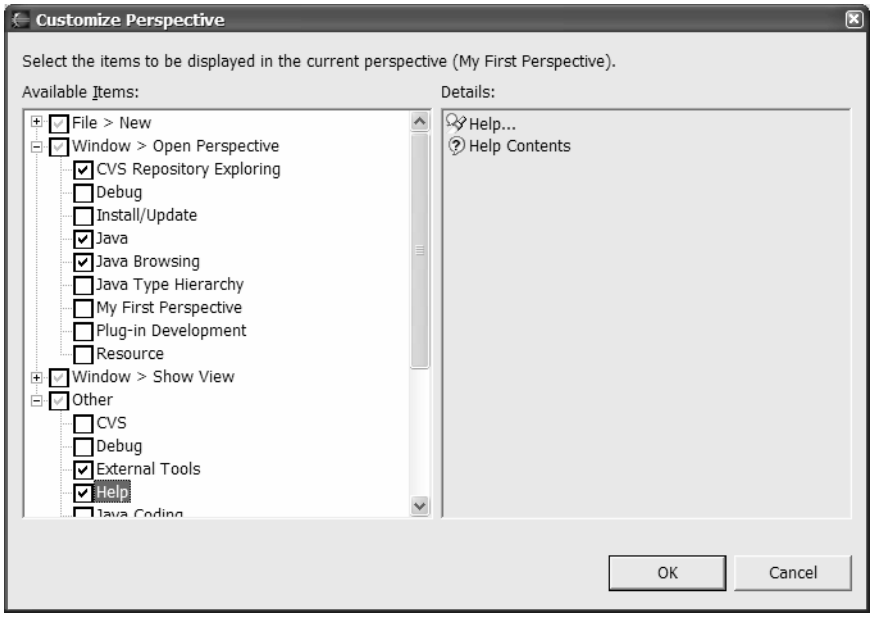


Figure 16-21
Customizing a Perspective

4. To make these changes permanent, you need to save your perspective again with **Window > Save Perspective As...**. Select **My First Perspective** to replace it with your changes. Or restore your perspective to its original state by selecting **Window > Reset Perspective**.
5. When you create a new project, the default perspective will be opened. If you create a new simple project while in your custom perspective, the Resource perspective will be opened. You can change the setting for the default perspective. Select **Window > Preferences**. Expand **Workbench** and select **Perspectives**. Under **Available perspectives**, select **My First Perspective** and then select **Make Default**. Select **OK**.

Exercise Activity Review

- The basic structure of Eclipse
- To navigate the user interface
- To move and resize views
- To compare and replace projects and editions of resources
- To recover deleted files and projects
- To customize a perspective

(Optional) Exercise 17 Using the Java Development Tools (JDT)

Introduction

The objective of this lab is to provide a hands-on demonstration of using Eclipse's Java Development Tools (JDT) to edit, run, and debug Java programs. We'll start with a basic "Hello World" program and then get into more detail on various JDT capabilities.

Note: This material has been tested on 2.1, but screen images still reflect the 2.0 user interface.

Skill Development Goals

At the end of this exercise, you should be able to:

- Write and run a simple "Hello World" program.
- Use the Java editor to more quickly and efficiently write Java code, including quick fix, content assist, code generation, and refactoring.
- Run Java programs using scrapbook pages

The time available during a formal class may not let you complete the entire exercise. Consider this subset first:

- Part 1 – all ("Hello World")
- Part 2 – 1 – 10 ("Quick Fix")
- Part 3 – 1 – 3 ("Code Generation")

- Part 4 – 1 – 4 (“Refactoring”)
- Part 5 – 1 – 10 (“Debugging”)

If you finish you can return to do the steps you skipped.

Exercise Instructions

Part 1. Hello World

Let’s start with the basics: The minimum required to create a class that we can execute, and two ways you can run the class.

1. Start Eclipse, if required. Select **Window > Open Perspective > Java** to open the Java perspective.

   @since 2.1 - Eclipse 2.1 provides additional filters that control the visibility of resources in the Package Explorer view. The simple projects created earlier now show up in the Package Explorer view by default. You can remove non-Java projects from the Package Explorer view by using the **Filters…** view menu choice and selecting the **non-Java Projects** option.

2. Select Create a Java Project. Name the project com.ibm.lab.usingjdt.helloworld. Leave the other values as the defaults and select Finish to create the project.
3. Ensure your project is selected in the Package Explorer view and select New Java Class on the action bar. Just use the default package and name the class `HelloWorld`. Under method stubs select only the main method option. Select **Finish** to generate the class.
4. A default package and the file `HelloWorld.java` is created for the HelloWorld class. Edit the method main as shown in Figure 17-1 and save your changes by selecting **Save** from the context menu or by pressing Ctrl+S.

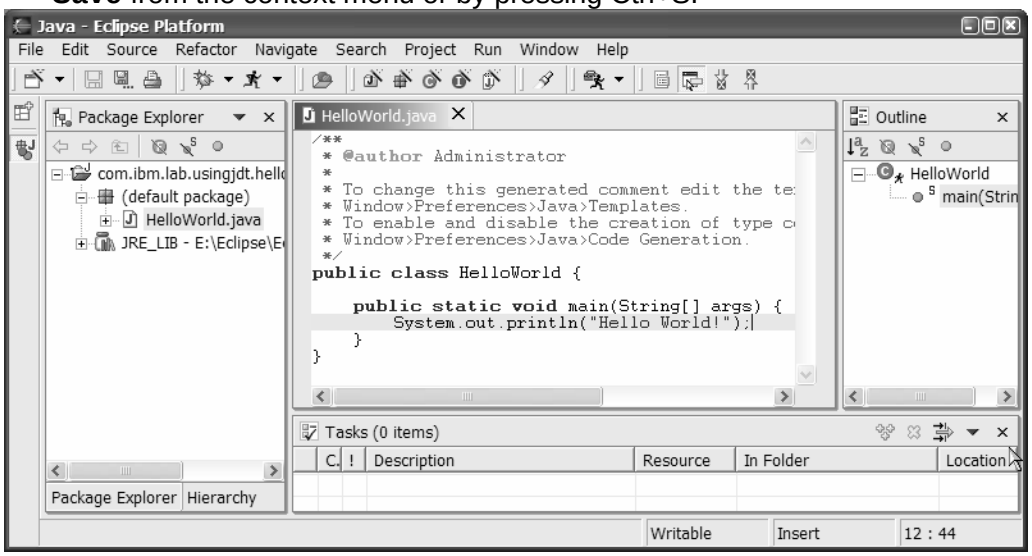


Figure 17-1
Hello World Class

5. Expand `HelloWorld.java` in the Package Explorer view. The "runner" decoration on the class icon indicates the class contains a main method and can be executed. Select class HelloWorld in the Package Explorer view and then select **Run > Run As > Java Application**. The main method in the HelloWorld class runs and and output is shown in the Console view.

Figure 17-2
Hello World Class Output

6. Now let's see another way to run a Java program. Ensure your project is selected in the Package Explorer view, select New Scrapbook Page, and call the page HelloWorld. File HelloWorld.jpage is added to your project. Scrapbook pages are another way to execute Java code. In addition to running your program, you can use them to try out snippets of Java code. Enter the following Java expression in the scrapbook page.

```
String[] array = {};
HelloWorld.main(array)
```

7. Select the entire expression (both lines) and then select **Display Result of Evaluating Selected Text** from the editor toolbar. The expression is evaluated, which causes method main of the HelloWorld class to execute. The return value displays in the scrapbook page (in this case, there is no return value) and output is shown in the Console view.

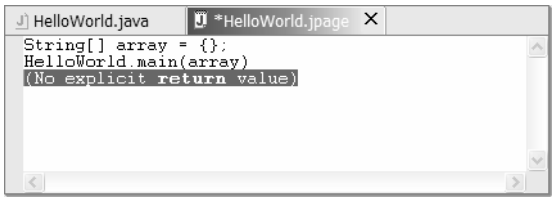


Figure 17-3
Hello World Scrapbook Page Output

Part 2. Quick Fix

In Part 2, we'll see how to navigate and fix errors in Java code. A lab template project named `com.ibm.lab.usingjdt` contains several packages. Be sure you have this in your workspace. If you do not, refer to the instructions below.

Import Using Java Project

Import com.ibm.lab.usingjdt project from the `C:\Lab_Import_Master\JavaProjects` directory. This directory should have been created using the lab templates zip file which is required to support this and other exercises.

1. Open the Import wizard using the Eclipse **File > Import** menu option.
2. Select the **Existing Project into Workspace** option

3. Use the **Browse** button to locate and select the `com.ibm.lab.usingjdt` folder (**not** the `com.ibm.lab.soln.usingjdt` folder).
4. Press **Finish**, this will import the project into your current workspace; the project will be part of the workspace but physically exist at the file system location referenced during import. The errors in the Tasks view after the import are expected.

Quick Fix Instructions

Each package in the imported project contains code for one part of this exercise, as indicated by the name of the package. The example code represents several iterations of a program to generate numbers, prime numbers at first. While the code is relatively simple and at times may appear a bit contrived, the example allows us to focus on demonstrating the use of JDT with a minimal amount of code.

The same class is defined in multiple packages. We've done this to provide on set of programs we can carry through the different parts of the exercise. The packages hold different versions of the same classes so that you don't have to do parts 1 through 5 if all you want is part 6. In this part of the exercise, you'll be working with package com.ibm.lab.usingjdt.quickfix. You should see a number of errors in package quickfix. These are what you are going to navigate to and fix.

1. Select Window > Preferences > Java > Editor > Annotations. Ensure that the two options on this page for error identification and resolution are selected (checked). Select OK.
2. Before getting started, let's simplify what is shown in the Package Explorer view. The package names are too long and we really don't need to see the JRE entries. Open the Java > Appearance preferences page. Select the **Compress all package name segments, except the final segment** option and enter "0" as the Compression pattern. Select OK to save the preference setting change.

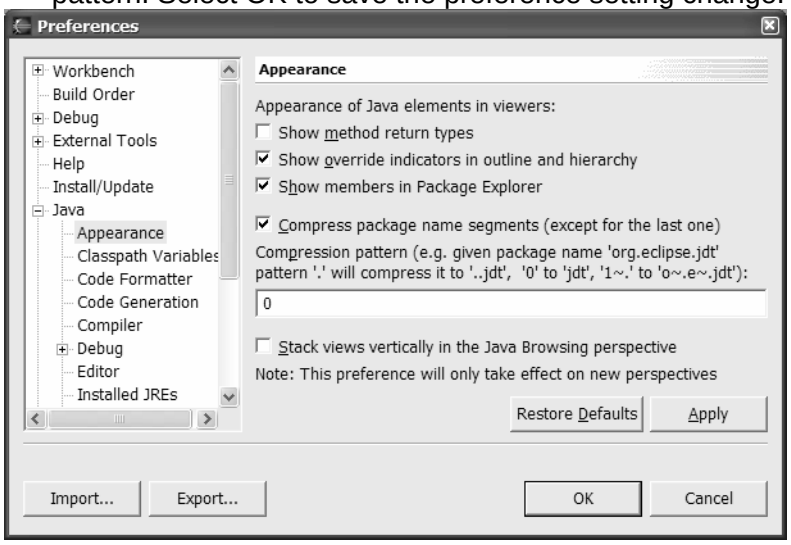


Figure 17-4

Java Appearance Preferences

3. To hide the JRE entries, select **Filters...** from the Package Explorer pulldown and then select **Referenced libraries**. Select OK.

Figure 17-5
Filtering Package Explorer Contents

4. In package quickfix, open class `PrimeNumberGenerator`. The JDT indicates errors in a number of ways:

   - Error clues (small red rectangles) in the overview ruler in the right margin of the editor.
   - Error clues (red underlining) on text in the editor.
   - Quick fix icons on the marker bar in the left margin of the editor, indicating there are suggestions to fix the error.
   - Build Error entries in the Tasks view.
   - Build Error markers on the marker bar. Sometimes these are overlaid with the quick fix icon.
   - Build Error label decorations on icons in the Java views and editor tab(s)

   In the next several steps, we'll see how to navigate to the errors and get information about them.

