Eclipse's Rich Client Platform, Part 1: Getting started

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Before you start

About this tutorial

The first part of a two-part series, this tutorial explores Eclipse's Rich Client Platform (RCP). An example application shows you how to assemble an RCP to create an elegant, client-side interface for your own business applications. The application creates a front end for the Google API and gives you the ability to query and display search results. Having an application that demonstrates some of these technologies in action provides an understanding of the platform and its usefulness within some of your projects.

You should understand how to navigate Eclipse 3.0 and have a working knowledge of Java to follow this tutorial. You do not need a background in Eclipse plug-in development or an understanding of technologies such as the Standard Widget Toolkit (SWT) and JFace. You'll explore each one of these complementary technologies in detail over the course of this tutorial. After a brief introduction to these technologies, the tutorial explores the code and supporting files so you can grasp how to construct an RCP application. If you're new to Eclipse or its complementary technologies, refer to the Resources on page 30 at the end of this tutorial for more information.

Tools

Throughout the series, you'll explore various areas of the Eclipse Plug-In Development Environment in detail. While not a prerequisite, you'll find this tutorial easier to follow if you download, install, and configure Eclipse 3.0, a 1.4 Java Virtual Machine, and Apache Ant. If you don't have these tools installed, please reference, download, and install the following resources:

- Eclipse 3.0 is available at: *http://www.eclipse.org/downloads/index.php*
- Java 2 Standard Edition, Software Development Kit (SDK) is available at: *http://java.sun.com/j2se/1.4.2/download.html*
- Apache Ant 1.6.1 is available at: *http://ant.apache.org/*

About the author

Jeff Gunther, a *Studio B* (http://www.studiob.com/) author, is the General Manager and founder of Intalgent Technologies, an emerging provider of software products and solutions utilizing the Java 2 Enterprise Edition and Lotus Notes/Domino platforms. Jeff is an application and infrastructure architect with experience in architecting, designing, developing, deploying, and maintaining complex software systems. His diverse experience includes full lifecycle development of software running on multiple platforms, from Web servers to embedded devices. You can contact him at: *jeff.gunther@intalgent.com*.

Section 2. Overview of Eclipse and the RCP

The maturation of Eclipse

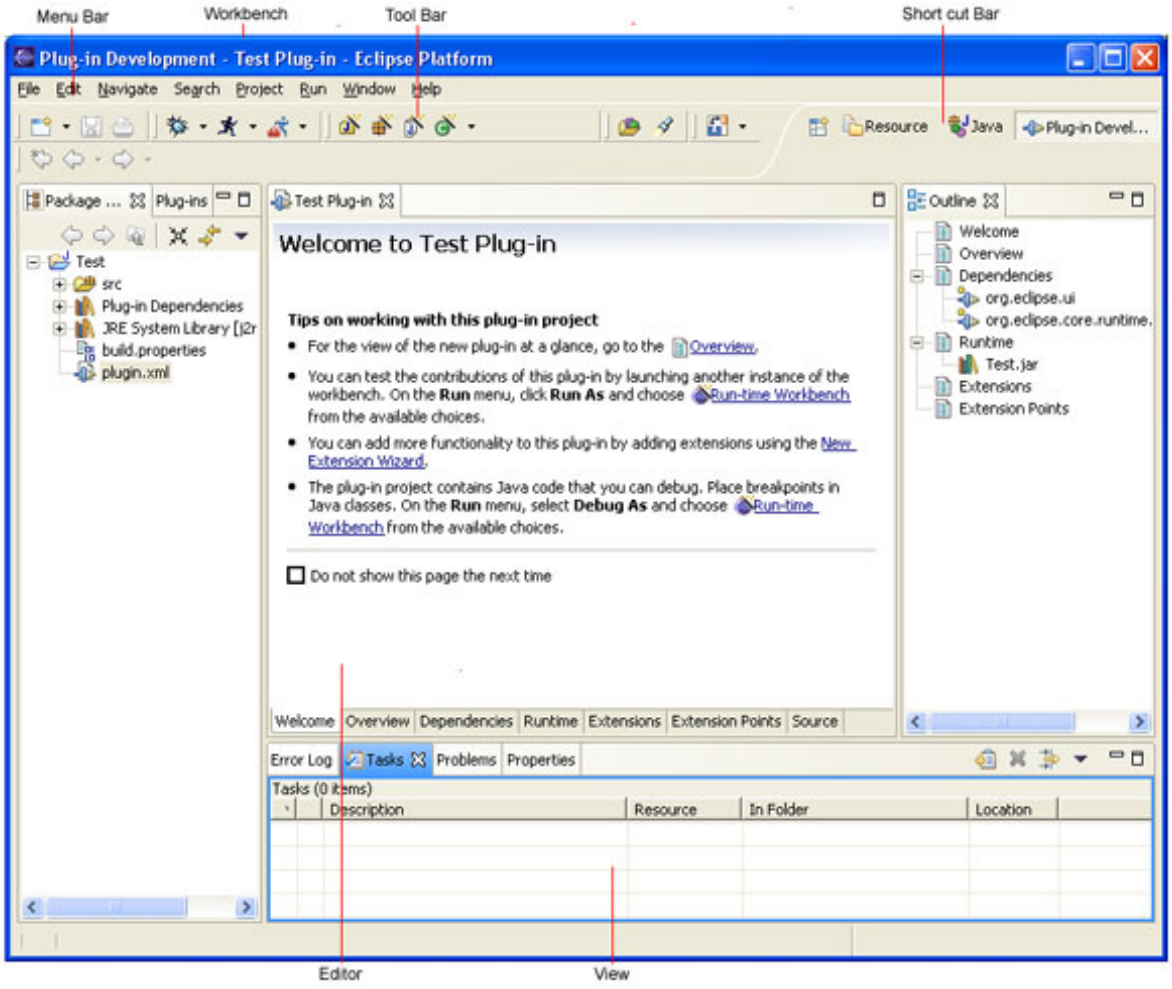
Over the past few years the Eclipse project has grown dramatically and matured into a powerful development environment. While you might traditionally think of Eclipse as an integrated development environment (IDE) for software development, the 3.0 release of Eclipse will broaden the scope of the platform's relevance in the marketplace. A little over a year ago members of the Eclipse community recognized that many elements of the Eclipse IDE could be utilized in non-IDE applications. When constructing business applications, developers could use the elegance of the plug-in architecture, the responsive, native-looking user interface, and the easy-to-use help system. By utilizing a common framework for developing business applications, developers can focus their energies on addressing the specific requirements of their application instead of wasting time reinventing a set of core components. Eclipse 3.0 milestone 5 introduced the development community to the RCP.

What is the RCP?

With the days of the browser wars behind us, many developers and users alike are frustrated with the lack of innovation and advancement of the desktop Web browser. While Web browsers enable organizations to deploy back-office applications to a large number of users, trying to provide a useable interface that supports multiple browsers on multiple operating systems burdens developers and managers. The RCP is an exciting concept that looks to address the need for a single cross-platform environment to create highly-interactive business applications.

Essentially, the RCP provides a generic Eclipse workbench that developers can extend to construct their own applications. An application consists of at least one custom plug-in and uses the same user-interface elements as the Eclipse 3.0 IDE. Before jumping into creating the plug-in, familiarize yourself with the basic elements of the Eclipse user interface, as Figure 1 shows.

Figure 1. The basic elements of the Eclipse user interface



The basic elements of the environment include:

1. **Workbench** -- The overarching container for all windows.
2. **Perspective** -- A visual container for all the opened views and editors.
3. **View** -- A visual container to display resources of a particular type. Typically, a view contains a data grid or tree structure. In Figure 1, the tasks view is an example of a view that is used within the Java perspective.
4. **Short Cut Bar** -- A set of icons that enables the user to quickly access different perspectives.
5. **Menu Bar** -- A set of content-sensitive actions that gives the user the ability to execute some predefined function.
6. **Tool Bar** -- A set of context-sensitive actions that enables the user to execute some predefined function. All the items found within the toolbar appear within the menu bar.
7. **Editor** -- Editors are the primary tool users employ to display and manipulate data. In the case of the Eclipse IDE, developers use an editor to edit Java source files.

Standard Widget Toolkit and JFace

If you look at the source code that makes up the Eclipse Platform, you notice that the Java standard windowing toolkits are not used. During the development of the Eclipse Platform, the project produced two user-interface toolkits that you can use outside of the Eclipse project. These toolkits include:

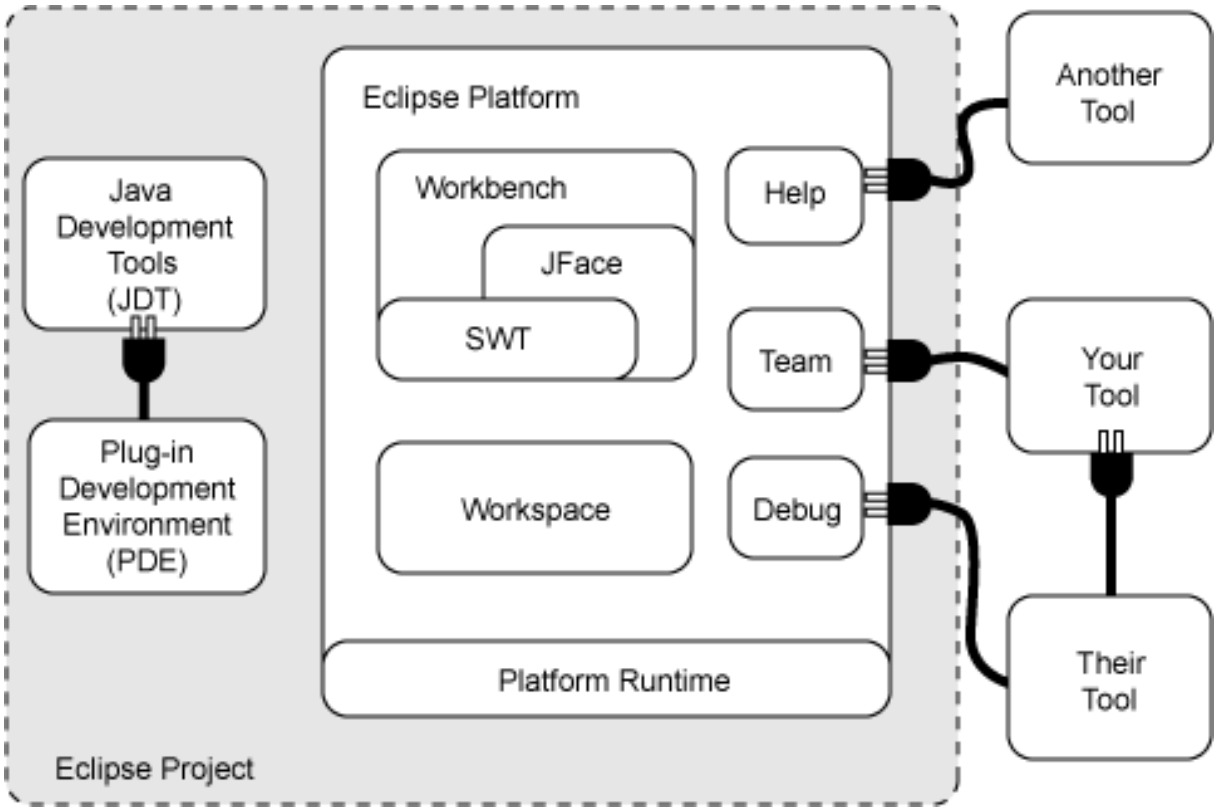
- Standard Widget Toolkit (SWT) -- SWT provides a platform-independent API that is tightly integrated with the operating system's native windowing environment. SWT's approach provides Java developers with a cross-platform API to implement solutions that "feel" like native desktop applications. This toolkit overcomes many of the design and implementation trade-offs that developers face when using the Java Abstract Window Toolkit (AWT) or Java Foundation Classes (JFC).
- JFace -- The JFace toolkit is a platform-independent user interface API that extends and interoperates with the SWT. This library provides a set of components and helper utilities that simplify many of the common tasks in developing SWT user interfaces. This toolkit includes many utility classes that extend SWT to provide data viewers, wizard and dialog components, text manipulation, and image and font components.