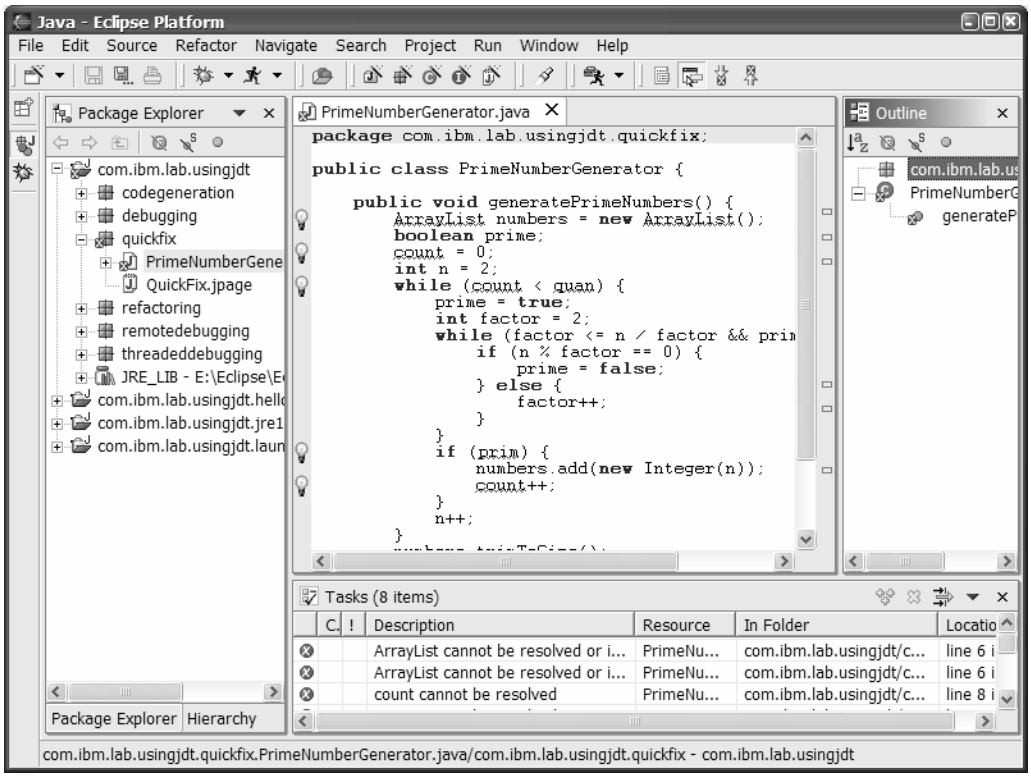


Figure 17-6
Java Error Indicators

5. Click on an error indicator in the overview ruler; these are the small red rectangles. The text scrolls to the line with the error and the error is highlighted. This is a useful way to navigate to errors without having to manually scroll the file. Hollow rectangles indicate errors found through code analysis as you edit. Solid rectangles indicate compilation errors.
6. Close the file and double click on one of the errors in the Tasks view or select Go To from the context menu. The file with the error is opened and the error highlighted. If an error message in the Tasks view is truncated, you can hover over it in the Tasks view to see the full message.
7. Hover over text underlined in red in the editor and you'll see a description of the error, or position the cursor in the error text and press F2. Hover over one of the error indicators in the left margin to see the error message. When a quick fix icon is not shown you can also click on an error indicator in the left margin to select the error text.
8. Let's use quick fix to fix these errors. Click on the first quick fix icon, the one for the `Arraylist` reference. You will see a list of possible solutions (see Figure 17-7). As you select each of the proposed solutions you'll see the code that would be generated or a description of the fix. In this case, we omitted an import statement. Select `import 'java.util.ArrayList'` and press Enter. The error clues disappear, but the build errors remain. They will do so until you re-build the file. Save the file. The build problem is also resolved. On to the rest of the errors.

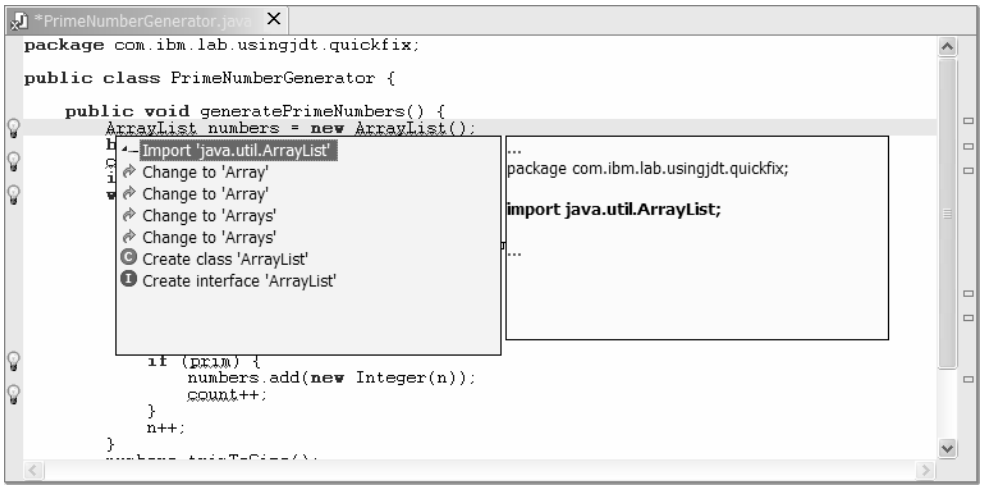


Figure 17-7
Quick Fix Proposed Solutions

9. Go to the next problem by selecting Go to Next Problem . The error message is displayed in the message area at the bottom of the window. Press Ctrl+1 to activate quick fix. In this case, we omitted the definition of a local variable. Select Create local variable 'count' to see the code that would be generated, including the original initialization. Press Enter. The original line setting count to 0 is no longer necessary, so delete it.
10. Go to the next error on the while statement. If you use Go to Next Problem to do this, you will see another instance of variable count not being resolved. You've already fixed this so select Go to Next Problem again to get to the error on `quan`. We did not define `quan`, the number of prime numbers to generate. Click on the quick fix icon in the marker bar in the left margin of the editor. Again, we have several options. Let's make quan a field on `PrimeNumberGenerator`. Select this option and press Enter. By default, the new field is `private`. Change this to a `public` field. Initially, we're going to access this field directly outside the class for simplicity. We'll change this later.

Note: This is a good point to move on to the next part if you are just trying to experience a subset of each lab part.

11. Go to the next error; a simple misspelling. The JDT can detect and fix these, too. Click on the quick fix icon, select Change to '`prime`', and press Enter.
12. On to the final quick fix icon; a missing definition for results. In this case, we want to define results as a field of `PrimeNumberGenerator`. Click on the quick fix icon, select Create field '`results`', and press Enter. All of the quick fix icons should be gone. The build error indicators remain. Save the file and these should disappear as well.
13. We're almost through with this first version of our code. We have our results, but no way yet for them to be accessed. Select field `results` in the Outline view and then select **Source > Generate Getter and Setter...** from the context menu. The getter and setter methods for `results` should be selected.

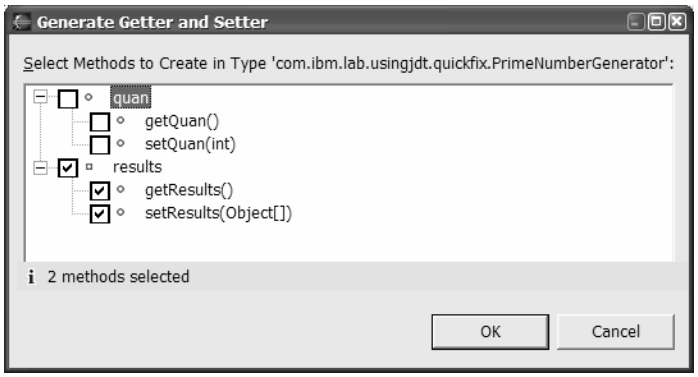


Figure 17-8
Generating Getter and Setter Methods
Select OK to generate the getResults and setResults methods.

14. In method `generatePrimeNumbers`, replace the "`results = numbers.toArray()`" statement with "`setResults(numbers.toArray());`".

    We'll see a way to combine these two steps later when we go through you some refactoring exercises. Save the file.

15. Great! We're finished with the first version of the code. The code should look like this, minus the generated comments.

```
package com.ibm.lab.usingjdt.quickfix;
import java.util.ArrayList;

public class PrimeNumberGenerator {
 public int quan;
 private Object[] results;

 public void generatePrimeNumbers() {
    int count = 0;
    ArrayList numbers = new ArrayList();
    boolean prime;
    int n = 2;
    while (count < quan) {
       prime = true;
       int factor = 2;
       while (factor <= n / factor && prime) {
          if (n % factor == 0) {
             prime = false;
          } else {
             factor++;
          }
       }
       if (prime) {
          numbers.add(new Integer(n));
          count++;
       }
       n++;
    }
    numbers.trimToSize();
    setResults(numbers.toArray());
```



```
  }
  public Object[] getResults() {
     return results;
  }
  public void setResults(Object[] results) {
     this.results = results;
  }
}
```

16. Let's test the code. Select package `quickfix` in the Package Explorer view, select Create a Scrapbook Page, and call it `QuickFix`. Enter the following expression in the scrapbook page to test your code.

```
PrimeNumberGenerator p = new PrimeNumberGenerator();
p.quan = 10;
p.generatePrimeNumbers();
p.getResults();
```

17. We need to set an import statement in the scrapbook page so `PrimeNumberGenerator` can be resolved. This is because there are additional definitions of the class in other packages in the project. We'll be using these in other parts of this exercise. From the scrapbook page editor context menu, select **Set Imports** and then select **Add Type....** Enter the first few charaters of the `PrimeNumberGenerator` type and then select it and `com.ibm.lab.usingjdt.quickfix` for the Qualifier. Select **OK** and then **OK** again to set the import.

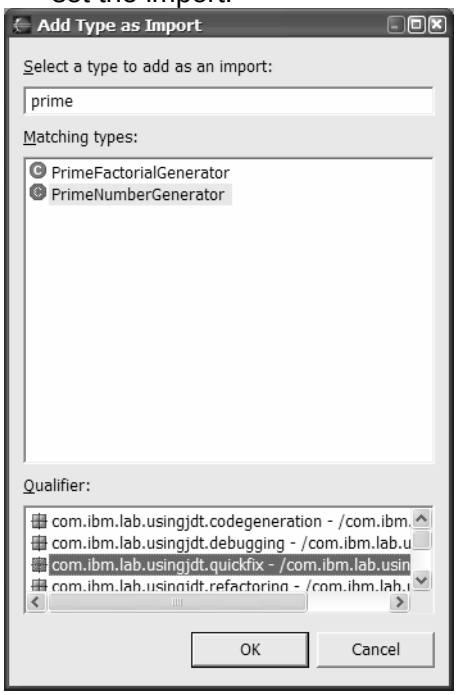


Figure 17-9
Setting Imports for a Scrapbook Page

18. In the scrapbook page, select the entire expression you entered and then select **Inspect** from the context menu. Method generatePrimeNumbers runs and the output is shown in the Expressions view. Select Show Details Pane to open a pane in the bottom of the view; it will show the results of the selected object's toString() method.

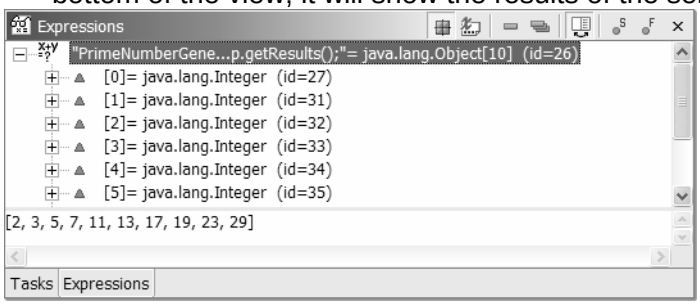


Figure 17-10
PrimeNumberGenerator Results

When you're through, close the open editors open on PrimeNumberGenerator.java and QuickFix.jpage and close the Expressions view.

Part 3. Code Generation

In this part, we're going to take a closer look at how you can generate code with content assist, including completing statements, generating code from templates, and generating anonymous inner classes. We're going to enhance class `PrimeNumberGenerator` with methods to output and to sort its results, the prime numbers.

Unless we specify otherwise, when we refer to a class or some other Java element or resource, we are referring to the element or resource in the `com.ibm.lab.usingjdt.codegeneration` package. The code in this package is the same as the code in package `quickfix` now, if you did all of Part 2.

1. First, we're going to code the method to output the prime numbers. In package `codegeneration`, open class `PrimeNumberGenerator`. Just before the comments for the `getResults` method, enter a new line, begin typing "`pub`", and with the insertion cursor just after the "b" press Ctrl+Space to active content assist and see a list of suggestions (see Figure 17-11).

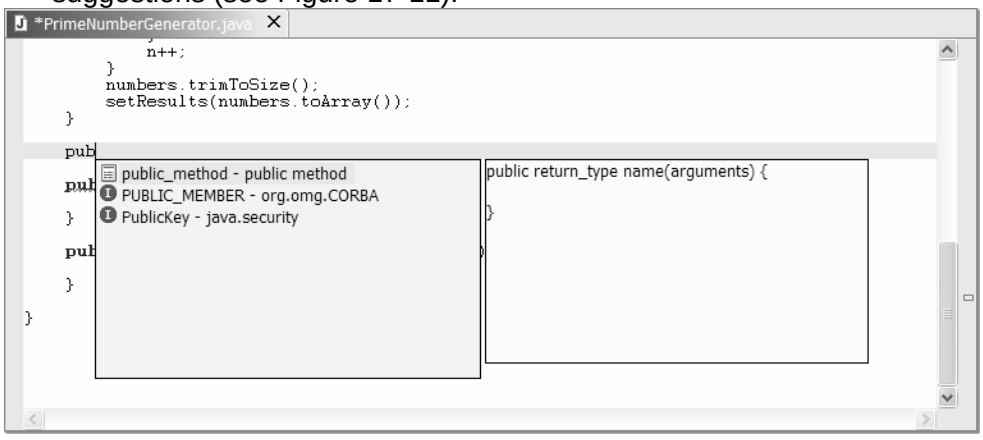


Figure 17-11
Content Assist Suggestions

2. Select **public_method – public method**. This is a code template for defining public methods. Press Enter. Don't enter any more keystrokes, you are ready to use the template (see Figure 17-12).

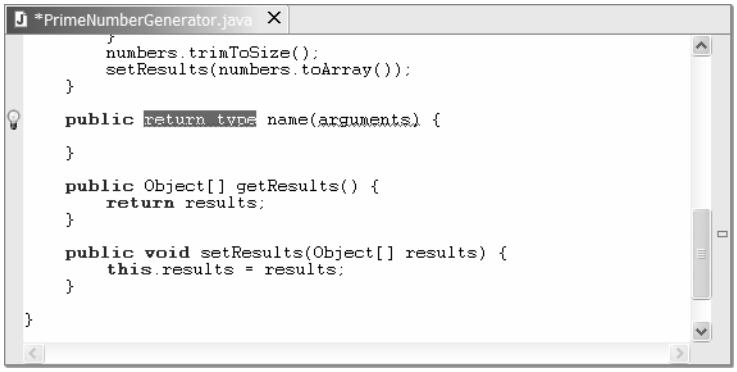


Figure 17-12
Public Method Template

You now have a stub for the new method. Because the method is incomplete (and incorrect), you'll see the quick fix icon. You could use quick fix at this point, but there is an easier way.

3. The placeholder for the return type of the method is selected. Enter a return type of `void` for the method. Press Tab and `name` is selected. This is the next placeholder you can modify. Type `outputResults`. Press Tab again and type over the `arguments` placeholder with "`String prefix`". Press Tab again and the cursor is placed in the body of the method for you to begin coding. Neat!

   If you open the preference page **Java > Editor > Templates**, you can see a list of all the currently available JDT editor templates. You can modifiy these or add your own. This powerful function in the JDT editor is open ended.

Note: This is a good point to move on to the next part if you are just trying to experience a subset of each lab part.

4. On to the rest of the method. Enter statement "`Date d = new Date();`".

You'll get an error clue with this statement. Hover over the text that's underlined in red and you'll see that Date cannot be resolved. We neglected the import statement. We saw previously how we could use quick fix here. We can also use code generation to fix this. Select one of the Date references and then select Source > Add Import from the context menu. Here we have two choices. Select java.util.Date and then OK to generate the import statement.

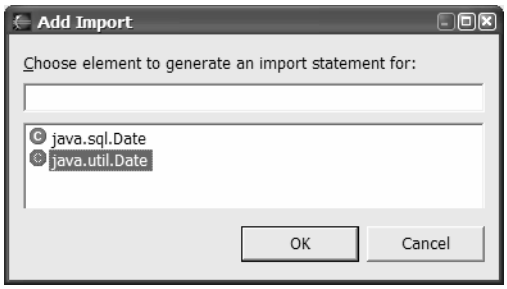


Figure 17-13
Generating an Import Statement

5. Create a new line after Date d = new Date();. On the new line type “System.“ and then press Ctrl+Space. Select field out and press Enter. Type a period, “.”, and then press Ctrl+Space. Select println(String arg0) and then press Enter. The statement is completed, the cursor positioned, and a prompt appears indicating the parameter type.

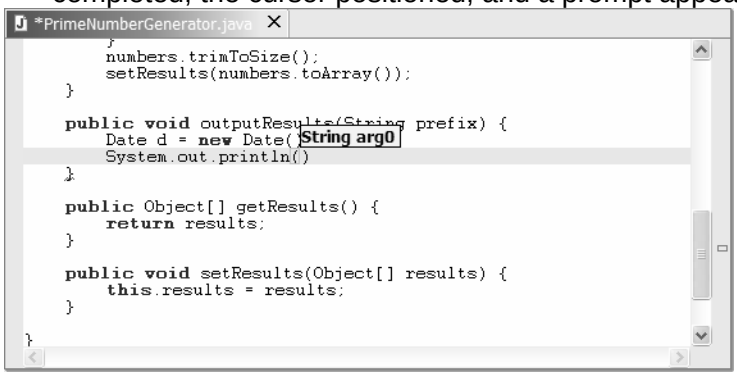


Figure 17-14
Parameter Hint

Type “pr” and press Ctrl+Space. Select prefix from the list and press Enter. Enter “ + d.” and press Ctrl+Space. Select toString() and press Enter. Add a semicolon to complete the statement. When you have many choices in the content assist prompt, you can continue typing to narrow the choices. You should have the following statement.

```
System.out.println(prefix + d.toString());
```

6. After this statement, create a new line, begin typing “for“, and press Ctrl+Space. Select template for – iterate over array in the list and press Enter to add a block of code for the for statement. Tab to array, delete this and press Ctrl+Space. Add getResults(). Press Tab to go to the new line in the body of the for statement. Enter the following statement with content assist.

```
System.out.println(prefix + getResults()[i]);
```


7. We're finished coding the method. Let's add a Javadoc comment. Position the insertion cursor in the code in method outputResults and select Source > Add Javadoc comment from the context menu. Create a new line in the comment and press Ctrl+Space. Content assist also allows you to add content to Javadoc comments, e.g. an @author tag.

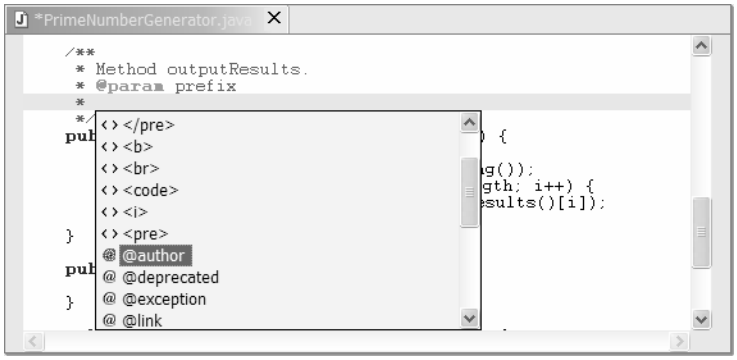


Figure 17-15
Generating a Javadoc Comment

8. Save the file. Your code for `outputResults` look like this.

```
/**
 * Method outputResults.
 * @author Shavor, et al
 * @param prefix
 */
public void outputResults(String prefix) {
  Date d = new Date();
  System.out.println(prefix + d.toString());
  for (int i = 0; i < getResults().length; i++) {
        System.out.println(prefix + getResults()[i]);
  }
}
```

Wow! This gives you an idea of the power of content assist and code generation. Save your changes. Let's test the code.

In the Package Explorer view, select package codegeneration. Select Create a scrapbook page and call it CodeGeneration. Select **Set Imports** from the context menu of the scrapbook page editor and add an import for package com.ibm.lab.usingjdt.codegeneration. Enter the following expression. Content assist works in scrapbook pages; try it.

```
PrimeNumberGenerator p = new PrimeNumberGenerator();
p.quan = 50;
p.generatePrimeNumbers();
p.outputResults("");
```

Select the entire expression and then select **Execute** from the context menu. The results appear in the Console view.

9. Let's look at a more involved example of content assist in which we'll define an anonymous inner class and override a method. To do this, we're going to create a method to sort the results. Since the results are already in order because of the way we generate them, our sort routine will sort in reverse order. Granted, this is a bit contrived, but it will illustrate what we want with a minimal amount of code.

   In the editor for `PrimeNumberGenerator`, add import statements for `java.util.Arrays` and `java.util.Comparator`.

10. Below method outputResults, use content assist to create a new public method, return type void, name sortResults, with no parameters. Position the cursor in the body of the method, type "Arrays." and press Ctrl+Space. Select sort(Object[] a, Comparator c) and press Enter. For the first parameter, enter "getResults(),". Once you type the comma, watch as the highlighting on the parameter prompt changes to indicate you need to enter the Comparator for the second argument.

11. For the second argument, press Enter to start a new line. Begin typing "new Com", and press Ctrl+Space. Select Comparator from java.util and press Enter. Add "() { }" to create the body of the class. Here we're defining an anonymous innerclass to be used to sort our results array. Position the cursor in the body of the class definition, that is between "{" and "}", and press Ctrl+Space. Select compare and press Enter. Content assist knows what methods you can override.

12. Replace the return 0 line to complete the invocation of sort as follows. Save the file.

```
public void sortResults() {
  Arrays.sort(getResults(), new Comparator() {
    public int compare(Object arg0, Object arg1) {
      if (((Integer) arg0).intValue()
          < ((Integer) arg1).intValue()) {
        return 1;
      } else {
        return -1;
      }
    }
  });
}
```

13. Test sortResults with the following code in CodeGeneration scrapbook page.

```
PrimeNumberGenerator p = new PrimeNumberGenerator();
p.quan = 20;
p.generatePrimeNumbers();
p.sortResults();
p.outputResults("");
```

This adds output to what is already displayed in the Console view. To clear the Console view, select Clear Console  .

When you're through, don't forget to close the editors open on PrimeNumberGenerator.java and CodeGeneration.jpage.

Part 4. Refactoring

In Part 4, we're going to look at refactoring Java. We're going to use these refactoring capabilities to clean up the PrimeNumberGenerator class, reorganize it, and add a new class to generate the prime factorials.

Unless we specify otherwise, when we refer to a class or some other Java element or resource, we are referring to the element or resource in the `com.ibm.lab.usingjdt.refactoring` package. The code in this package is the same as the code in package `codegeneration` now, if you did all of Part 3.

1. Recall in Part 3 (If you did it all), we made `results` a field and then updated a reference to it. We can do this in one step by refactoring. Open class `PrimeNumberGenerator` in package `refactoring`. Select field `quan` in the Package Explorer view, the Outline view, or the editor and then select **Refactor > Encapsulate Field...** from the context menu. If you have editors open with unsaved changes, you will be prompted to save them; do so.

   Leave the selections on the first page of the Refactoring wizard as they are and select **Preview >**. You will see the proposed changes, as shown in Figure 17-16. On this page of the Refactoring wizard, you can select which code changes you want to keep. Expand the list in the top pane to see the list of fields and methods that will be changed. Select entries in the list to see their respective changes in the bottom panes. You can also select the rectangles on the overview ruler to the right of the source panes to go directly to the respective change. Leave all of the proposed code changes selected and then select **OK** to make the changes.

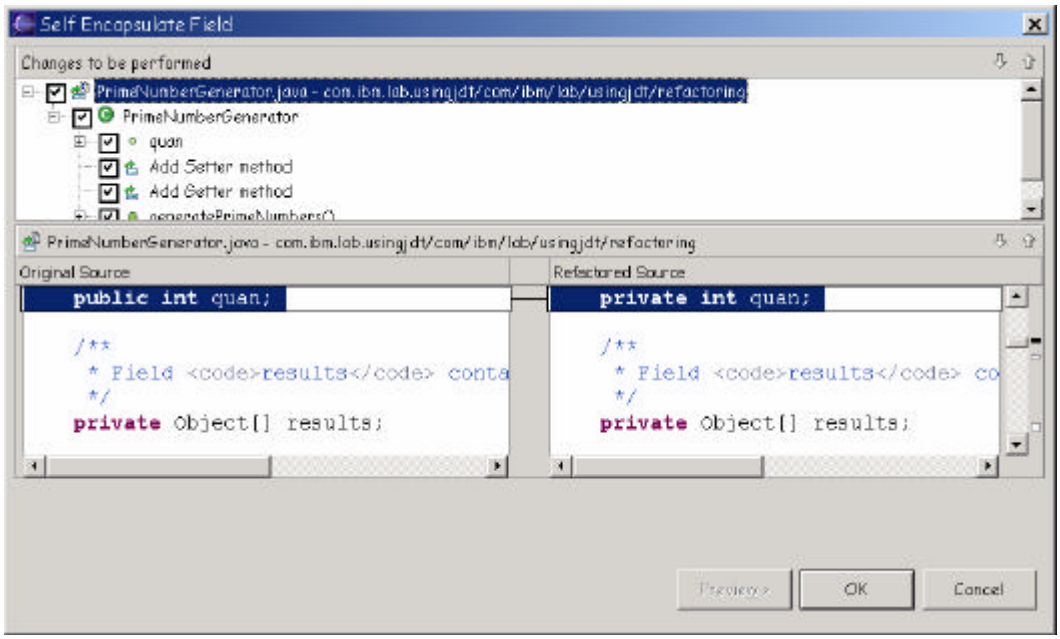


Figure 17-16
Self Encapsulate Refactoring

2. The name `quan` is not very descriptive. Let's change it to quantity. Select field `quan` (in a view or in the editor) and then select **Refactor > Rename...** from the context menu. Enter `quantity` for the name. Select the last two options to rename the getter and setter methods, too.

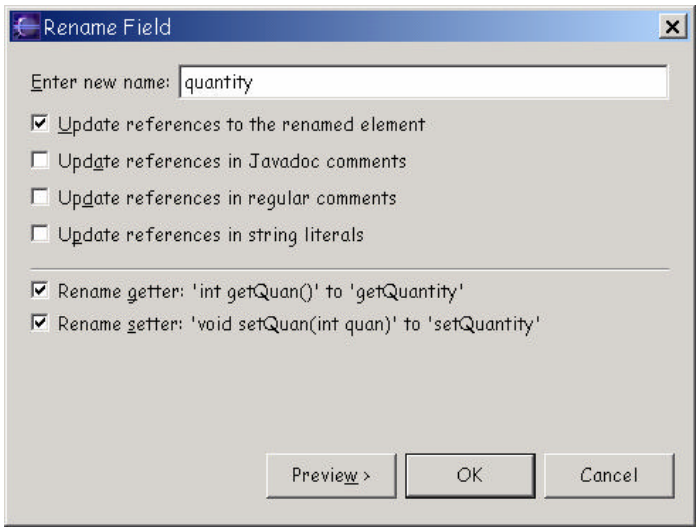


Figure 17-17
Rename Refactoring

3. Select **Preview >**. Review the proposed code changes. Select **OK**.
4. Select the `generatePrimeNumbers` local variable n in the editor and then select **Refactor > Rename...** to rename it to something more descriptive like `candidate`. Save the file. Your code should look like this:

```
public class PrimeNumberGenerator {
 private int quantity;

 private Object[] results;

 public void generatePrimeNumbers() {
    int count = 0;
    ArrayList numbers = new ArrayList();
    boolean prime;
    int candidate = 2;
    while (count < getQuantity()) {
       prime = true;
       int factor = 2;
       while (factor <= candidate / factor && prime) {
          if (candidate % factor == 0) {
             prime = false;
          } else {
             factor++;
          }
       }
       if (prime) {
          numbers.add(new Integer(candidate));
          count++;
       }
       candidate++;
    }
    numbers.trimToSize();
    setResults(numbers.toArray());
 }
```



```
  /**
   * Method outputResults.
   * @author Shavor, et al
   * @param prefix
   */
  public void outputResults(String prefix) {
     Date d = new Date();
     System.out.println(prefix + d.toString());
     for (int i = 0; i < getResults().length; i++) {
        System.out.println(prefix + getResults()[i]);
     }
  }
  public void sortResults() {
     Arrays.sort(getResults(),
     new Comparator() {
        /**
         * @see java.util.Comparator#compare(Object, Object)
         */
        public int compare(Object arg0, Object arg1) {
           if (((Integer) arg0).intValue() < ((Integer)
arg1).intValue()) {
              return 1;
           } else {
              return -1;
           }
        }
     });
  }
  public Object[] getResults() {
     return results;
  }
  public void setResults(Object[] results) {
     this.results = results;
  }
  public void setQuantity(int quan) {
     this.quantity = quan;
  }
  public int getQuantity() {
     return quantity;
  }
}
```

Note: This is a good point to move on to the next part if you are just trying to experience a subset of each lab part.