During the development of an RCP application you extensively use the SWT and JFace classes. Refer to the Resources on page 30 for more information about these two toolkits.

Eclipse plug-in architecture

Through the influence of its predecessor, IBM Visual Age for Java, architects made the Eclipse Platform easily extensible. Figure 2 below illustrates the major components of the Eclipse architecture.

Figure 2. The major components of the Eclipse architecture



Outside of the base files that make up the Eclipse Platform runtime, all of Eclipse's functionality is implemented through the use of plug-ins. A plug-in is the base building block that developers use to add new capabilities and functionality to the environment. (See Resources on page 30 for an excellent developerWorks article on *Developing Eclipse plug-ins*.) The Eclipse runtime is responsible for managing the lifecycle of a plug-in within a workbench. All of the plug-ins for a particular environment are located in a plugin folder within the directory structure of an RCP application. Upon execution, the Eclipse runtime will discover all of the available plug-ins and use this information to create a global plug-in registry.

For a plug-in to participate within the workbench, it must define a set of extensions. An extension can add functionality directly to the base generic workbench, or extend other existing extensions. Each of these extensions is defined within a plug-in's manifest file. This XML file describes how all the extensions interoperate within the Eclipse runtime and defines the necessary dependencies. The next section covers the plug-in manifest and its tags in more detail.

Section 3. Getting started with the RCP

Steps to implement an RCP application

Before covering the specifics of developing an RCP application within Eclipse, review the general steps for implementing this type of project.

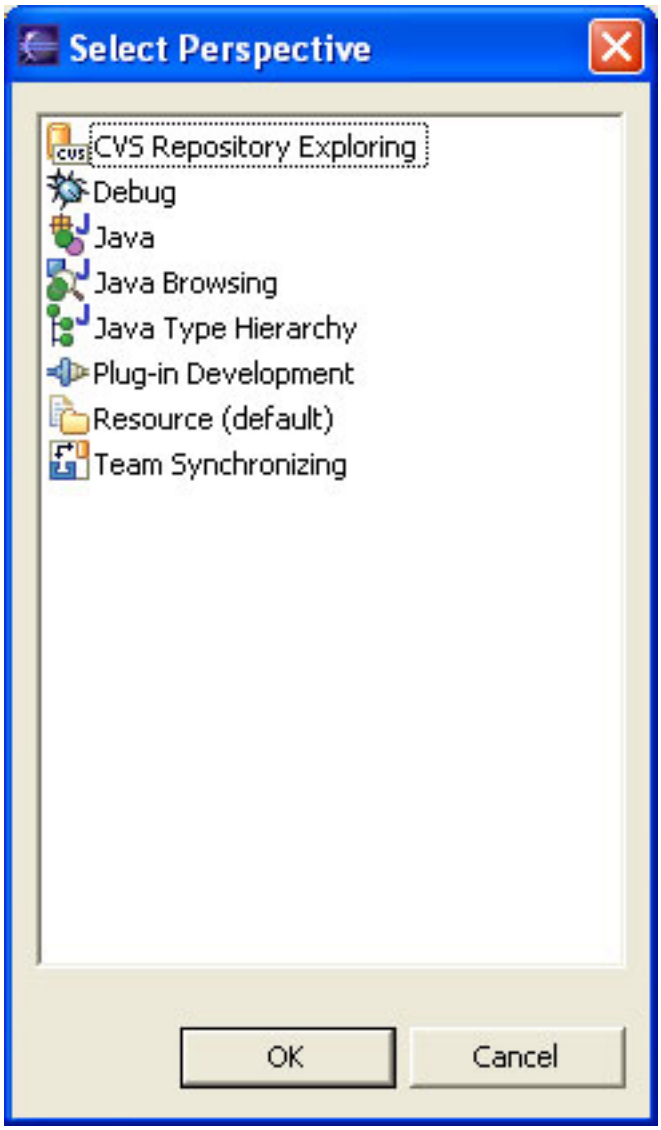
1. Identify extension points
2. Define the plug-in manifest
3. Implement extensions
4. Define the `WorkbenchAdvisor` class
5. Define the `Application` class
6. Export the application

This section shows how to access the Plug-in Development Environment and discusses the plug-in manifest.

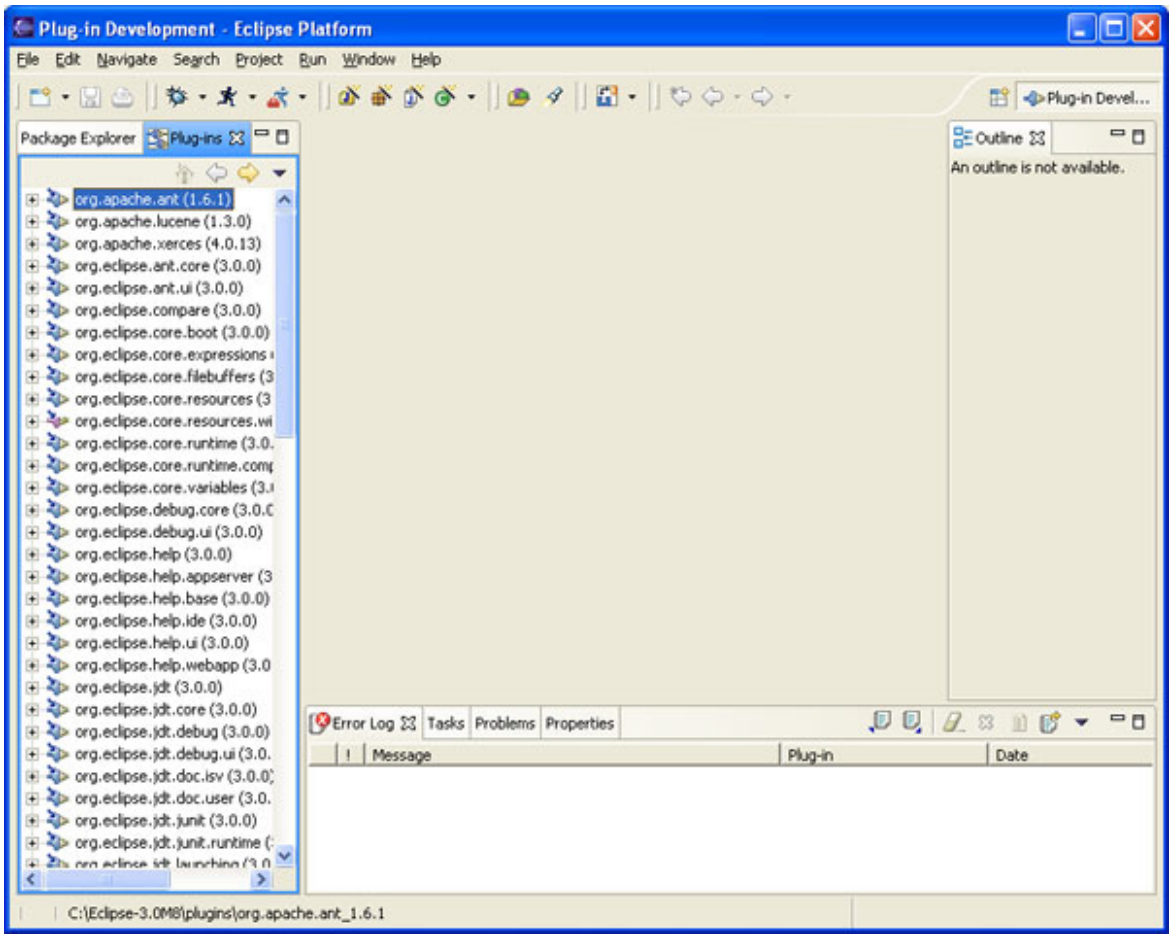
Using the Plug-in Development Environment

One of the components of the Eclipse IDE is a specialized perspective called the Plug-in Development Environment (PDE). This perspective provides everything you need to create and package a custom Eclipse plug-in or RCP application. Access this perspective by completing the following steps:

1. Launch Eclipse 3.0 from your workstation.
2. Select **Window > Open Perspective > Other** from the menu bar. This action will prompt you with the Select Perspective dialog, as Figure 3 shows:
   Figure 3. The Select Perspective dialog



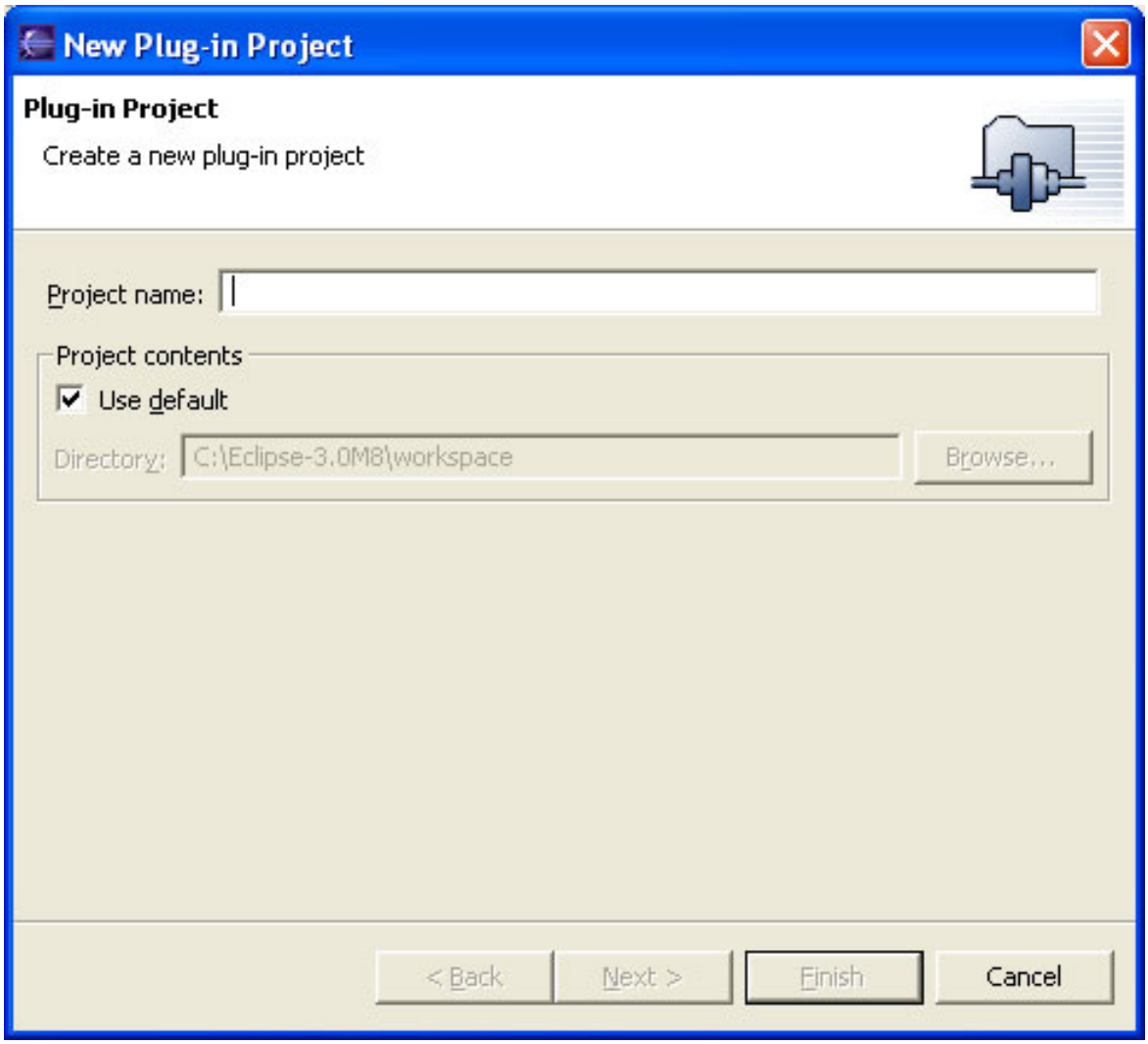
3. Choose **Plug-in Development** from the list of perspectives and then click **OK** to display the PDE perspective Figure 4 shows:
   Figure 4. The PDE perspective