5. We've cleaned up the code a bit. Now we're going add a class to generate the prime factorials, `PrimeFactorialGenerator`. To do this, we're going to create a superclass for `PrimeNumberGenerator` and `PrimeFactorialGenerator` called `NumberGenerator` and move methods and fields to this new superclass. Select package `refactoring` in the Package Explorer view and create class `NumberGenerator`. Do not select to have any method stubs generated.

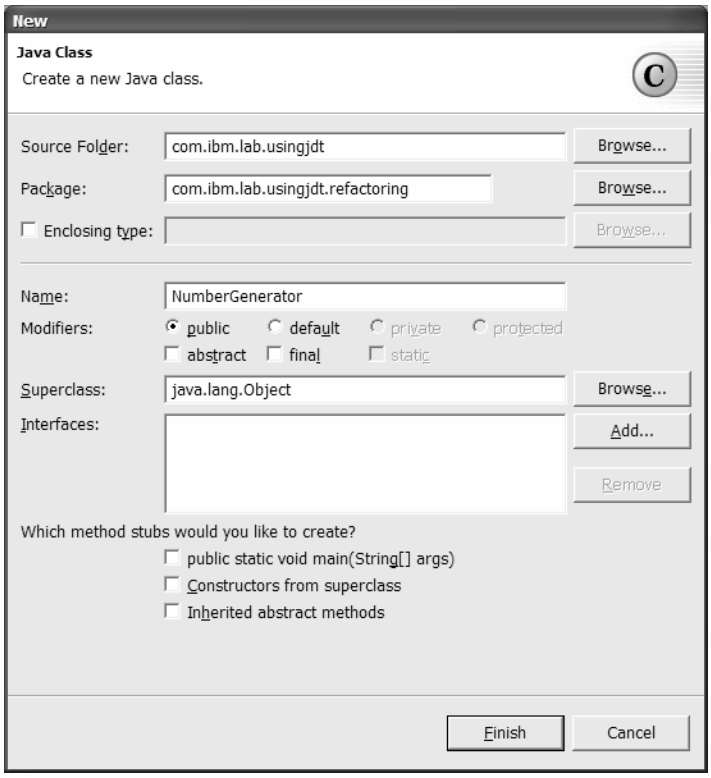


Figure 17-18
Creating Class NumberGenerator

6. In the superclass, we need a method the subclasses will override to generate numbers. Create the following method in `NumberGenerator` (try it by generating from a template).

```
public void generateNumbers() { }
```

7. Now we need to fix `PrimeNumberGenerator` to override this method. In class `PrimeNumberGenerator`, select method `generatePrimeNumbers` and then **Refactor > Rename...** to rename it to `generateNumbers`. If you did not save the changes to `NumberGenerator.java`, you will be prompted to do so.
8. `PrimeNumberGenerator` should be a subclass of `NumberGenerator`, so change class `PrimeNumberGenerator` to extend `NumberGenerator` by editing its definition as follows. (Hint, use Ctrl+Space to enter this additional code).

```
public class PrimeNumberGenerator extends NumberGenerator {
```

9. Because `PrimeNumberGenerator` and the class we will create to generate prime factorials will inherit from `NumberGenerator`, fields `quantity` and `results` and their getter and setter methods really belong in `NumberGenerator`, as do methods

`outputResults` and `sortResults`. Select the editor for `PrimeNumberGenerator`. In the Outline view, select fields `results` and `quantity` and then **Refactor > Pull Up...** from the context menu. Review the code changes and select **Finish**. You may have noticed that these changes span multiple files. In fact, when you perform a refactoring operation, the refactoring analysis includes all open projects in your workspace.

10. We need to pull up the getters and setters and methods `sortResults` and `outputResults` (six methods in all). In the Outline view for `PrimeNumberGenerator`, select these six methods, select **Refactor > Pull Up...** from the context menu, and then select **Next >**. In this Refactoring dialog, the code in the left pane is the code before the changes. Select **Finish**.

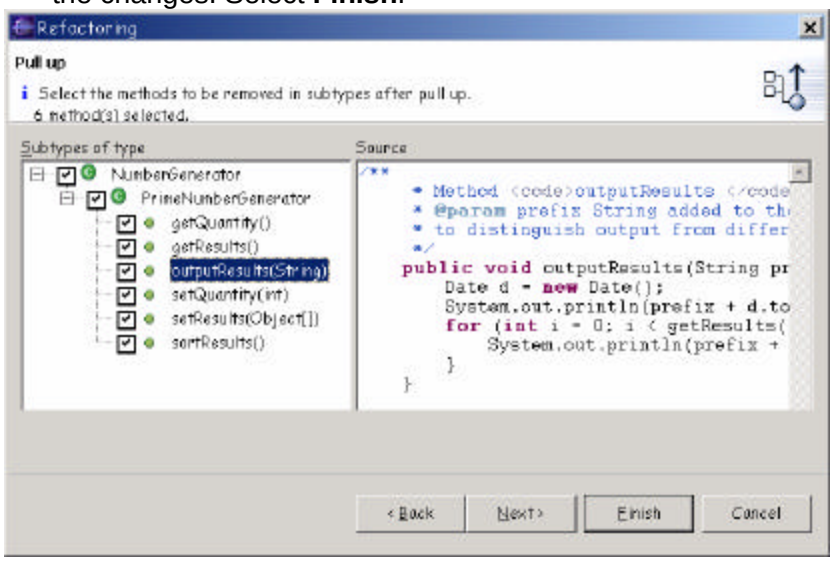


Figure 17-19
Pull Up Refactoring

11. The organization of classes `NumberGenerator` and `PrimeNumberGenerator` should now look like this:

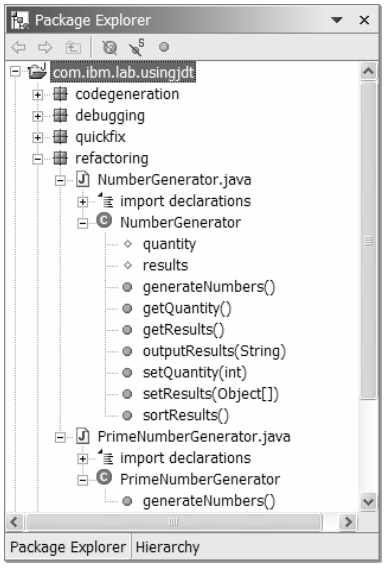


Figure 17-20
Code Organization After Refactoring

12. Now class `NumberGenerator` is defined and `PrimeNumberGenerator` is refactored. We now need to create the class to generate prime factorials. In package `refactoring`, create class `PrimeFactorialGenerator`. For its Superclass, select **Browse...** and then enter `NumberGenerator`. In the Superclass Selection dialog, for Qualifier be sure to specify package `com.ibm.lab.usingjdt.refactoring`. Select **OK** to set the superclass. Do not generate any of the method stubs. Select **Finish** to generate the class.
13. To generate the numbers, we need to override `generateNumbers` in `NumberGenerator`. Do this in the Package Explorer view by dragging method `generateNumbers` from class `NumberGenerator` and dropping it on class `PrimeFactorialGenerator` (not the .java file) to create a method stub with the same signature. Enter the following for method `generateNumbers` in class `PrimeFactorialGenerator`.

```
public void generateNumbers() {
  PrimeNumberGenerator primes = new PrimeNumberGenerator();
  primes.setQuantity(getQuantity());
  primes.generateNumbers();
  Object[] numbers = new Object[primes.getResults().length];
  int factorial = 1;
  for (int i = 0; i < primes.getResults().length; i++) {
     factorial = factorial *
        ((Integer) primes.getResults()[i]).intValue();
     numbers[i] = new Integer(factorial);
  }
  setResults(numbers);
}
```

14. Congratulations, you're done. Save your changes to the three files that you're editing. Your code should look something like this. To save trees, some sections of code unchanged from step 4 are replaced with comments and other comments eliminated.

```
package com.ibm.lab.usingjdt.refactoring;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Comparator;
import java.util.Date;

public class NumberGenerator {
  public void generateNumbers() { }

  public void outputResults(String prefix) {
  //Relocated from PrimeNumberGenerator
  //Code unchanged
  }

  public void sortResults() {
  //Relocated from PrimeNumberGenerator
  //Code unchanged
  }
```



```
 public Object[] getResults() {
    return results;
 }

 public void setResults(Object[] results) {
    this.results = results;
 }

 public void setQuantity(int quan) {
    this.quantity = quan;
 }

 public int getQuantity() {
    return quantity;
 }

 protected int quantity;

 protected Object[] results;
}

package com.ibm.lab.usingjdt.refactoring;

import java.util.ArrayList;
import java.util.Date;
import java.util.Comparator;
import java.util.Arrays;

public class PrimeNumberGenerator extends NumberGenerator {

 public void generateNumbers() {
 //Unchanged
 }
}

package com.ibm.lab.usingjdt.refactoring;

public class PrimeFactorialGenerator extends NumberGenerator {

 public void generateNumbers() {
    PrimeNumberGenerator primes = new PrimeNumberGenerator();
    primes.setQuantity(getQuantity());
    primes.generateNumbers();
    Object[] numbers = new Object[primes.getResults().length];
    int factorial = 1;
    for (int i = 0; i < primes.getResults().length; i++) {
       factorial = factorial *
          ((Integer) primes.getResults()[i]).intValue();
       numbers[i] = new Integer(factorial);
    }
    setResults(numbers);
 }
}
```


15. Give the new and refactored code a try by executing the following in a scrapbook page (remember to set an import statement).

```
PrimeFactorialGenerator f = new PrimeFactorialGenerator();
f.setQuantity(5);
f.generateNumbers();
f.sortResults();
f.outputResults("");

PrimeNumberGenerator n = new PrimeNumberGenerator();
n.setQuantity(5);
n.generateNumbers();
n.outputResults("");
```

When you're through, don't forget to close any editors open on .java and .jpage files.

Part 5. Debugging

In Part 1, we're going to look at debugging Java programs, including using different kinds of breakpoints, examining program execution, viewing and changing variable values, using launch configurations, and referring to classes in a runtime library. To help illustrate this exercise, we've provided project `com.ibm.lab.usingjdt`, containing several packages. This code represents several iterations of a program to generate numbers. It is the same code we used in the earlier steps.

If you have not completed all of the steps in the previous exercise parts, you may see errors in some of the packages. Don't worry about the errors, they're intentional. We put them there so they could be fixed during the exercise. In any case, you can ignore them for this exercise.

The same class is defined in multiple packages. Unless we specify otherwise, when we refer to a class or some other Java element or resource, we are referring to the element or resource in package `com.ibm.lab.usingjdt.debugging`. We've also added class `DebuggingExample` with a simple `main` method to drive the code and a launch configuration, `DebuggingExample.launch`.

1. Using the Java perspective, edit class `DebuggingExample` in package `debugging` and set a breakpoint on the first statement in the method by double clicking on the marker bar of the editor next to the line (see Figure 2-21).

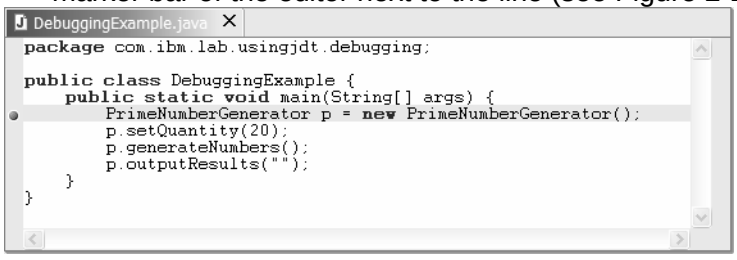


Figure 2-21
Setting a breakpoint

2. Select the **Debug** pulldown and then **Debug As > Java Application**. The `main` method in class `DebuggingExample` executes, the Debug perspective opens, and execution suspends before the line on which we defined the breakpoint. The breakpoint icon is decorated with a checkmark. This indicates that the class has been loaded by the Java VM running in this debug session. Each debugging session has its

own associated Java VM. This is important because it means that if you are not running a Java VM that supports hot code replace and you change the class, you must restart your debug session.

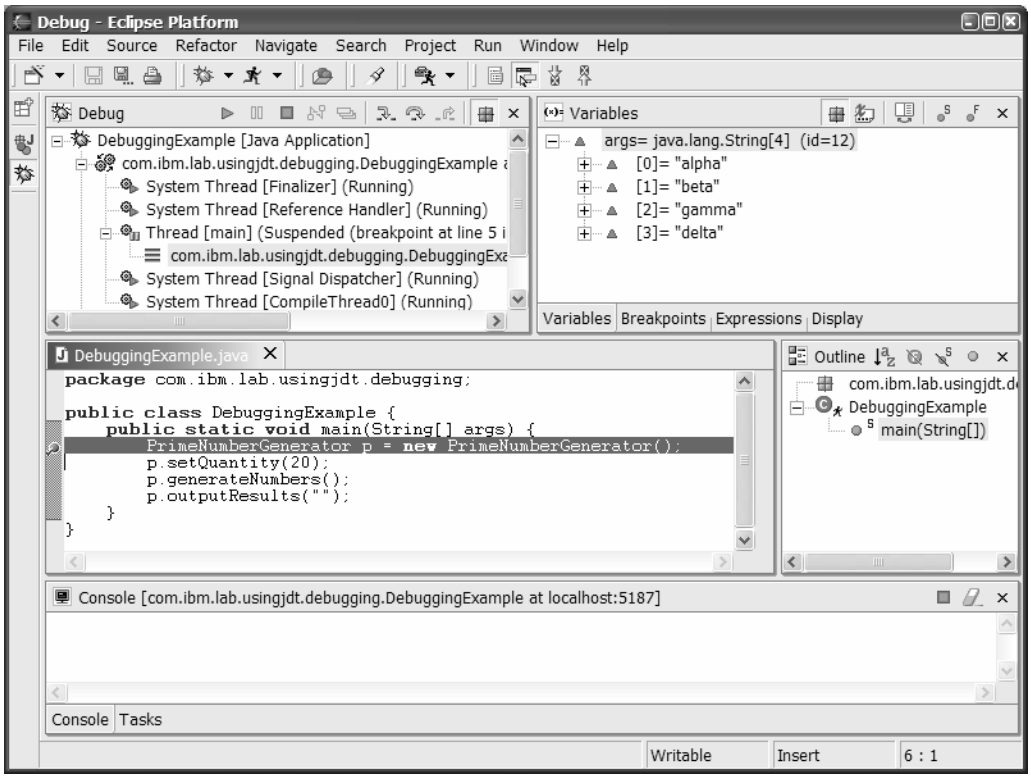


Figure 2-22
Debug Perspective

3. Select Debug Target `com.ibm.lab.usingjdt.debugging.DebuggingExample` in the Debug view and then select **Properties...** from the context menu. You see information on how the debug session was started, including the input parameters we defined in the launch configuration. Select **OK**.

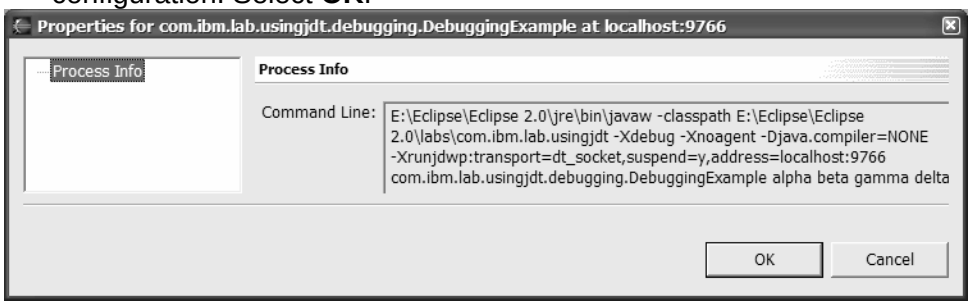


Figure 2-23
Debug Target Properties

4. In the next several steps, we'll look at the debugging commands you use to control program execution.

Select the current stack frame in the Debug view, and then select **Step Over** or press **F6**. One line executes and execution suspends on the next line. Variable p appears in the Variables view.

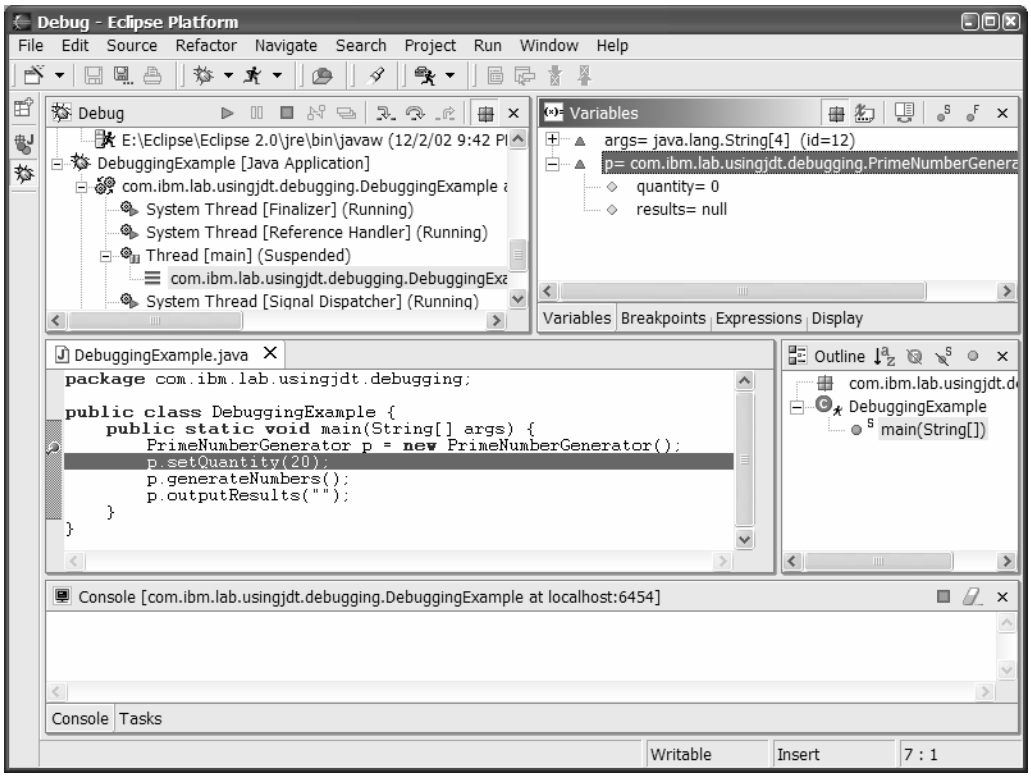


Figure 2-24
Step Over

5. Select **Step Into** or press **F5**. A new stack frame is created for method invocation `setQuantity()`, the editor opens on `NumberGenerator.java`, and execution suspends on the first statement in the method.

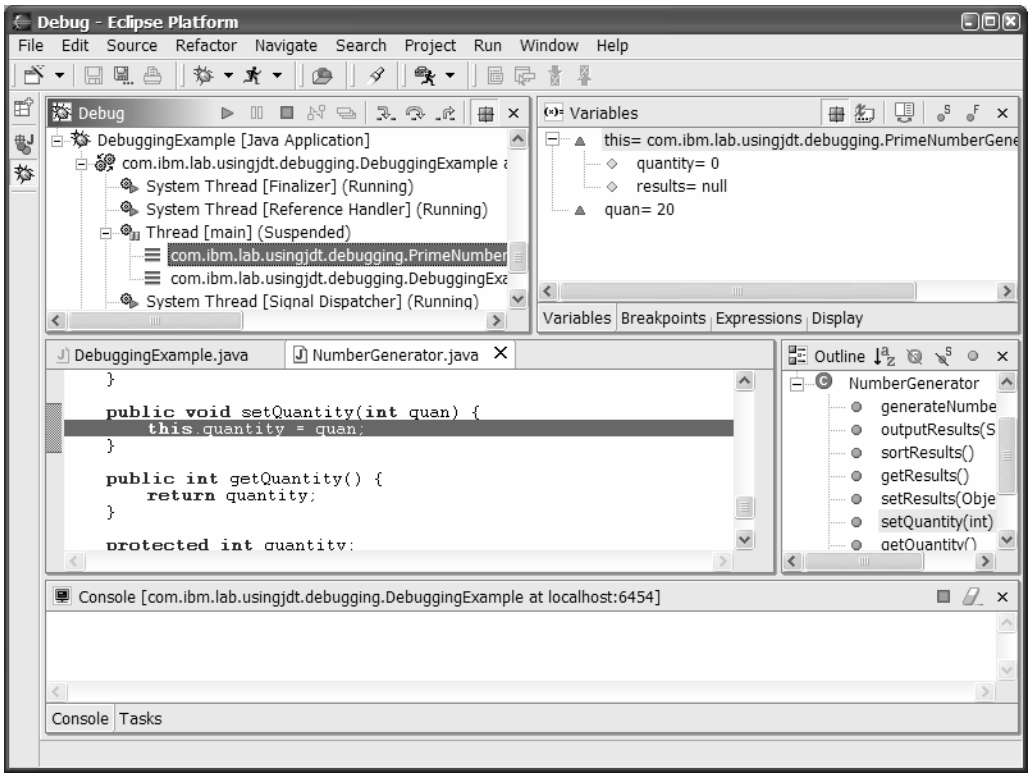


Figure 2-25
Step Into

6. **Step Over** this line and the next to exit the method. The top stack frame is discarded and execution suspends on the statement following the one you just stepped into. Step into `p.generateNumbers()`. Select **Step Return** or press **F7**. Execution resumes to the end of the method in the current stack frame, returns from the method execution, and suspends on p.outputResults(“”), the statement following the method invocation.
7. **Step Into** `p.outputResults()`. Set a breakpoint on the line with the `for` statement and select **Step Return**. Execution suspends at the breakpoint because rather than after the method returns, because the breakpoint is encountered first.
8. **Step over** the `for` statement. Hover the cursor over variable `i`. The value of the variable is displayed:

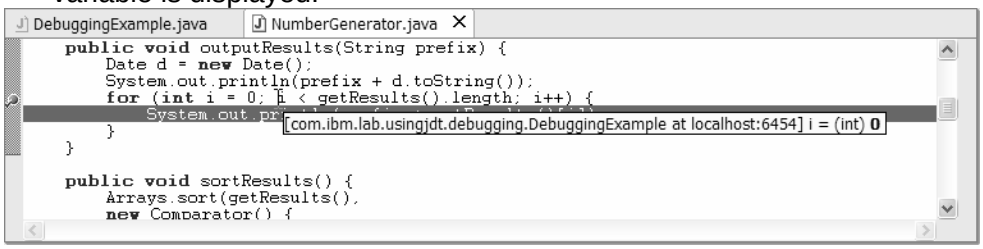


Figure 2-26
Hovering to View a variable value

9. Select **Run to Line** from the context menu or press **Ctrl+R**. Execution resumes and then suspends it on the line that was selected, after an iteration of the loop. You did

not have to define a breakpoint. You can verify this by the increase in the value of variable i.

10. Select **Step Return** to complete method `outputResults`. The output appears in the Console view. Select **Terminate** to stop execution or select **Resume** to continue execution to the end of the program. The status of your program in the Debug view shows it has terminated. Remove the terminated entries in the Debug view with **Remove All Terminated**.

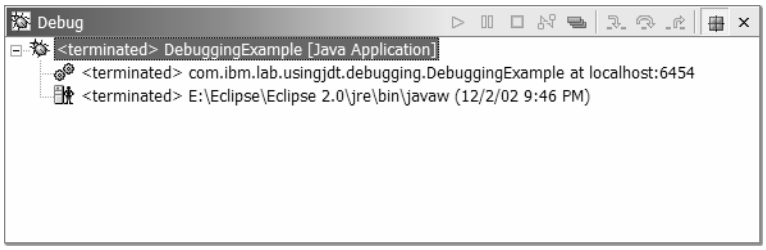


Figure 2-27
Terminated Debug Sessions

Note: This is a good point to consider if you want to return to a previous part of the exercise, or complete this part, given how much time might be left in the allotted lab time.

11. Now let's see how to view and change variable values. Select **Debug** or press **F11** to restart a debugging session on `DebuggingExample`. **Step Over** the first line and then **Step Into** `p.setQuantity(20)`.

12. Switch to the Variables view and select **Show Detail Pane**. Successively select the variables and watch as the values displayed in the detail pane in the bottom of the view. These are the values of the variables' `toString()` methods. You can provide useful debug information if you override of this method in your own classes to display your object's state. Select variable `quan`, select **Change Variable Value** from the context menu, and enter a new value. You can also double click on a value to change it. You cannot change a variable's value from the Detail Pane.

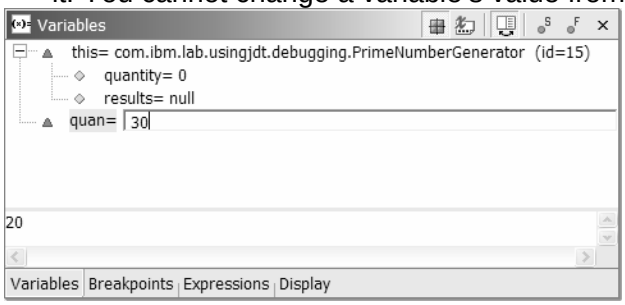


Figure 2-28
Changing a Variable's Value

13. Select **Step Return** then **Step Into** on line `p.generateNumbers()`. In the Variables view, expand `this` and verify field `quantity` has the changed value.

14. With the Variables view visible, **Step Over** lines to continue through an iteration of the `while` loop. The colors of the entries in the Variables view change as values change.

15. Breakpoints can have hit counts; let's see how this works. Set a breakpoint on the second line of the outer `while` loop, "`prime = true;`". From the context menu on

the breakpoint on the marker bar, select **Breakpoint Properties...**. Select to **Enable Hit Count** and set **Hit Count** to 5. Click on **OK**. This will cause execution to suspend the fifth time the breakpoint is encountered.

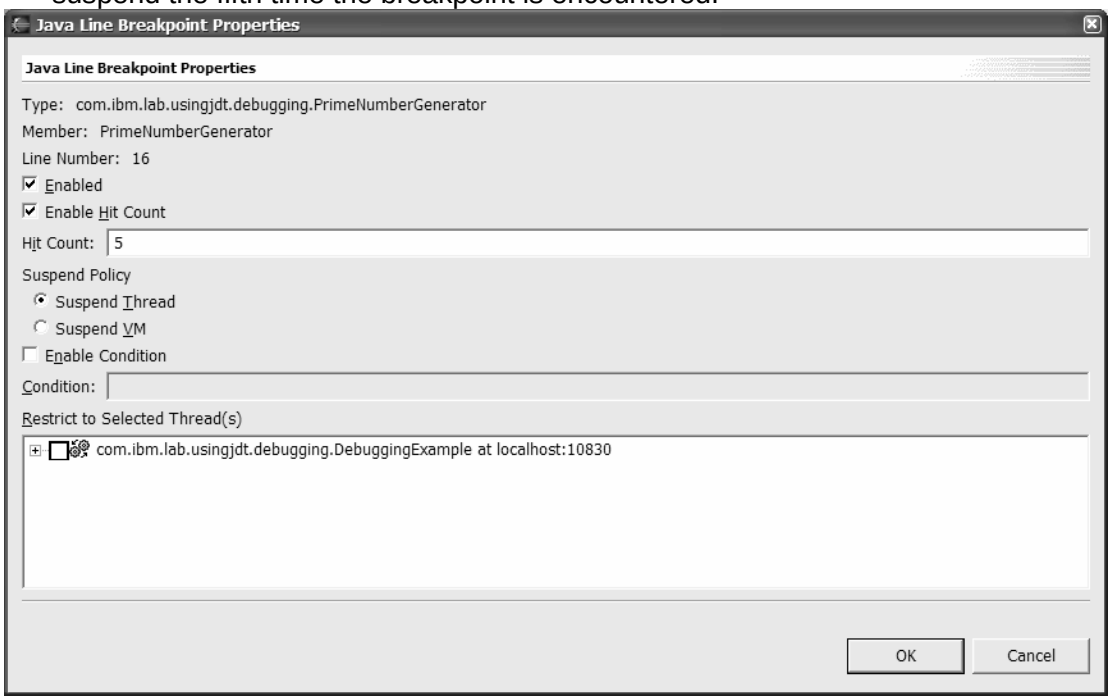


Figure 2-29
Setting a Breakpoint Hit Count

16. Hover over the breakpoint icon to verify its hit count. Hover over variable `candidate` in the editor and remember its current value.