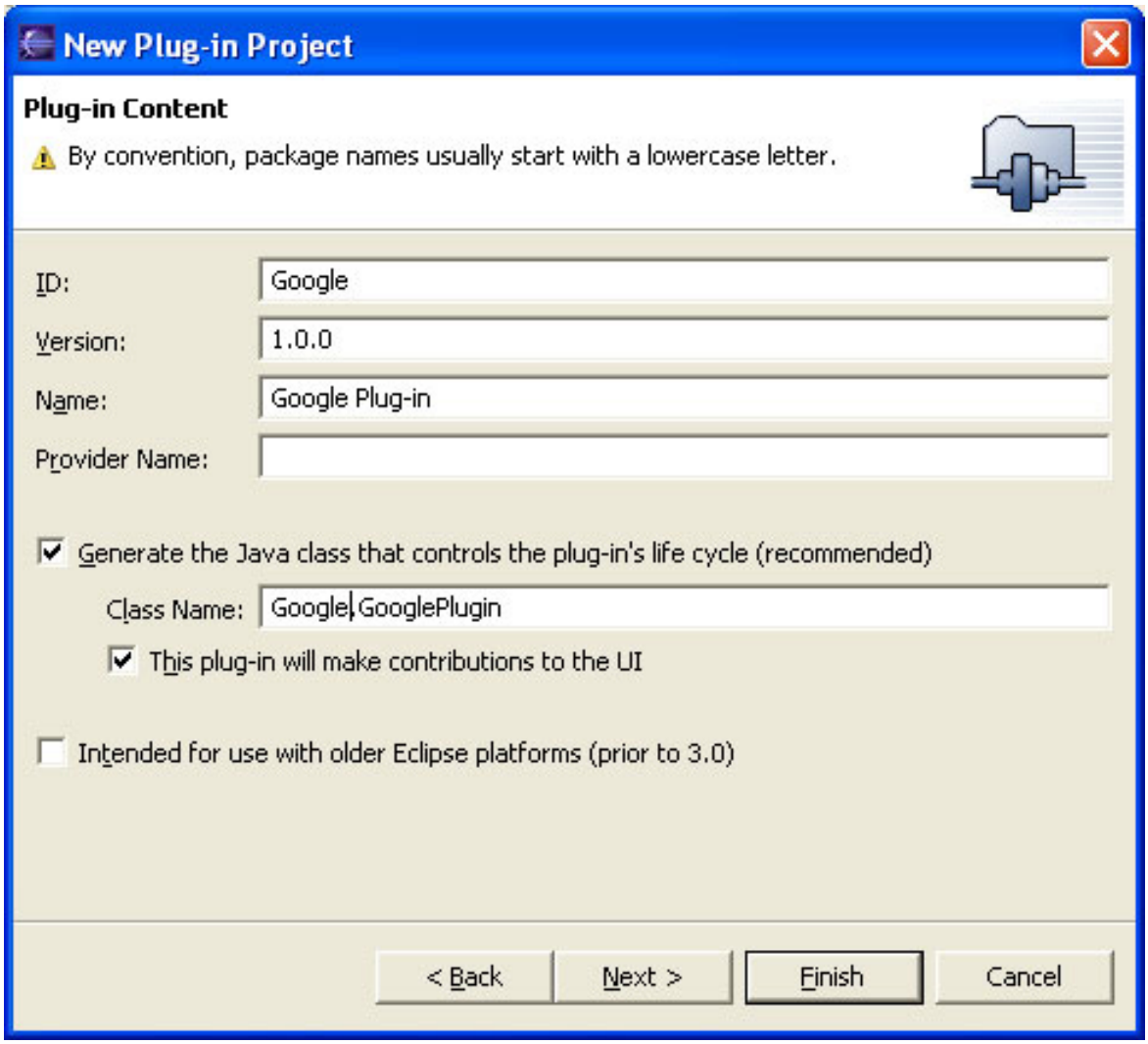
Creating the project

With the PDE perspective opened in Eclipse, complete the following steps to create a new project:

1. Select **File > New > Plug-in Project** from the menu bar to display the New Plug-in Project wizard Figure 5 shows:
   Figure 5. The New Plug-in Project wizard



2. Type **Google** into the Project name field.
3. Keep the defaulted values for this page and click **Next** to continue to the Plug-in Content page Figure 6 shows:
   Figure 6. The Plug-in Project Content page



4. Type **com.ibm.developerworks.google.GooglePlugin** into the Class Name field and click **Next** to continue to the Templates page.
5. Keep the defaulted values for the Templates page and click **Finish**.

Understanding the plug-in manifest

After you've completed the New Plug-in Project wizard, a project called Google will be added to the Packages Explorer and you'll be presented with a page entitled "Overview" as Figure 7 shows.

Figure 7. Welcome to Google Plug-in

This page is a powerful tool for editing the generated plug-in manifest. A plug-in manifest is responsible for defining the resources, dependencies, and extensions the Eclipse runtime will manage. The plug-in manifest for any project is located within the project's root directory and is called plugin.xml. Each tab across the bottom of this editor provides you with an easy way to access and manipulate a particular section of this file.

The plugin.xml tab allows you to view the XML that each section of the editor generates. For example, below you see the content of the plug-in manifest that the New Plug-in Project wizard initially generates.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin
   id="Google"
   name="Google Plug-in"
   version="1.0.0"
   provider-name=""
   class="com.ibm.developerworks.google.GooglePlugin">
```

```
        <runtime>
               <library name="Google.jar">
                        <export name="*"/>
               </library>
        </runtime>
        <requires>
               <import plugin="org.eclipse.ui"/>
               <import plugin="org.eclipse.core.runtime.compatibility"/>
        </requires>

</plugin>
```

During this discussion, you primarily use the plugin.xml view to edit the plug-in manifest. Although the editor is a helpful tool for learning the structure of a plug-in manifest, you must understand the tags it generates and how they contribute to the overall plug-in. The next two panels review each tag of a plug-in manifest and explain its purpose.

Using the plug-in manifest tags

In order to create a basic RCP application, you need to add some additional content to the plug-in manifest. Using the plugin.xml tab of the plug-in manifest editor, modify the XML within the editor to reflect the following changes:

```
1       <?xml version="1.0" encoding="UTF-8"?>
2       <?eclipse version="3.0"?>
3       <plugin
4          id="com.ibm.developerworks.google"
5          name="Google Plug-in"
6          version="1.0.0"
7          provider-name=""
8          class="com.ibm.developerworks.google.GooglePlugin">
9
10              <runtime>
11                     <library name="Google.jar">
12                            <export name="*"/>
13                     </library>
14              </runtime>
15
16              <requires>
17                     <import plugin="org.eclipse.ui"/>
18                     <import plugin="org.eclipse.core.runtime"/>
19               </requires>
20
21              <extension id="googleApplication"
22                     point="org.eclipse.core.runtime.applications">
23                     <application>
24                            <run class="com.ibm.developerworks.google.GoogleApplication"/>
25                     </application>
26              </extension>
27
28              <extension point="org.eclipse.ui.perspectives">
29                     <perspective
30                            ="com.ibm.developerworks.google.GooglePerspective"
31                     name="Google"
32                            class="com.ibm.developerworks.google.GooglePerspective"/>
33              </extension>
```

```
34
35        </plugin>
```

Next, you'll explore this plug-in manifest in detail.

Stepping through the plug-in manifest

Beginning with the `<plugin>` element, lines 3 through 8 start defining the body of the plug-in manifest. This base tag contains all the extensions, extension points, dependencies, and runtime constraints of the plug-in. In addition, the `<plugin>` tag has the following five attributes:

1. `name` -- This attribute defines the general name of the plug-in.
2. `id` -- This attribute defines a unique identifier for the plug-in. To reduce any naming collisions, you should derive this attribute from the Internet domain of the plug-in's author. In this example, the id for this plug-in has been changed to `com.ibm.developerworks.google`. This practice is consistent with other Java naming conventions like class packaging.
3. `version` -- This attribute defines the plug-in version in a major.minor.service format.
4. `provider-name` -- This attribute defines the author of this plug-in.
5. `class` -- This attribute defines the name of the plug-in class. Although a plug-in class is defined, an RCP application does not use this class during execution.

Lines 10 through 14 define the runtime section of the plug-in manifest. Similar to the concept of a classpath within a Java application, this section defines any local Java libraries that are necessary during execution. Each Java library is listed within the `<runtime>` element by using a `<library>` element. The library element can contain a series of nested `<export>` elements. Each export element defines the export mask for that particular library.

Lines 16 through 19 contain a `<requires>` element that defines any dependencies on other plug-ins. Each plug-in is itemized through the use of a single `<import>` element.

Lines 21 through 37 define two `<extension>` elements that the RCP application will use. The next panel reviews the basic concepts of extensions and extension points. The `<extension>` element has the following three attributes:

1. `point` -- This attribute defines a reference to an extension point being configured.
2. `id` - This optional attribute defines an identifier for this extension point configuration instance.
3. `name` - This optional attribute defines a general name for this extension.

Understanding extensions

As previously mentioned in the Eclipse plug-in architecture on page 5 panel, the Eclipse Platform is extremely extensible through the use of a relatively small runtime

kernel and its elegant plug-in architecture. The use of plug-ins adds new functionality to the runtime kernel. Each plug-in can contain any number of extensions that are integrated through the use of extension points. Similarly, a plug-in can define its own set of extension points that other developers can utilize within their own plug-ins or RCP applications.

Examine the two `<extension>` elements of the previously presented plug-in manifest.