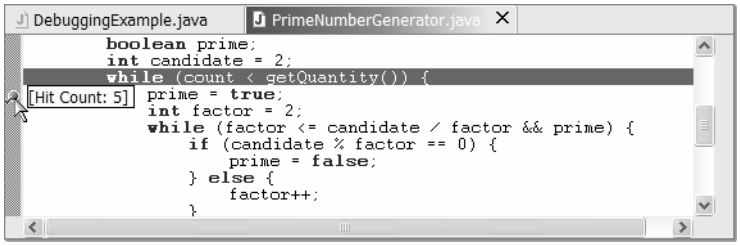


Figure 2-30
Viewing a Breakpoint Hit Count

17. Select **Resume**. Execution resumes and then suspends on the breakpoint after five iterations of the `while` loop. Hover again over variable `candidate` to verify this; its value should be incremented by five, or four depending on where you were in the loop. The breakpoint shows as disabled . Enable it for five more iterations by selecting **Enable Breakpoint** from the context menu. Select **Resume** again. Execution suspends on the same line after another five iterations. Hover over variable `candidate` to verify this.

18. Close the editor on class `PrimeNumberGenerator` and go to the Breakpoints view. You can quickly get back to the source where you've set a breakpoint by double-clicking one in the Breakpoints view. This will open the associated source file and select the line containing the breakpoint. Do this for the disabled breakpoint.

19. Let's change this breakpoint to suspend execution when a condition (Java expression) evaluates to true. In the Breakpoints view, select the disabled breakpoint and then select **Properties...** from the context menu. Check **Enabled**, un-check **Enable Hit Count** and check **Enable Condition** to make this a conditional breakpoint. Enter "`candidate == 40`" for the condition and select **OK**.

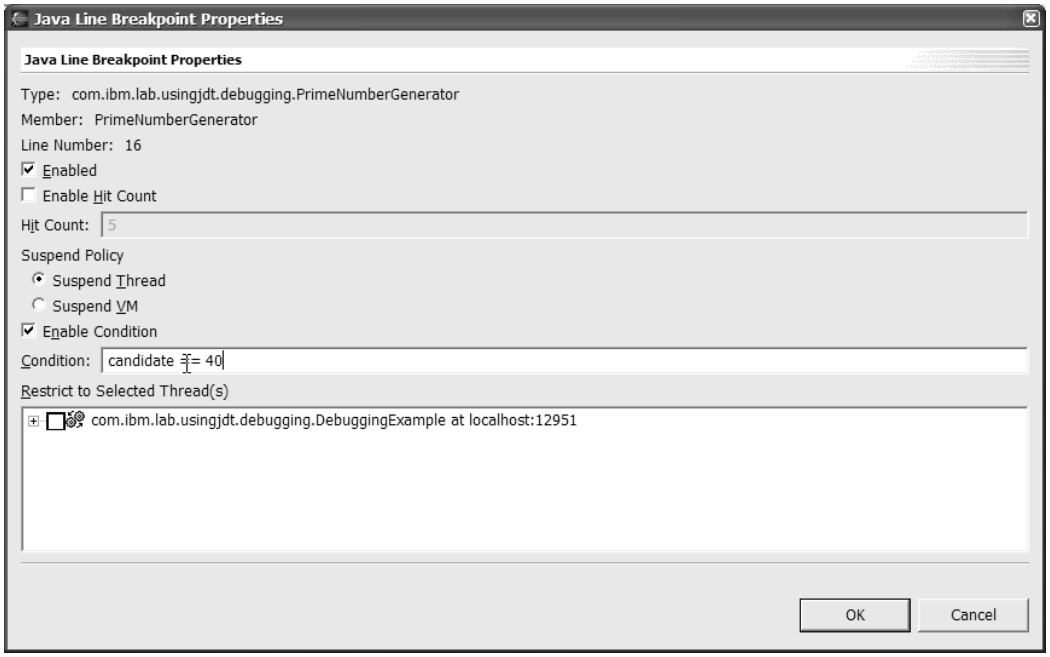


Figure 2-31
Setting a Breakpoint Condition
In the editor, the ? decoration on the breakpoint indicates it is a conditional one. Hover over the breakpoint icon in the editor to see the conditional expression.

20. Select **Resume**. Execution resumes and then suspends. Hover over variable `candidate` in the editor to verify its value is `40`.

21. Finally, let's look at evaluating expressions. **Step Over** lines until you are inside the inner `while` loop on the line, "`if (candidate % factor == 0) {`". In the editor, select expression "`candidate % factor == 0`". From the context menu, select **Display** to evaluate the expression and display the results in the Display view.

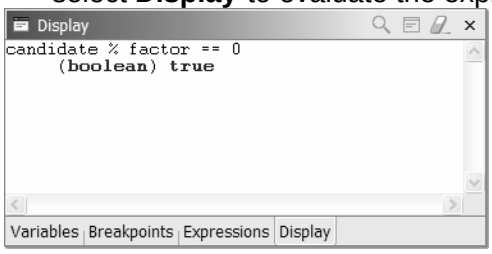


Figure 2-32
Displaying an Evaluated Expression

22. Enter an expression in the Detail pane of the Display view that can be evaluated in the context of the current stack frame, like `numbers.toArray()`. Content assist is available here. Select the expression you entered and then select **Inspect Result of Evaluating Selected Text** to display the results in the Expressions view. Select to display the Detail pane .

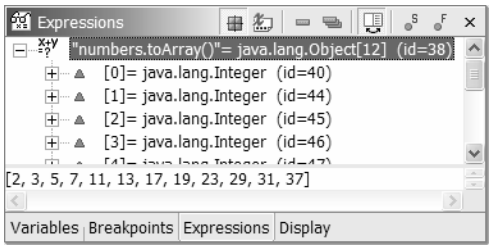


Figure 2-33
Inspecting an Evaluated Expression

23. In the Display view, enter `numbers.get(1)`, select it, then select **Inspect** from the context menu. Another entry is added to the Expressions view with the result.

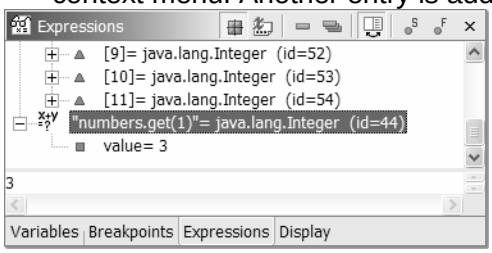


Figure 2-34
Evaluating an Expression in the Detail Pane

24. Expand the `numbers.toArray()` entry in the Expressions view and modify the value of the first entry to a number that is obviously not prime, like `100`. The result of evaluating `numbers.toArray()` returned an array of `Integers`, or more precisely an array of references to the `Integers` of variable `numbers`. In the Expressions view, when you change a value to a referenced object, you change the value in the current stack frame. The point here is that with object references, you are not just changing a value in the Expressions view. While you make the change there, you are actually changing the value of an object in your executing program and (potentially) altering its behavior.
25. Select **Resume** to continue execution to the end of the program. Verify that your changed value appears in the Console view.

When you're through, don't forget to close open editors and terminate and remove existing debugging sessions. Also remove your existing breakpoints.

Exercise Activity Review

At the end of these exercises, you should be able to:

- Write and run a simple "Hello World" program
- Navigate Java errors in JDT and use quick fix to use recommended solutions
- Complete Java expressions and generate Java code with content assist and templates
- Refactor your code

Exercise 18 Workbench JFace Text Editor

Introduction

Exercise Concepts

The objective of this lab is to create a JFace text editor for files with a `.release` file type. The editor will be designed to show content assist, syntax highlighting and content formatter features of JFace Text.

Skill Development Goals

At the end of this lab, you should be able to:

- Add an editor plug-in to the workbench
- Implement a simple JFace Text editor that implements content assist, syntax highlighting and content formatting features.

Exercise Instructions

Common lab setup tasks should have been performed such that a PDE project named `com.ibm.lab.editor` should be available in the PDE perspective.

Part 1: Basic Text Editor Creation

In this part we will create we will create a text editor for files of type `.release`. The platform text editor will be used initially.

1. This requires that we add an editor extension to the `plugin.xml` file.

   To implement this support, open the `plugin.xml` with the Plug-in Manifest Editor. Add the `org.eclipse.ui.editors` extension point to the `plugin.xml` using either the Extensions or Source tab. The xml source for the extension point is as follows:

```
<extension point = "org.eclipse.ui.editors">
  <editor
     id = "com.ibm.lab.ReleaseEditor"
     name="Lab: Release Notes Editor"
     icon="editor.gif"
     extensions="release"
     class="org.eclipse.ui.editors.text.TextEditor" >
  </editor>
</extension>
```

   The editor extension tag of the `plugin.xml` specifies the file extension your editor will work on.

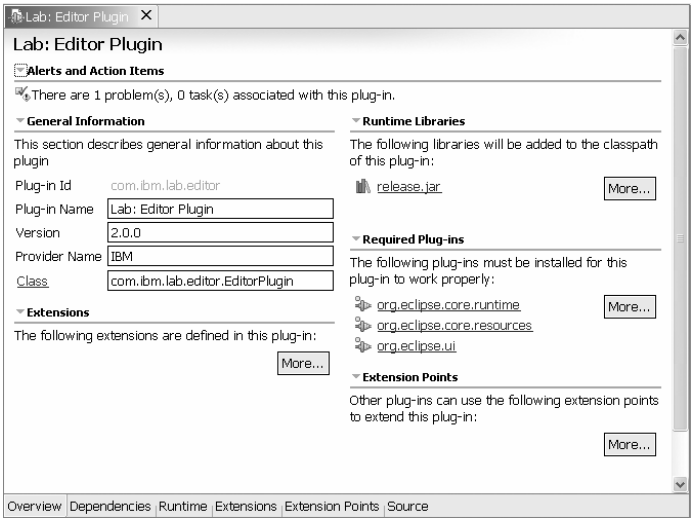


Figure 18-1
Manifest Editor for Editor Plug-in

2. Launch the runtime Workbench making sure the `com.ibm.lab.editor` plug-in is included.
3. The first step is create a file for the editor. If required, create a project and then copy the `test.release` file from the `com.ibm.lab.editor` project to the project in the workspace used by the runtime instance of the Workbench. Use copy/paste or drag and drop to copy the file between workspaces.
4. Open the `test.release` file by double clicking on it in the project root.

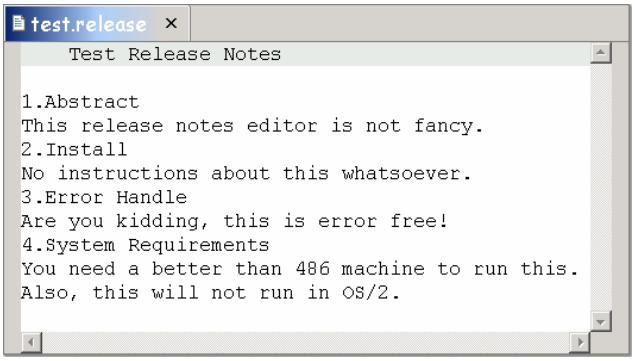


Figure 18-2
Basic Text Editor

Congratulations! You have just created your first text editor.

The `plugin.xml` entry for the editor extension identified the Eclipse platform text editor (**`org.eclipse.ui.editors.text.TextEditor`**) as the editor implementation for the `.release` files. Now it is time to build your own editor implementation.

Part 2: Create Custom Editor

It is now time to create a custom editor implementation that can be enhanced to provide tool specific function.

1. Change the class defined in the editor extension in the `plugin.xml` so that it identifies the custom editor we will implement, the `ReleaseEditor` class. Change the xml for the editor extension to look like this:

```
<extension point = "org.eclipse.ui.editors">
  <editor
     id = "com.ibm.lab.ReleaseEditor"
     name="Lab: Release Notes Editor"
     icon="editor.gif"
     extensions="release"
     class="com.ibm.lab.editor.ReleaseEditor">
  </editor>
</extension>
```


2. Create a `ReleaseEditor` class in the package `com.ibm.lab.editor` using the following options:

   - Extend `org.eclipse.ui.editors.text.AbstractTextEditor`
   - Select the Constructors from superclass option
   - Do not select the Generate the Inherited abstract methods option

Figure 18-3
New Java Class Wizard

Congratulations! You have just created your first custom text editor.

3. Time to take the editor for a test drive. Try the following:

   - Create a bookmark and/or task on the .release file. Is it shown in the editor?
   - Close the editor, and then open the editor by using the Bookmark or Tasks view to find and double-click on the marker created previously. Does the editor open? Is the cursor positioned somewhere in the text?
   - Make a change to the text and select **Edit > Undo** (or press **Ctrl+z**). Is the change backed out? Do you get multiple undo support?
   - Close the editor but do not save the file.

   This is an indication of how the function provided by the `AbstractTextEditor` in an "unconfigured" state. You get basic text editing function, but it needs more work to match the capabilities of the platform text editor.

The `ReleaseEditor` inherits basic text editing capability, but the full JFace text editing framework has more to offer.

Part 3: Create a Document Provider

The capabilities of the `AbstractTextEditor` can be enhanced when the appropriate support is configured. The first step is to add a document provider.