```
1 ...
2              <extension id="googleApplication"
3                      point="org.eclipse.core.runtime.applications">
4                      <application>
5                             <run class="com.ibm.developerworks.google.GoogleApplication"/>
6                      </application>
7              </extension>
8
9              <extension point="org.eclipse.ui.perspectives">
10                      <perspective
11                             id="com.ibm.developerworks.google.GooglePerspective"
12                             name="Google"
13                             class="com.ibm.developerworks.google.GooglePerspective"/>
14             </extension>
15 ...
```

Lines 2 through 7 define the first extension through the `org.eclipse.core.runtime.applications` extension point. This extension declares the entry point for an RCP application. Within this extension element, an `<application>` element is defined. A `<run>` element is within this tag. This `<run>` element contains the class name that will be executed when the RCP application is started. The second extension is between lines 10 through 17. This extension defines a perspective through an extension point entitled `org.eclipse.ui.perspectives`. This extension point adds perspectives to the generic workbench. The next section explores the use of perspectives in more detail.

For more information about the various types of extension points that come with Eclipse 3.0, refer to the Resources on page 30.

Section 4. Defining a perspective

Overview of perspectives

Perspectives within the Eclipse workbench are a visual container for all opened views and editors. In the previous panel, Using the Plug-in Development Environment on page 7, you opened a specialized perspective called the PDE to start the Google plug-in project. This perspective is specifically designed to provide developers with a set of tools to develop custom plug-ins. End users of the perspective can see that the creators of the PDE paid a lot of attention to the location and placement of the tools within the workbench. As you begin the process of creating perspectives within your own RCP applications, take into account the following considerations:

1. *Define the perspective's purpose* -- Since the Eclipse workbench only displays a single perspective at a time, you want to group logical and functional areas of your application into a unified perspective. This approach minimizes the need for the user to toggle between different perspectives to accomplish a particular task. As you work through and define each perspective's purpose, also keep in mind that a view or editor cannot be shared between different perspectives. The number of perspectives that any application will have is largely dependent on the application's size and complexity. For our example Google application, only one perspective is initially defined.
2. *Define the perspective's behavior* -- Depending on your application, a perspective with its collective views, editors, and actions can be designed to perform distinct functions. For example, the Java Browsing perspective within Eclipse 3.0 is designed to provide you various types of information that are filtered based on a set of selection criteria. This perspective's behavior filters information for you using a series of consecutive views. In contrast, the Java perspective is a collection of views, editors, and actions that give you the ability to edit and compile Java code. This perspective's behavior is task-oriented and gives the end user a set of tools to accomplish a particular goal.

Creating a basic perspective

After creating your plug-in project, creating a perspective is a two-step process. First, modify the plug-in manifest to include a new extension that uses the `org.eclipse.ui.perspectives` extension point. Second, using the attributes from the new extension point, create a perspective class. Based on the earlier discussion of extensions and extension points, the plug-in manifest for the Google application already includes the following extension:

```
...
<extension point="org.eclipse.ui.perspectives">
       <perspective
              id="com.ibm.developerworks.google.GooglePerspective"
              name="Google"
              class="com.ibm.developerworks.google.GooglePerspective"/>
</extension>
...
```

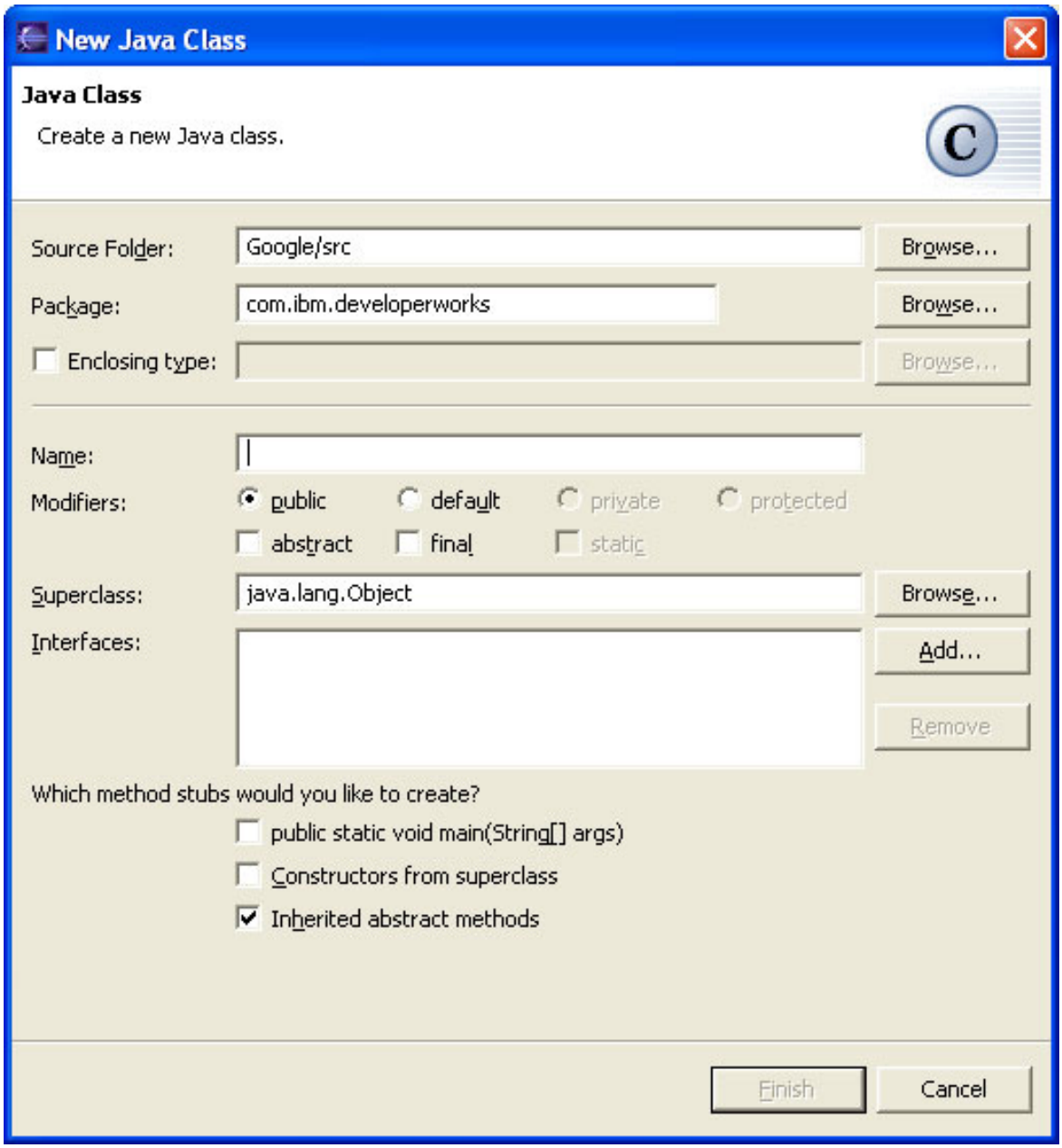
The `<perspective>` element has the following attributes:

- `id` -- This attribute defines a unique identifier for the perspective.
- `name` -- This attribute defines a name for this perspective, and the workbench window menu bar uses it to represent this perspective.
- `class` -- This attribute contains the fully qualified name of the class that implements the `org.eclipse.ui.IPerspectiveFactory` interface.

Creating a basic perspective, continued

To create the perspective class within the Google project, complete the following steps:

1. Select **File > New > Class** from the menu bar to display the New Java Class wizard.
   Figure 8. New Java Class wizard



2. Type **GooglePerspective** into the Name field.
3. Click on the **Add** button to display the Implemented Interfaces Selection dialog box.
4. Type **org.eclipse.ui.IPerspectiveFactory** into the Choose Interfaces field and click **OK**.
5. Click the **Finish** button to create the new class.

The wizard generates the following source code:

```
1       package com.ibm.developerworks.google;
2
3       import org.eclipse.ui.IPageLayout;
4       import org.eclipse.ui.IPerspectiveFactory;
```

```
5
6       public class GooglePerspective implements IPerspectiveFactory {
7
8               public void createInitialLayout(IPageLayout layout) {
9
10              }
11      }
```

The `createInitialLayout` method found on lines 8 through 10 defines the initial layout of all the views and editors within the perspective. For the time being, you don't need to modify this method. You'll modify it in Part two of this series once you define a view.

Section 5. Defining the WorkbenchAdvisor and Application classes

Introducing the WorkbenchAdvisor

The previous panels have focused on the various components that contribute to an RCP application. The next series of panels focus on pulling everything together. One of the core tasks in constructing an RCP application is to create a class that implements the abstract class `org.eclipse.ui.application.WorkbenchAdvisor`. The `WorkbenchAdvisor` class is responsible for configuring the workbench that displays when an RCP application executes.

The `WorkbenchAdvisor` class contains the following methods that provide developers access to the lifecycle of the generic workbench:

- `initialize` -- This method is called first before any windows are displayed.
- `preStartup` -- This method is executed second, but is called before the first window is opened. This method is useful to temporarily disable items during startup or restore.
- `postStartup` -- This method is called third after the first window is opened and is used to re-enable items temporarily disabled in the `preStartup` method.
- `postRestore` -- This method is called after the workbench and its windows have been recreated from a previously-saved state.
- `preShutdown` --This method is called just after the event loop has terminated, but before any windows have been closed.
- `postShutdown` --This is the final method that is called after the event loop has terminated.

The `WorkbenchAdvisor` class contains the following methods that provide developers access to the lifecycle of the workbench window:

- `preWindowOpen` -- This method is called as each window is opened.
- `fillActionBars` -- This method is called after the `preWindowOpen` method, and it configures a window's action bars.
- `postWindowRestore` -- This method is called after a window has been recreated from a previously-saved state.
- `postWindowOpen` -- This method is called after a window has been opened. This method is useful to register any window listeners.
- `preWindowShellClose` -- This method is called when the user closes the window's shell.

The `WorkbenchAdvisor` class contains the following methods that provide developers access to the event loop of the workbench:

- `eventLoopException` -- This method is called to handle the exception of the event loop crashing.
- `eventLoopIdle` -- This method is called when no more events need to be processed.

Creating the WorkbenchAdvisor class

To create a `WorkbenchAdvisor` class, complete the following steps from within the PDE:

1. Select **File > New > Class** from the menu bar to display the New Java Class wizard.
2. Type **GoogleWorkbenchAdvisor** into the Name field.
3. Click on the **Browse** button to display the Superclass Selection dialog box.
4. Type **org.eclipse.ui.application.WorkbenchAdvisor** into the Choose a type field and click **OK**.
5. Click the **Finish** button to create the new class.

The wizard generates the following source code:

```
1       package com.ibm.developerworks.google;
2
3       import org.eclipse.ui.application.WorkbenchAdvisor;
4
5
6       public class GoogleWorkbenchAdvisor extends WorkbenchAdvisor {
7
8              public String getInitialWindowPerspectiveId() {
9
10                    return null;
11             }
12      }
```

You need to make a few minor modifications to this class before you try to execute the RCP application within the PDE. First, you need to modify the `getInitialWindowPerspectiveId` method on lines 7 through 9. This method should return the identifier of the initial perspective for the new workbench window. Since you defined the Google perspective in the previous section as `com.ibm.developerworks.GooglePerspective`, this string will be returned to the calling function. Second, you need to add a method called `preWindowOpen`. This method allows you to set the workbench's window title and size. See the modified class below:

```
package com.ibm.developerworks.google;

import org.eclipse.swt.graphics.Point;
import org.eclipse.ui.application.IWorkbenchWindowConfigurer;
import org.eclipse.ui.application.WorkbenchAdvisor;

public class GoogleWorkbenchAdvisor extends WorkbenchAdvisor {

      public String getInitialWindowPerspectiveId() {
            return "com.ibm.developerworks.google.GooglePerspective";
      }

      public void preWindowOpen(IWorkbenchWindowConfigurer configurer) {
               super.preWindowOpen(configurer);
               configurer.setTitle("Google");
               configurer.setInitialSize(new Point(300, 300));
```

```
                configurer.setShowMenuBar(false);
                configurer.setShowStatusLine(false);
                configurer.setShowCoolBar(false);
        }}
```

Creating the Application class

Before executing the application, you need to create an `Application` class. Similar to the main method within a Java class, this class is the main entry point for the RCP application. This class implements the `org.eclipse.core.runtime.IPlatformRunnable` interface as defined within the plug-in manifest under the `org.eclipse.core.runtime.applications` extension point.

To create an `Application` class, complete the following steps from within the PDE:

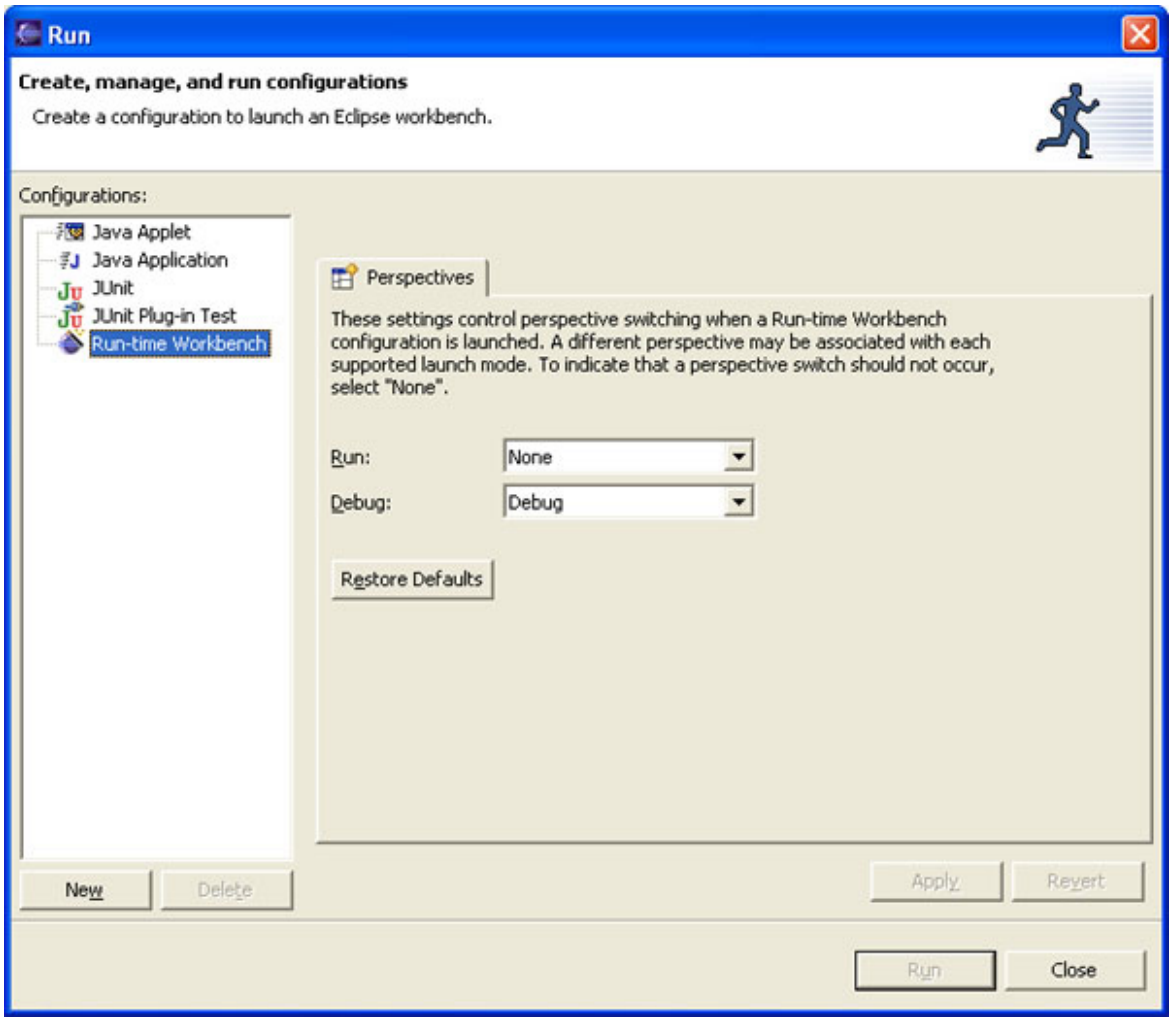
1. Select **File > New > Class** from the menu bar to display the New Java Class wizard.
2. Type **GoogleApplication** into the Name field.
3. Click on the **Add** button to display the Implemented Interfaces Selection dialog box.
4. Type **org.eclipse.core.runtime.IPlatformRunnable** into the Choose Interfaces field and click **OK**.
5. Click the **Finish** button to create the new class.
6. Add the following run method to the generated class. For most RCP applications, this run method will not need to be customized and can be re-used.