To develop a custom text editor with add-ons, we need a class that produces and manages documents containing textual representation of editor input elements. This class has been provided for you in the template.

Note that the class `ReleaseEditorDocumentProvider` extends `FileDocumentProvider`. In this class, the key method to implement is `createDocument(Object)`. This method is a factory method to create an element's textual representation. We will go back to this method and supply the missing code later.

1. Create a `ReleaseEditorDocumentProvider` class in the package `com.ibm.lab.editor` using the following options:
   - Extend `org.eclipse.ui.editors.text.FileDocumentProvider`
   - Select the Constructors from superclass option
   - Select the Generate the Inherited abstract methods option
2. Use the JDT to generate a default implementation for the `createDocument()` method. Do this by placing the cursor at an appropriate place for a new method in the editor, typing a few characters ("cre") and pressing Ctrl+Space, then select `createDocument(Object element)` from the list and press enter.

Now it is time to connect the document provider to the editor. This can be done in the editor logic or by defining an extension point. We will use the extension point option.

3. Add a document provider extension to the `plugin.xml` file.

   To implement this support, open the `plugin.xml` with the Plug-in Manifest Editor. Add the `org.eclipse.ui.documentProviders` extension point to the `plugin.xml` using either the Extensions or Source tab. The xml source for the extension point is as follows:

```
<extension point="org.eclipse.ui.documentProviders">
   <provider
    extensions="release"
     class="com.ibm.lab.editor.ReleaseEditorDocumentProvider"
     id="com.ibm.lab.editor.ReleaseEditorDocumentProvider">
   </provider>
</extension>
```

The platform will use the document provider for all `.release` files.

Part 4: Implement a Document Partitioner

A document partitioner is used to divide the document into disjoint regions. For our release notes editor, we will partition our document into three kinds of regions or content types:

- Release title
- Header
- Body or section

If you review the content of the `test.release` file in the text editor you have implemented so far you see something like this:

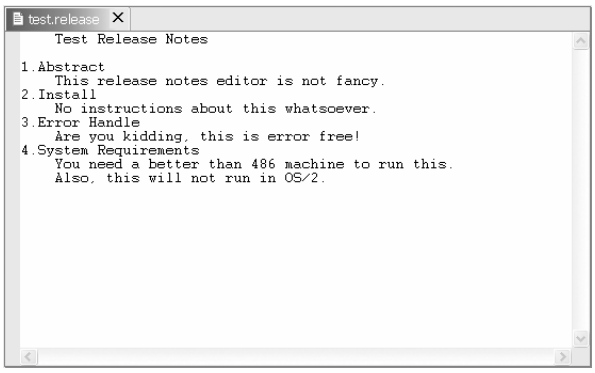


Figure 18-4
Current Editor Status [editor_04.tif]

Based on this view you can see the regions we want to define:

- The release title is: **Test Release Notes**
- The headers are: 1.Abstract, 2.Install, 3.Error Handle, 4.System Requirements
- Sections are below each header – for example, one of the sections is:
  This release notes editor is not fancy.

1. Create a class named `ReleaseEditorPartitioner` that implements the `IDocumentPartitioner` interface in the `com.ibm.lab.editor` package. This document partitioner will be used to divide the document into these disjoint regions.

   Select these options in the New Java Class wizard:

   - Constructors from superclass
   - Inherited abstract methods

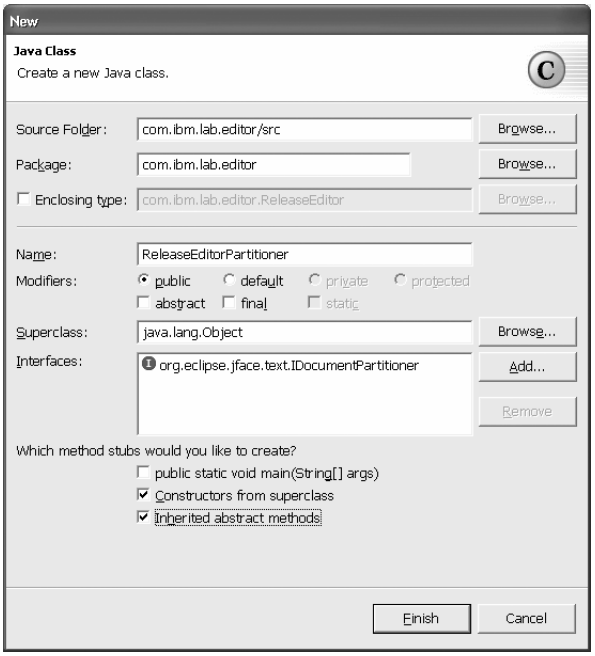


Figure 18-5
New Java Class Wizard [editor_05.tif]

Alternatively, you can wait and add stubs for inherited abstract methods to a class with the Add Unimplemented Methods choice from its context menu in the Outline view, and then create the constructor manually.

You should have nine methods, including the constructor, after the class is correctly generated.

2. Add the following fields to support the predefined set of content types:

```
public final static String RELEASE_DEFAULT =  "__release_default";
public final static String RELEASE_TITLE = "__release_title";
public final static String RELEASE_HEADER = "__release_header";
private IDocument fDocument;
```

3. Create a utility method `getLineEndOffset()` as follows:

```
private int getLineEndOffset(int line, IDocument document)
  throws BadLocationException {

  int length = document.getLineLength(line);
  int start = document.getLineOffset(line);
  return start + length - 1;
}
```


You will use this method later. Use the Organize Imports option to resolve any warnings about missing imports. Make sure you select the JFace package.

4. The method `getPartition(int)` returns the partition containing the given character position of `fDocument`. Instead of returning null, modify the `getPartition(int)` method so that it returns a `TypedRegion`, an indexed text representation consisting of start, length, and content type:

```
return new TypedRegion(0, fDocument.getLength(), RELEASE_DEFAULT);
```

Adjust imports as required.

5. The method `computePartitioning(int, int)` returns the partitioning of the given section of the document.

   The highlighted sections of code identify the function of the method:

   - The first line is defined as the RELEASE_TITLE
   - Any line that starts with a digit is defined as a RELEASE_HEADER
   - Anything else is defined as a RELEASE_DEFAULT

   Modify the method so that it looks like this:

```
public ITypedRegion[] computePartitioning(int offset, int length) {
 List list = new ArrayList();
 try {
    int start, nextOffset;
    boolean isHeader = true;
    int docLength = fDocument.getLength();

       if (offset == 0) {
          nextOffset = getLineEndOffset(1, fDocument);
          list.add(new TypedRegion(0, nextOffset + 1, RELEASE_TITLE));

          int i = 1;
          while (nextOffset + 1 < docLength) {
             start = nextOffset + 1;
             if (Character.isDigit(fDocument.getChar(start)))
                isHeader = true;
             else
                isHeader = false;

             nextOffset = getLineEndOffset(i + 1, fDocument);
             if (isHeader) {
                list.add(new TypedRegion(
                   start, nextOffset - start + 1, RELEASE_HEADER));
             } else {
                list.add(new TypedRegion(
                   start, nextOffset - start + 1, RELEASE_DEFAULT));
             }
             i = i + 1;
          }
```



```
        } else {
           if (Character.isDigit(fDocument.getChar(offset)))
              isHeader = true;
           else
              isHeader = false;

           if (isHeader)
              list.add(new TypedRegion(offset, length, RELEASE_HEADER));
           else
              list.add(new
                   TypedRegion(offset, length, RELEASE_DEFAULT));
        }
     } catch (BadLocationException x) {
  }

  if (list.isEmpty())
     list.add(new TypedRegion(offset, length, null));

  TypedRegion[] result = new TypedRegion[list.size()];
  list.toArray(result);
  return result;
}
```

Adjust the imports selecting the `java.util.List` option.

6. Modify the `connect()` method as follows:

```
public void connect(IDocument document) {
  org.eclipse.jface.util.Assert.isNotNull(document);
  fDocument = document;
}
```

This method is called to connect the partitioner to a document.

7. Modify the `getContentType()` method as follows:

```
public String getContentType(int offset) {
  return IDocument.DEFAULT_CONTENT_TYPE;
}
```

This method is called by a document to obtain the document type that is supported by the partitioner.

8. After each document change the document's partitioning must be updated. Make sure that the document partitioner is set on each document produced by the document provider.

   Go to the `ReleaseEditorDocumentProvider` class and modify the `createDocument(Object)` method to set the document partitioner to your newly created `ReleaseEditorPartitioner`. Modify the method so that it looks like this:


```
protected IDocument createDocument(Object element) throws CoreException
{
  IDocument document = super.createDocument(element);

  if (document != null) {
    IDocumentPartitioner partitioner = new
       ReleaseEditorPartitioner();
    partitioner.connect(document);
    document.setDocumentPartitioner(partitioner);
  }

  return document;
}
```

Part 5: Implement a SourceViewerConfiguration

The capabilities of the `AbstractTextEditor` can be enhanced when the appropriate support is configured. The first step is to create and add the `SourceViewerConfiguration` to the `ReleaseEditor`.

1. Create a class named `ReleaseEditorSourceViewerConfiguration` in the `com.ibm.lab.editor.configuration` package which will be the source-viewer configuration class for the `ReleaseEditor`.

   The source-viewer configuration class is used to enable function such as content-assist and content formatting, and color highlighting. that enables the content-assist function of the editor. When using the wizard to create this class:

   - Extend `org.eclipse.jface.text.source.SourceViewerConfiguration`
   - Choose only the Constructors from superclass option.

2. Add the highlighted code to the `ReleaseEditor` constructor:

```
public ReleaseEditor() {
  super();

  setSourceViewerConfiguration(new
     ReleaseEditorSourceViewerConfiguration());
 setRangeIndicator(new DefaultRangeIndicator());

}
```

   Adjust the imports for the `ReleaseEditorSourceViewerConfiguration` class from the `com.ibm.lab.editor.configuration` package.

The steps that follow will add additional function to the editor by modifying the source-viewer configuration and adding additional code.

Part 6: Add Menu Contributions and Context Menu Options

1. Integrate the editor action contributor

   In the `plugin.xml`, add the `contributorClass` attribute to the editor extension. Add this line:

```
contributorClass="com.ibm.lab.editor.ReleaseEditorActionContributor"
```

   Your editor extension should look like this:

```
<editor
   id = "com.ibm.lab.ReleaseEditor"
   name="Lab: Release Notes Editor"
   icon="editor.gif"
   extensions="release"
   class="com.ibm.lab.editor.ReleaseEditor"
   contributorClass="com.ibm.lab.editor.ReleaseEditorActionContributor">
</editor>
```

   Make sure you move the **>** from the `class=` entry to the `contributorClass=` entry.

   The class `ReleaseEditorActionContributor` sets up an action bar contributor who contributes release notes editor-related actions to the workbench Edit menu. This class has been provided for you. We will now step through the implementation:

   - Go to the constructor `ReleaseEditorActionContributor()` method of the `ReleaseEditorActionContributor` class. The first line in the constructor reads the `EditorPluginResources.properties` file, which was supplied along with the other lab template code in the `com.ibm.lab.editor` project directory.

```
ResourceBundle bundle = EditorPlugin.getDefault().getResourceBundle();
```

     If you want to provide support for NLS translation you should consider using a `.properties` file for your editor. Resource bundles contain locale-specific objects. When your program needs a locale-specific resource, a String for example, your program can load it from the resource bundle that is appropriate for the current user's locale.

     In this way, you can write program code that is largely independent of the user's locale, isolating most, if not all, of the locale-specific information in resource bundles. This allows you to write programs that can be easily localized, or translated into different languages.

   - This line of code will add a content assist entry to the workbench Edit menu:

```
fContentAssistProposal = new RetargetTextEditorAction(bundle,
```



```
  "ContentAssistProposal.");
```

`RetargetTextEditorAction` is the action used by an editor action bar contributor to establish placeholders in menus or action bars which can be retargeted to dynamically changing actions, for example, those which come from the active editor.

The targeted action is initialized by the `setAction(IAction)` method; for an example see the method `setActiveEditor()` in the `ReleaseEditorActionContributor` class.

2. Add a context menu to the `ReleaseEditor`

   We need to implement `createActions()` to add the `TextOperationAction` action entries for content-format, content-assist, content tip.

   This method creates the editor's standard actions and connects them with the global workbench actions. Add this method to the `ReleaseEditor` class:

```
protected void createActions() {
  super.createActions();

  ResourceBundle bundle =
    EditorPlugin.getDefault().getResourceBundle();

  setAction("ContentFormatProposal",
    new TextOperationAction(bundle, "ContentFormatProposal.",
      this, ISourceViewer.FORMAT));

  setAction("ContentAssistProposal",
    new TextOperationAction(bundle, "ContentAssistProposal.",
      this, ISourceViewer.CONTENTASSIST_PROPOSALS));

  setAction("ContentAssistTip",
    new TextOperationAction(bundle, "ContentAssistTip.",
      this, ISourceViewer.CONTENTASSIST_CONTEXT_INFORMATION));
}
```

   Adjust imports as required.

3. Add this method to the `ReleaseEditor` class to set up the editor's context menu before it is made visible:

```
public void editorContextMenuAboutToShow(IMenuManager menu) {
  super.editorContextMenuAboutToShow(menu);

  addAction(menu, "ContentFormatProposal");
  addAction(menu, "ContentAssistTip");
  addAction(menu, "ContentAssistProposal");
}
```


Adjust imports as required.

The `addAction()` invocation above retrieves the `TextOperationAction`, with the action id such as "`ContextAssistProposal`", which was created by the `createActions()`method.

4. Test the editor. The new menu options should be on the context menu and the common edit pull-down menu for the workbench. These actions are not yet enabled as we have not implement their logic.

Part 7: Implement Syntax/Color Highlighting

In this lab exercise, you will be implementing color highlighting for the .release notes editor. We will show keywords in red and words that start with a vowel in green.

To implement syntax highlighting either:

- Define a presentation damager and presentation repairer
- Use the `RuleBasedDamagerRepairer`

Note: If you define a presentation repairer, you need to implement the `createPresentation(TextPresentation, TypedRegion)` method.

In this exercise we will use `RuleBasedDamagerRepairer`.

1. Create a class `ReleaseEditorBodyScanner` that extends `RuleBasedScanner` in the `com.ibm.lab.editor.scanner` package.

   Do not select **Constructors from superclass** or **Inherited abstract methods**.

   This class defines the key words if any that will be highlighted and the rules that are used to define how sequences or patterns of characters are to be styled in the text viewer. For example, rules could be provided to style keywords, links, and so on.

   To customize the class:

   a. Add the following field:

```
private static String[] fgKeywords =
  { "Abstract", "Install", "Error", "Handle", "System", "Requirements"
};
```

   b. Add the following constructor:

```
public ReleaseEditorBodyScanner(ReleaseEditorColorProvider provider) {
  IToken keyword =
    new Token(
    new TextAttribute(
        provider.getColor(ReleaseEditorColorProvider.KEYWORD)));
  IToken defaultText =
    new Token(
        new TextAttribute(
            provider.getColor(ReleaseEditorColorProvider.DEFAULT)));
```



```
  IToken vowel =
    new Token(
        new TextAttribute(
            provider.getColor(ReleaseEditorColorProvider.VOWEL)));

  List rules = new ArrayList();

  // Add generic number rule.
  rules.add(new NumberRule(defaultText));

  // Add generic whitespace rule.
  rules.add(new WhitespaceRule(
        new ReleaseEditorWhitespaceDetector()));

  rules.add(new ReleaseEditorVowelRule(
            new ReleaseEditorVOWELWordDetector(), vowel));

  // Add word rule for keywords, types, and constants.
  WordRule wordRule = new WordRule(
           new ReleaseEditorWordDetector(), defaultText);
  for (int i = 0; i < fgKeywords.length; i++) {
    wordRule.addWord(fgKeywords[i], keyword);
  }

  rules.add(wordRule);

  IRule[] result = new IRule[rules.size()];
  rules.toArray(result);
  setRules(result);
}
```

Use the organize imports option to add any required imports. Remember to choose the java.util.List and JFace options when more than one choice is available.

Note: The `ReleaseEditorColorProvider` class was provided as part of the lab base template code (you don't need to write this bit of the code). This class is used to provide colors for the `.release` editor.

2. Modify `ReleaseEditorSourceViewerConfiguration` to the following variables:

```
private static ReleaseEditorColorProvider fgColorProvider;
private static ReleaseEditorBodyScanner fgBodyScanner;
```

3. Modify the `ReleaseEditorSourceViewerConfiguration()` constructor to add the following lines of code after `super()`:

```
fgColorProvider = new ReleaseEditorColorProvider();
fgBodyScanner= new ReleaseEditorBodyScanner(fgColorProvider);
```


4. Add the `getReleaseEditorBodyScanner()` method to the `ReleaseEditorSourceViewerConfiguration` class:

```
public static RuleBasedScanner getReleaseEditorBodyScanner() {
  return fgBodyScanner;
}
```

5. Override `getPresentationReconciler(ISourceViewer sourceViewer)` in the `ReleaseEditorSourceViewerConfiguration` class:

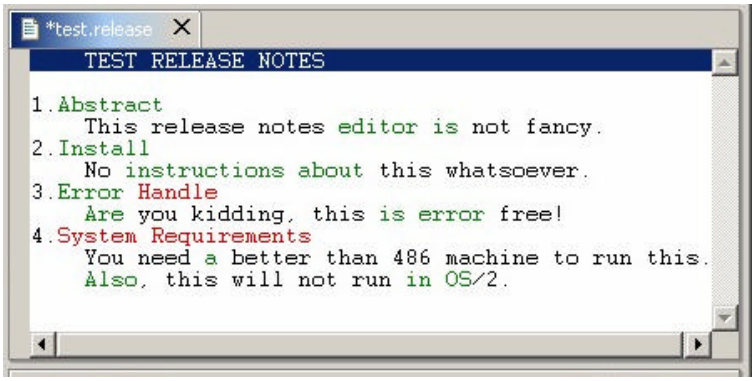
```
public IPresentationReconciler getPresentationReconciler(
   ISourceViewer sourceViewer) {

   PresentationReconciler reconciler = new PresentationReconciler();

   DefaultDamagerRepairer dr = new DefaultDamagerRepairer(
     getReleaseEditorBodyScanner());
   reconciler.setRepairer(dr, "__release_default");
   reconciler.setDamager(dr, "__release_default");

   dr = new DefaultDamagerRepairer(
     getReleaseEditorBodyScanner());
   reconciler.setRepairer(dr, "__release_header");
   reconciler.setDamager(dr, "__release_header");

   dr = new DefaultDamagerRepairer(
    getReleaseEditorBodyScanner());
   reconciler.setRepairer(dr, "__release_title");
   reconciler.setDamager(dr, "__release_title");

   return reconciler;
}
```

6. You are now ready to test your color highlighting code. Start the workbench and edit a .release file.

   Notice that words that begin with a vowel are in green, while those that are keywords are shown in red. The last rule executed wins, so keywords that start with a vowel are green. It is a good idea to version your code again now.

Part 8: Implement a Content Formatter (OPTIONAL)

In this part of the exercise we will add a content formatter to our .release notes editor. The content formatter will capitalize all the letters of the title (the first line of a .release file). The sections in the file will also be indented.



To implement a content formatter:

- Implement your formatting strategy in a class that implements the `IFormattingStrategy` interface.
- Add this strategy as the `ContentFormatter` for the editor by adjusting the `SourceViewerConfiguration.getContentFormatter()` method.
- Make the content-format available in the editor user interface by using a menu item and/or a shortcut key.

1. Create a `ReleaseEditorDefaultStrategy` class in the `com.ibm.lab.editor.configuration` package that implements `IFormattingStrategy`.

   Select both the **Constructors from superclass** and **Inherited abstract methods** options.

2. Create a method `getTabs(int)` that returns a String of tabs:

```
private String getTabs(int num) {
  String result = "";
  for (int i = 0; i < num; i++) {
    result += "\t";
  }
  return result;
}
```

3. Customize the format() method.

   The method `format(String, boolean, String, int[])` formats the content string and adapts the positions according to the reformatting (review the javadoc for the method).

The strategy is told whether the beginning of the content string is a line start. It is also told how the indentation string to be used looks like. Change this method to look like this:

```
public String format(
  String content, boolean isLineStart,
  String indentation, int[] positions) {
    if (content.charAt(0) != '\t')
      return getTabs(1) + content;
      else
      return content;
}
```

4. Review the other formatting classes that were provided in the template.

   The `ReleaseEditorTitleStrategy` and `ReleaseEditorHeaderStrategy` classes were provided to you when you imported the lab base.

   If you review these classes you will see that no special formatting is performed on the header while the title is converted to upper case characters.

5. Add the formatters to the editor.

   To add these formatters to the .release editor, implement the `getContentFormatter(ISourceViewer sourceViewer)` method in the `ReleaseEditorSourceViewerConfiguration` class:

```
public IContentFormatter getContentFormatter(
  ISourceViewer sourceViewer) {
  ContentFormatter formatter = new ContentFormatter();

  IFormattingStrategy titleStrategy = new
    ReleaseEditorTitleStrategy();
  IFormattingStrategy headerStrategy = new
    ReleaseEditorHeaderStrategy();
  IFormattingStrategy defaultStrategy = new
    ReleaseEditorDefaultStrategy();

  formatter.setFormattingStrategy(defaultStrategy, "__release_default");
  formatter.setFormattingStrategy(headerStrategy, "__release_header");
  formatter.setFormattingStrategy(titleStrategy, "__release_title");

  return formatter;
}
```

6. You can now test your code. Start the workbench and open the `test.release` file. Put your mouse pointer on the editor area, open the pop-up menu and select **Content Format**.

Part 9: Implement a simple content assist for the release notes editor (OPTIONAL)

A content-assist processor is responsible for determining a list of proposals. A proposal is a possible completion for the portion of the document for which content-assist was requested. In the simplest case you can provide content assist by:

- Creating the content-assist processor class that is specific to the content to be edited
- Creating a source-viewer configuration class to enable the function in your editor application
- Make the content-assist available in the editor user interface by using a menu item and/or a shortcut key.

In our exercise, the content-assist processor class is provided. We will modify this class so that when you press **CTRL-Space**, or when you select the **Edit > Content Assist** menu item, or when you do a pop-up menu **Content assist** from the release notes area, you will get a list of proposals.

Note that the class `ReleaseEditorCompletionProcessor` implements the interface `org.eclipse.jface.text.contentassist.IContentAssistProcessor`. This class is provided in the `com.ibm.lab.editor.configuration` package.

1. Make a list of proposal items that we want to show when content-assist is invoked. In the `ReleaseEditorCompletionProcessor` class, add a protected final static field named `fgProposals` of type `String[]`:

```
protected final static String[] fgProposals =
  {"Abstract", "Install", "Error", "Handle", "System", "Requirements"};
```

2. Replace the `computeCompletionProposals(ITextViewer, int)` method so that it returns an array of `IContentAssistProposal` instances:

```
public ICompletionProposal[] computeCompletionProposals(
    ITextViewer viewer, int documentOffset) {

  ICompletionProposal[] result = new
      ICompletionProposal[fgProposals.length];
  for (int i = 0; i < fgProposals.length; i++) {
    IContextInformation info = new ContextInformation(
      fgProposals[i], "'" + fgProposals[i] + "' Release Editor popup");

    result[i] = new CompletionProposal(
      fgProposals[i],
      documentOffset,
      0,
      fgProposals[i].length(),
      null,
      fgProposals[i],
      info,
      "Release notes keyword: '" + fgProposals[i] + "'");
  }
  return result;
}
```


When content assist is requested at run time, the following method is called to build the code-assist completion list the `computeProposals()` method in the `CompletionProposalPopup` class is called.

The `computeProposals()` method calls the `computeCompletionProposals(ITextViewer,int)` method in the `ContentAssist` class - passing the viewer whose document is used to compute the proposals and the document position as parameters to build the completion list.

The position is used to determine the appropriate content-assist processor to invoke. The process completes after the `computeCompletionProposals(ITextViewer, int)` method in the `ReleaseEditorCompletionProcessor` class is called.

In this simplified case, the input parameters are ignored and the same list of proposals are returned each time.

3. Enable code assist by adding the completion processor to the editor configuration.

   Override the `getContentAssistant(ISourceViewer)` method in the `ReleaseEditorSourceViewerConfiguration` class:

```
public IContentAssistant getContentAssistant(ISourceViewer sourceViewer)
{

  ContentAssistant assistant = new ContentAssistant();
  assistant.setContentAssistProcessor(
     new ReleaseEditorCompletionProcessor(),
     IDocument.DEFAULT_CONTENT_TYPE);

  assistant.enableAutoActivation(true);
  assistant.setAutoActivationDelay(500);
  assistant.setProposalPopupOrientation(
      IContentAssistant.PROPOSAL_OVERLAY);
  assistant.setContextInformationPopupOrientation(
     IContentAssistant.CONTEXT_INFO_ABOVE);
  return assistant;
}
```

   Adjust the imports as required.

4. Override the `getConfiguredContentTypes(Object)` method inherited from the superclass as follows:

```
public String[]getConfiguredContentTypes(ISourceViewer sourceViewer) {
  return new String[] {
    ReleaseEditorPartitioner.RELEASE_DEFAULT,
    ReleaseEditorPartitioner.RELEASE_HEADER,
    ReleaseEditorPartitioner.RELEASE_TITLE };
} )
```


5. You are now ready to test your content-assist processing:
   a. Start the Workbench using **Run Edit Configurations…** .
   b. Edit the test.release file and make sure that your mouse pointer is over the Test Release Notes editor area, i.e. focus is in the editor area.
   c. Check that **Content Assist** shows on the **Edit** menu. Check that after you select the **Content Assist** menu item, you will get the list of proposals.

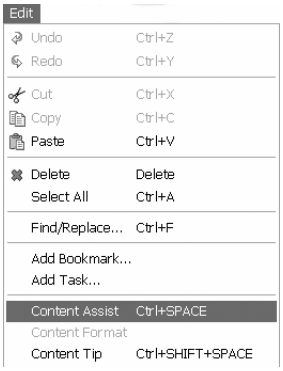


Figure 18-6
Edit Menu with Content Assist Activated *[editor_06.tif]*

   d. Check that **Content Assist** shows on the pop-up menu when you are in the Release Notes editor area.

Figure 18-7
Context Menu with Content Assist Activated *[editor_07.tif]*

   e. Add a 5th header in test.release file. Invoke content assist by pressing **Ctrl+Space**, thru the popup menu or thru the **Edit** menu. Did you get the list of proposals as shown below?

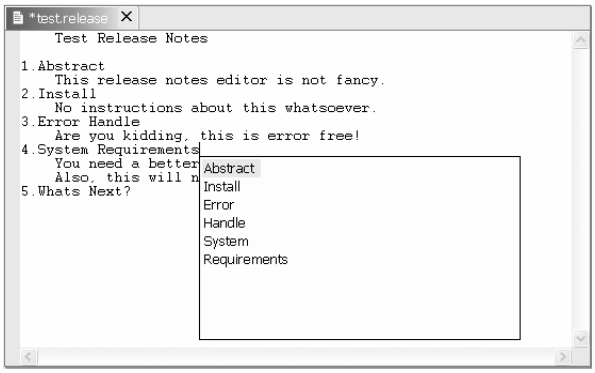


Figure 18-8
Content Assist Menu [editor_08.tif]

Exercise Activity Review

What you did in this exercise:

- Used Workbench extensions to create an editor for files of type .release.
- Implemented content-assist, syntax highlighting and content formatter features in your editor

Eclipse Platform Enablement
D/3ECA
IBM Corporation – RTP, NC

Eclipse Platform Enablement
D/3ECA
IBM Corporation – RTP, NC