```
...
      public Object run(Object args) throws Exception {
               WorkbenchAdvisor workbenchAdvisor = new GoogleWorkbenchAdvisor();
               Display display = PlatformUI.createDisplay();
               int returnCode = PlatformUI.createAndRunWorkbench(display,
                             workbenchAdvisor);
               if (returnCode == PlatformUI.RETURN_RESTART)
                       return IPlatformRunnable.EXIT_RESTART;
               else
                       return IPlatformRunnable.EXIT_OK;
      }
...
```

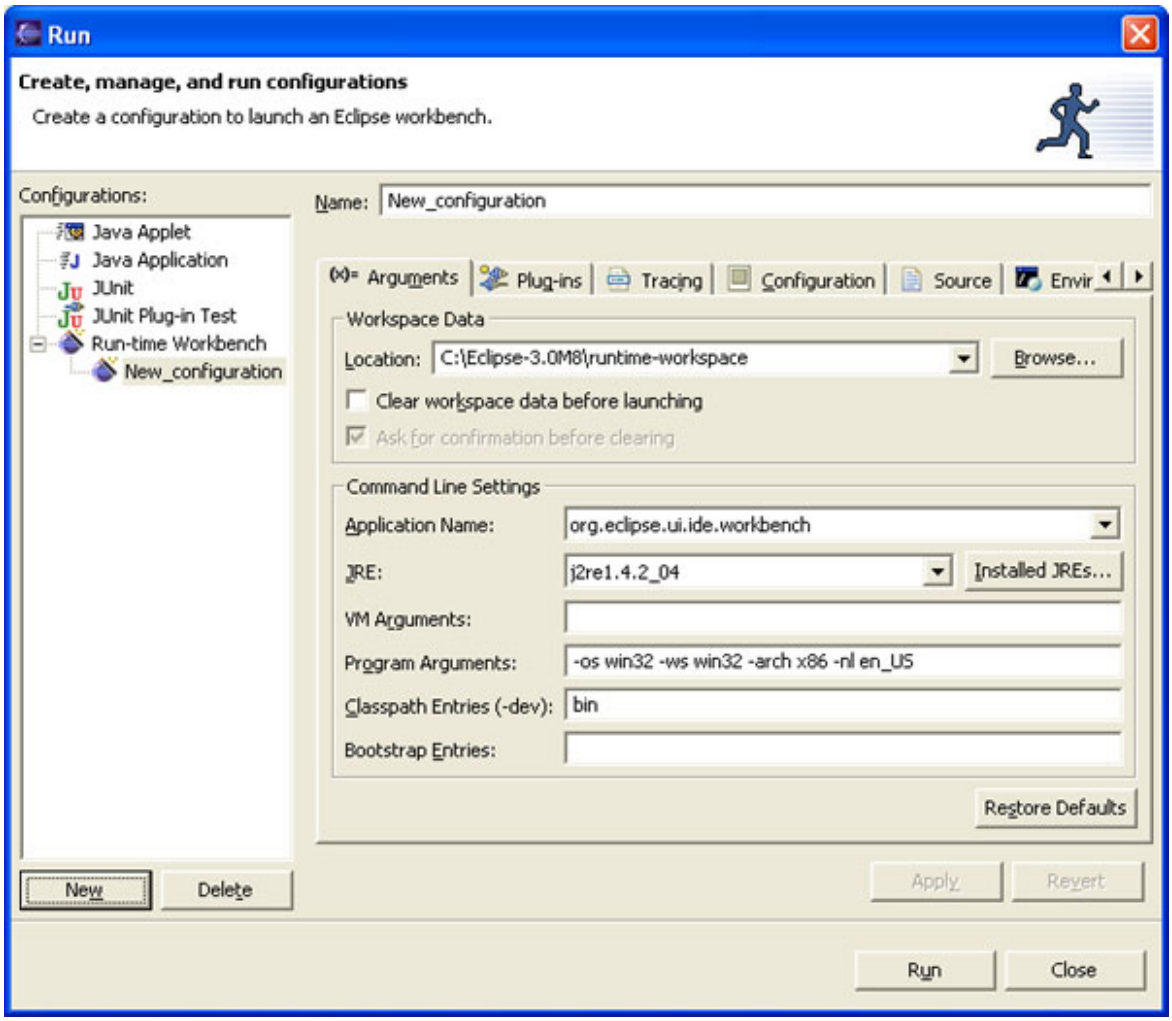
Launching the application with the PDE

To launch the application within the PDE, complete the following steps:

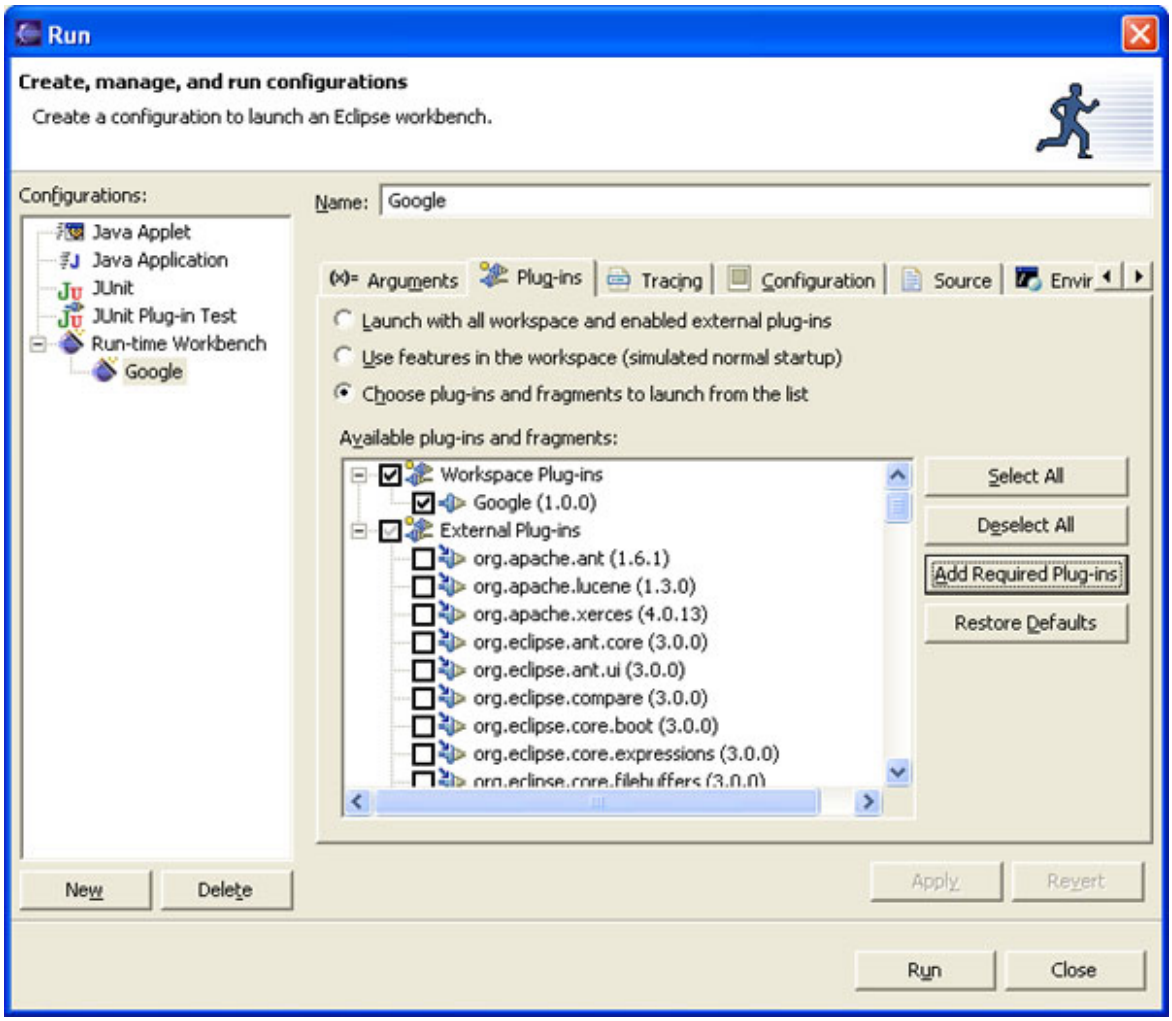
1. Select **Run > Run...** from the menu bar to display the Run dialog as Figure 9 shows.
   Figure 9. The Run dialog



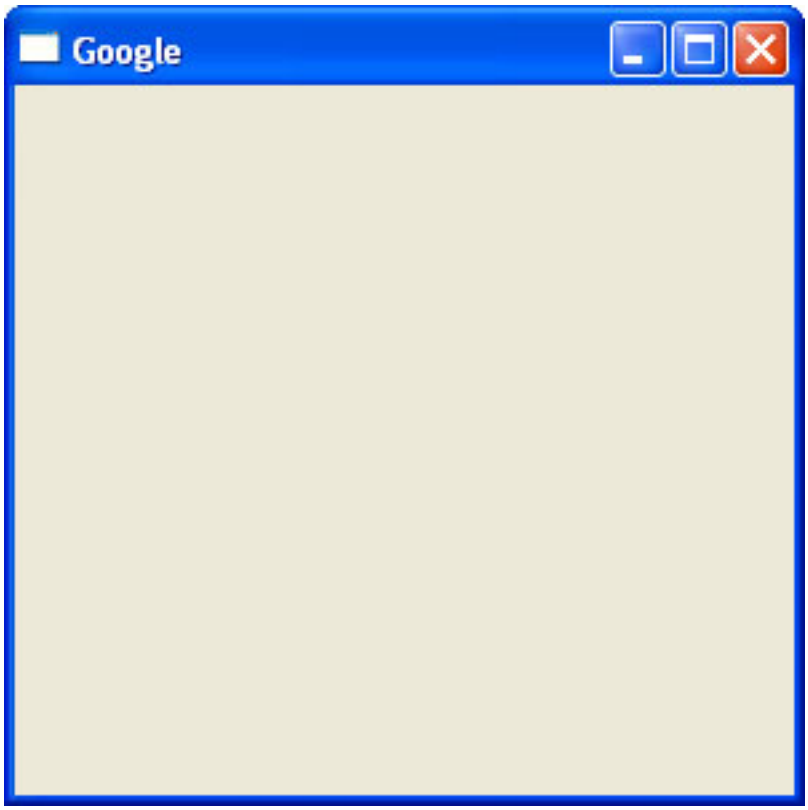
2. Highlight **Run-time Workbench** within the Configurations field and click the **New** button to display a new run-time workbench configuration as Figure 10 shows:
Figure 10. A new Run-time Workbench configuration



3. Type **Google** into the Name field.
4. Select **Google.googleApplication** from the Application Name field.
5. Click on the Plug-ins tab as Figure 11 shows:
   Figure 11. The Plug-ins tab of the Run dialog



6. Select the radio button **Choose plug-ins and fragments to launch from the list**.
7. Click the **Deselect All** button.
8. Check the **Workspace Plug-ins** option. This also selects the Google project.
9. Click the **Add Required Plug-ins** button. This action determines which plug-ins are necessary to execute the application. You will use this list when you assemble the stand-alone application.
10. Click the **Apply** button.
11. Click the **Run** button to execute the application. If everything is configured properly, a window entitled "Google" should display as Figure 12 shows. Although this window doesn't perform any function, it does demonstrate how you can use the PDE to create a generic workbench.
    Figure 12. The new Google window

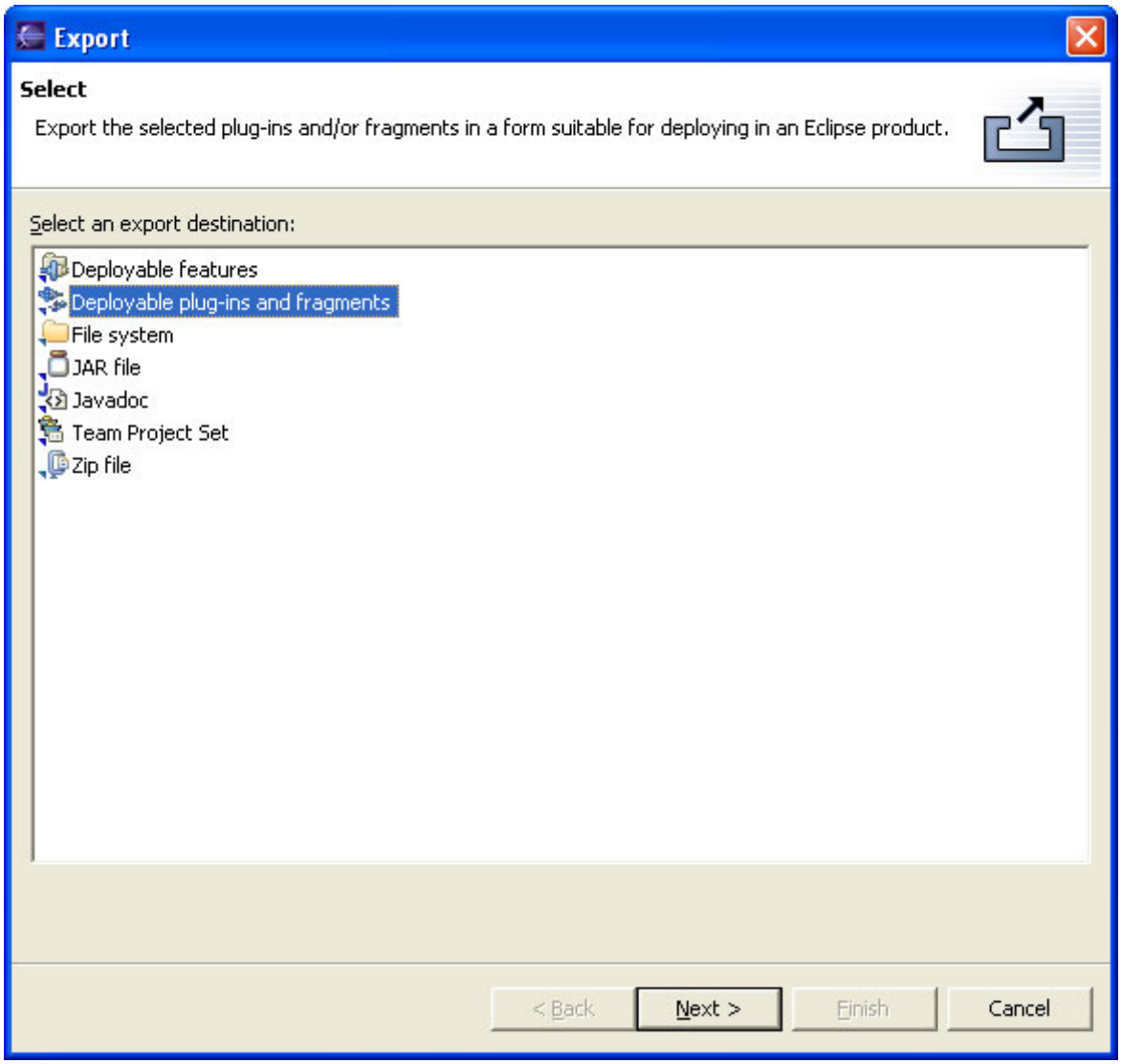

Google

Section 6. Creating a stand-alone application

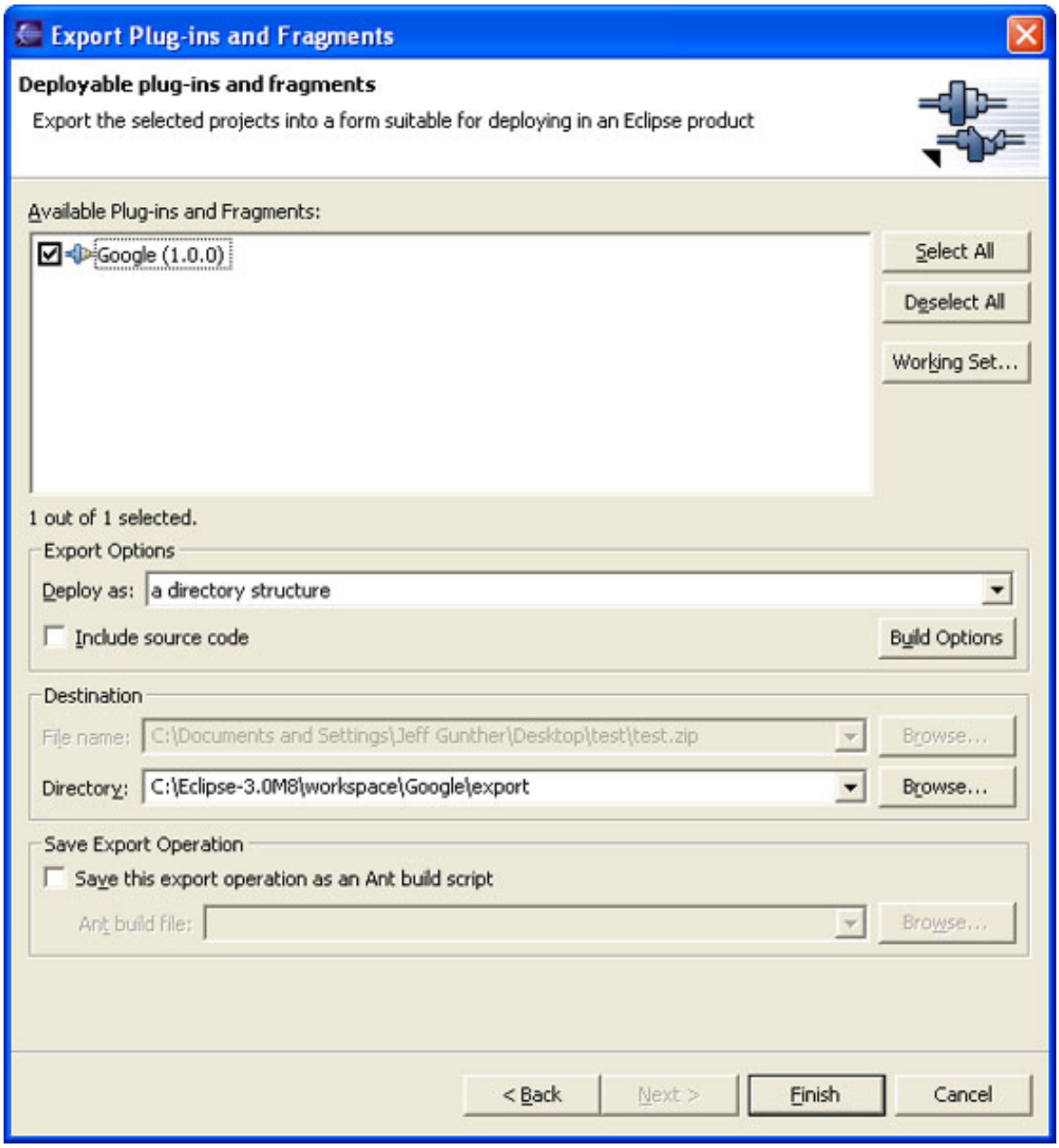
Exporting the application

So far you have focused on how to run an RCP application within the Eclipse IDE. In this section, you'll focus on how to create a stand-alone application by completing the following steps within the PDE:

1. Select **File > Export...** from the menu bar to display the Export dialog as Figure 13 shows:
   Figure 13. The Export dialog



2. Select **Deployable plug-ins and fragments** from the list of export options.
3. Click **Next** to display the Export Plug-ins and Fragments page of the Export wizard as Figure 14 shows:
   Figure 14. Export Plug-ins and Fragments page of the Export wizard



4. Check the Google plug-in.
5. Select **a directory structure** under the Deploy as field.
6. Click the **Browse** button and choose an export location.
7. Click **Finish** to build the project.

Preparing the directory structure

To complete the stand-alone application, you need to copy some files from the Eclipse IDE directory into Google's export directory. Unfortunately, Eclipse 3.0 doesn't provide a tool to copy all the necessary dependent plug-ins and JAR files into the export directory, so you need to complete the following steps:

1. Copy startup.jar from the root directory of the Eclipse 3.0 IDE to the root of the Google application's export directory.
2. Copy the following directories from the Eclipse 3.0 IDE plug-in directory to the plugin directory of the Google application's export directory:
   - org.eclipse.core.expressions_3.0.0
   - org.eclipse.core.runtime_3.0.0
   - org.eclipse.help_3.0.0
   - org.eclipse.jface_3.0.0
   - org.eclipse.osgi_3.0.0
   - org.eclipse.swt.win32_3.0.0 (Windows only)
   - org.eclipse.swt.gtk_3.0.0 (Linux only)
   - org.eclipse.swt_3.0.0
   - org.eclipse.ui.workbench_3.0.0
   - org.eclipse.ui_3.0.0
   - org.eclipse.update.configurator_3.0.0

Testing the application

To test the application, you need to create a launch script. Using your favorite text editor, create a file entitled google.bat (Windows) or google.sh (Linux) with the following content:

```
java -cp startup.jar org.eclipse.core.launcher.Main -application
com.ibm.developerworks.google.googleApplication
```

After you've completed this task, your export directory should have the following structure:

```
- google.bat (Windows only)
- google.sh (Linux only)
- startup.jar
+ ----- plugins
        + ----- org.eclipse.core.expressions_3.0.0
        + ----- org.eclipse.core.runtime_3.0.0
        + ----- org.eclipse.help_3.0.0
        + ----- org.eclipse.jface_3.0.0
        + ----- org.eclipse.osgi.services_3.0.0
        + ----- org.eclipse.osgi.util_3.0.0
        + ----- org.eclipse.osgi_3.0.0
        + ----- org.eclipse.swt.win32_3.0.0 (Windows only)
        + ----- org.eclipse.swt.gtk_3.0.0 (Linux only)
        + ----- org.eclipse.swt_3.0.0
        + ----- org.eclipse.ui.workbench_3.0.0
        + ----- org.eclipse.ui_3.0.0
        + ----- org.eclipse.update.configurator_3.0.0
```

Section 7. Summary and resources

Summary

The RCP will extend and evolve as developers begin to understand and utilize it within their applications. Although we have only barely developed the example application, the companion source code and plug-in manifest demonstrate how to construct a basic RCP application. While this first tutorial provided an overview of the RCP, the next part of this series explores the inner-workings of the generic workbench and the development of the Google RCP application.

Resources

- *Download* (part1-src.zip) the companion source code package for the sample RCP application demonstrated in this tutorial.
- Download *Eclipse 3.0* (http://www.eclipse.org/downloads/index.php) from the Eclipse Foundation.
- Download *Java 2 SDK, Standard Edition 1.4.2* (http://java.sun.com/j2se/1.4.2/download.html) from Sun Microsystems.
- Download *Ant 1.6.1* (http://ant.apache.org/) or higher from the Apache Software Foundation.
- Get an introduction to the core components of the Eclipse Platform from the whitepaper, "*Eclipse Technical Overview*" (Eclipse Web site).
- Find more resources for developing plug-ins on the Eclipse Web site, including, *Platform Extension Points* (Eclipse Web site).
- Find more resources for how to use the Standard Widget Toolkit and JFace on *developerWorks*, including:
  - *Integrate ActiveX controls into SWT applications* (*developerWorks*, June 2003)
  - *Developing JFace wizards* (*developerWorks*, May 2003)
  - *Integrate ActiveX controls into SWT applications* (*developerWorks*, June 2003)
- Find more resources for how to use Eclipse on developerWorks, including:
  - *Developing Eclipse plug-ins* (*developerWorks*, December 2002)
  - *XML development with the Eclipse Platform* (*developerWorks*, April 2003)

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

For more information about the Toot-O-Matic, visit www-106.ibm.com/developerworks/xml/library/x-toot/ .

Eclipse's Rich Client Platform, Part 2: Extending the generic workbench

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Before you start

About this tutorial

The second of a two-part series, this tutorial explores Eclipse's Rich Client Platform (RCP). The first part of the series, *Eclipse's Rich Client Platform, Part 1: Getting started*, began with a review of the Eclipse project and the relevance of the RCP within the marketplace. It discussed the Eclipse plug-in architecture and outlined the necessary steps to implement an RCP application. After providing the necessary background information, you began creating a project within the Eclipse 3.0 IDE. You defined a plug-in manifest, were introduced to extensions and extension points, and created a basic perspective. Using these components, you created some additional supporting Java classes and launched a stand-alone RCP application.

Part 2 of this series leverages the discussion from Part 1 and explores how you can use other Eclipse user-interface components such as views, actions, and wizards to assemble a complete application. In this tutorial, you'll create a front end for the Google API that will give you the ability to query and display search results from Google's extensive catalog of Web sites. Having an application that demonstrates some of these technologies in action will provide you with an understanding of the RCP platform.

You should understand how to navigate Eclipse 3.0 and have a working knowledge of Java to follow this tutorial. You do not need a background in Eclipse plug-in development or an understanding of technologies such as the Standard Widget Toolkit (SWT) and JFace. *Part 1* provided a brief introduction to each of these complementary technologies. This tutorial explores the code and supporting files so you can grasp how to construct an RCP application.

Tools

While not a prerequisite, you'll find this tutorial easier to follow if you download, install, and configure Eclipse 3.0, a 1.4 Java Virtual Machine, and Apache Ant. If you don't have these tools installed, please reference, download, and install the following resources:

- Eclipse 3.0 is available at: *http://www.eclipse.org/downloads/index.php*
- Java 2 Standard Edition, Software Development Kit (SDK) is available at: *http://java.sun.com/j2se/1.4.2/download.html*
- Apache Ant 1.6.1 is available at: *http://ant.apache.org/*

About the author

Jeff Gunther, a *Studio B* (http://www.studiob.com/) author, is the General Manager and founder of Intalgent Technologies, an emerging provider of software products and solutions utilizing the Java 2 Enterprise Edition and Lotus Notes/Domino platforms. Jeff is an application and infrastructure architect with experience in architecting, designing, developing, deploying, and maintaining complex software systems. His diverse experience includes full lifecycle development of software

running on multiple platforms, from Web servers to embedded devices. You can contact him at: *jeff.gunther@intalgent.com*.

Section 2. Defining a view

Overview of views

Views within the Eclipse workbench are visual containers that allow users to display or navigate resources of a particular type. As you begin creating views within your own RCP application, remember to review the view's purpose before starting development. Since a view's responsibility is to display data from your domain model, group similar types of objects into the view. For example, most users of the Eclipse IDE make extensive use of the tasks view within the Java perspective. This view displays types of auto-generated errors, warnings, or information associated with a resource that the developer needs to review and resolve. This approach minimizes the need for the user to toggle between views to accomplish a particular task. The number of views that any application has is largely dependent on the application's size and complexity. The example Google application developed in this tutorial has two views -- one for searching and one for displaying the Web page from the search results, both of which you'll create in this section.

Defining an **`org.eclipse.ui.views`** extension

Similar to other components within Eclipse, to create a new view, you must define a new extension within the project's plug-in manifest. You define views using the `org.eclipse.ui.perspectives` extension point. Using the plugin.xml tab of the plug-in manifest editor within the Google project, add the following content to begin the process of creating the views.

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

                  <category
                        id="com.ibm.developerworks.google.views"
                        name="Google">
                  </category>

                  <view
                        id="com.ibm.developerworks.google.views.SearchView"
                        name="Search"
                        category="com.ibm.developerworks.google.views"
                        class="com.ibm.developerworks.google.views.SearchView"
                        icon="icons/google.gif">
                  </view>

                  <view
                        id="com.ibm.developerworks.google.views.BrowserView"
                        name="Browser"
                        category="com.ibm.developerworks.google.views"
                        class="com.ibm.developerworks.google.views.BrowserView"
                        icon="icons/google.gif">
                  </view>

      </extension>
...
```

The `SearchView` allows users to search Google and display the search results in a

table. The `BrowserView` contains an SWT browser control and displays a particular URL based on the user's action within the search results table.

Next, you'll look at the view extension's elements and attributes in more detail.

Stepping through the `org.eclipse.ui.views` extension point

Use the `<extension>`, `<category>`, and `<view>` elements to define the view extension point. A category is used within the Show View Dialog to group similar views. Each view can appear under multiple categories.

The `<category>` element has the following attributes:

- `id` -- This attribute defines a unique identifier for the category.
- `name` -- This attribute defines a name for this category, and the workbench uses it to represent this category.
- `parentCategory` -- This optional attribute defines a list of categories separated by '/'. This element creates category hierarchies.

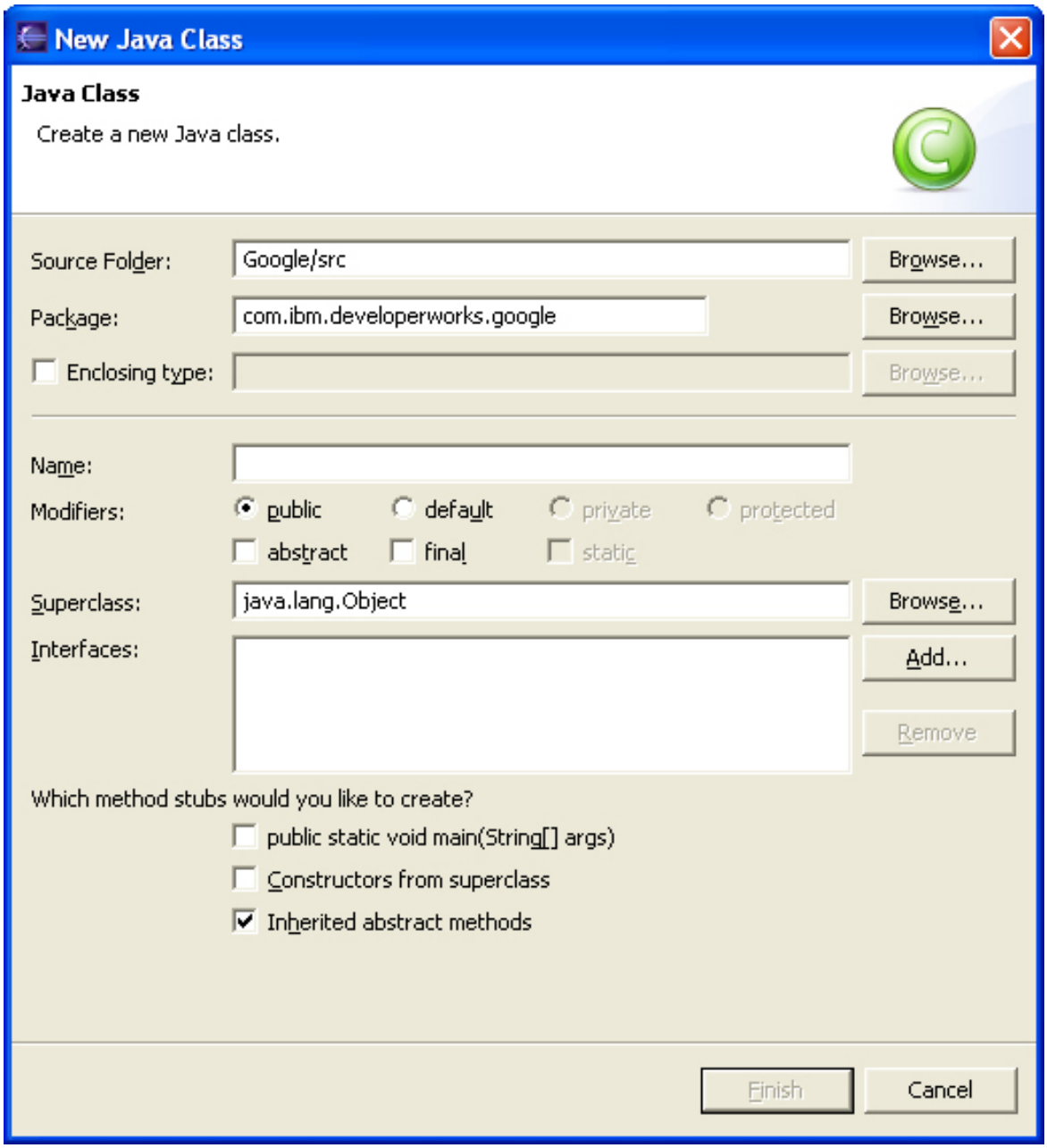
The `<view>` element has the following attributes:

- `id` -- This attribute defines a unique identifier for the view.
- `name` -- This attribute defines a name for this view, and the workbench uses it to represent this view.
- `category` -- This optional attribute defines the categories identifiers. Each category is separated by a '/' and must exist within the plug-in manifest prior to being referenced by the `<view>` element.
- `class` -- This attribute contains the fully-qualified name of the class that implements the `org.eclipse.ui.IViewPart` interface.
- `icon` -- This optional attribute contains a relative name of the icon associated with the view.
- `fastViewWidthRatio` -- This optional attribute contains the percentage of the width of the workbench that the view will take up. This attribute must be a floating point value between 0.05 and 0.95.
- `allowMultiple` -- This optional attribute indicates whether this view allows for the creation of multiple instances within the workbench.

Creating the `SearchView` class

To create the `SearchView` class within the Google project, complete the following steps:

1. Select **File > New > Class** from the menu bar to display the New Java Class wizard.
   Figure 1. New Java Class wizard



2. Type **com.ibm.developerworks.google.views** into the Package field.
3. Type **SearchView** into the Name field.
4. Click on the **Browse** button to display the Superclass Selection dialog box.
5. Type **org.eclipse.ui.part.ViewPart** into the Choose a Type field and click **OK**.
6. Click the **Finish** button to create the new class.

Implementing the **`SearchView`** class

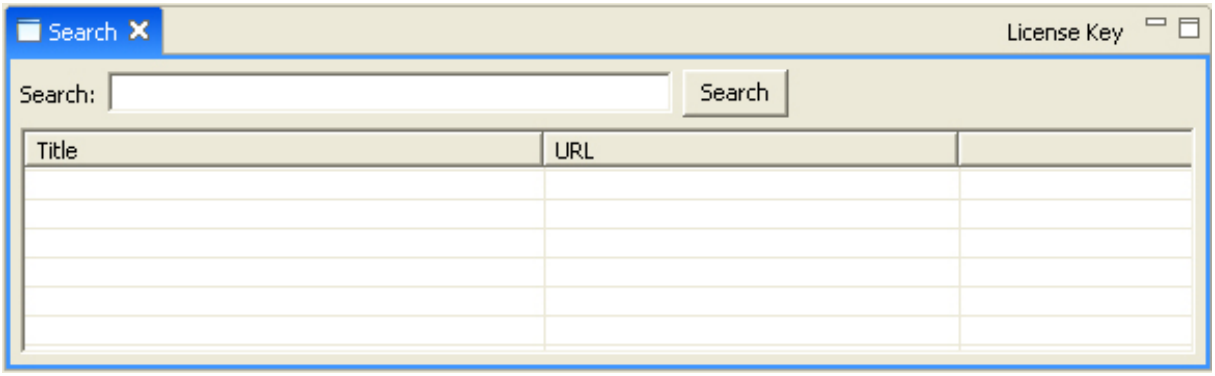
After the class is created, the `createPartControl` and `setFocus` methods must be implemented. The `createPartControl` method is responsible for creating the view's user-interface controls. In this case, an SWT GridLayout layout is used to

arrange the SWT label, SWT text, SWT button, and an SWT table on the view's composite. For more information about SWT's various layouts or how to use SWT user-interface components, please refer to the Resources on page 29 at the end of this tutorial.

The code within the `createPartControl` method renders the user interface Figure 2 shows.

```
...
    public void createPartControl(Composite parent)
    {
        GridLayout gridLayout = new GridLayout();
        gridLayout.numColumns = 3;
        gridLayout.marginHeight = 5;
        gridLayout.marginWidth = 5;

        parent.setLayout(gridLayout);

        Label searchLabel = new Label(parent, SWT.NONE);
        searchLabel.setText("Search:");

        searchText = new Text(parent, SWT.BORDER);
        searchText.setLayoutData(new GridData(GridData.GRAB_HORIZONTAL
                | GridData.HORIZONTAL_ALIGN_FILL));

        Button searchButton = new Button(parent, SWT.PUSH);
        searchButton.setText(" Search ");
...
        GridData gridData = new GridData();
        gridData.verticalAlignment = GridData.FILL;
        gridData.horizontalSpan = 3;
        gridData.grabExcessHorizontalSpace = true;
        gridData.grabExcessVerticalSpace = true;
        gridData.horizontalAlignment = GridData.FILL;

        tableViewer = new TableViewer(parent, SWT.FULL_SELECTION | SWT.BORDER);
        tableViewer.setLabelProvider(new SearchViewLabelProvider());
        tableViewer.setContentProvider(new ViewContentProvider());
        tableViewer.setInput(model);
        tableViewer.getControl().setLayoutData(gridData);
        tableViewer.addDoubleClickListener(this);

        Table table = tableViewer.getTable();
        table.setHeaderVisible(true);
        table.setLinesVisible(true);

        TableColumn titleColumn = new TableColumn(table, SWT.NONE);
        titleColumn.setText("Title");
        titleColumn.setWidth(250);

        TableColumn urlColumn = new TableColumn(table, SWT.NONE);
        urlColumn.setText("URL");
        urlColumn.setWidth(200);
...
```

Figure 2. Search view of the Google application



Implementing the **`SearchView`** class, continued

In addition to the `createPartControl` method, the `setFocus` method must be implemented. In this case the focus defaults to an SWT Text field that allows a user to input search criteria for Google. This method is called upon the view being rendered within the workbench.

```
...
    public void setFocus()
    {
        searchText.setFocus();
    }
...
```

Once a user double clicks on a row within the search results table, the Web site loads within another view that contains an SWT browser control. This is accomplished by having the `SearchView` implement the `IDoubleClickListener` interface. The `IDoubleClickListener` interface requires a `doubleClick` method to be added to the `SearchView`.

```
...
    public void doubleClick(DoubleClickEvent event)
    {
        if (!tableViewer.getSelection().isEmpty())
        {

            IStructuredSelection ss = (IStructuredSelection) tableViewer
                    .getSelection();
            GoogleSearchResultElement element = (GoogleSearchResultElement) ss
                    .getFirstElement();

            BrowserView.browser.setUrl(element.getURL());
        }

    }
...
```

Find the complete source code for the `SearchView` class below:

```
package com.ibm.developerworks.google.views;

import org.eclipse.jface.dialogs.MessageDialog;
```

```
import org.eclipse.jface.viewers.DoubleClickEvent;
import org.eclipse.jface.viewers.IDoubleClickListener;
import org.eclipse.jface.viewers.IStructuredSelection;
import org.eclipse.jface.viewers.TableViewer;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.events.SelectionListener;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Table;
import org.eclipse.swt.widgets.TableColumn;
import org.eclipse.swt.widgets.Text;
import org.eclipse.ui.internal.dialogs.ViewContentProvider;
import org.eclipse.ui.part.ViewPart;

import com.google.soap.search.GoogleSearch;
import com.google.soap.search.GoogleSearchFault;
import com.google.soap.search.GoogleSearchResult;
import com.google.       soap.search.GoogleSearchResultElement;
import com.ibm.developerworks.google.GoogleApplication;

public class SearchView extends ViewPart implements IDoubleClickListener
{
    public static final String ID = "com.ibm.developerworks.google.views.SearchView";

    private TableViewer tableViewer;

    private Text searchText;

    private GoogleSearchResultElement model;

    public void createPartControl(Composite parent)
    {
        GridLayout gridLayout = new GridLayout();
        gridLayout.numColumns = 3;
        gridLayout.marginHeight = 5;
        gridLayout.marginWidth = 5;

        parent.setLayout(gridLayout);

        Label searchLabel = new Label(parent, SWT.NONE);
        searchLabel.setText("Search:");

        searchText = new Text(parent, SWT.BORDER);
        searchText.setLayoutData(new GridData(GridData.GRAB_HORIZONTAL
                | GridData.HORIZONTAL_ALIGN_FILL));

        Button searchButton = new Button(parent, SWT.PUSH);
        searchButton.setText(" Search ");
        searchButton.addSelectionListener(new SelectionListener()
        {

            public void widgetSelected(SelectionEvent e)
            {

                GoogleSearch search = new GoogleSearch();
                search.setKey(GoogleApplication.LICENSE_KEY);
                search.setQueryString(searchText.getText());
                try
                {
```

```
                    GoogleSearchResult result = search.doSearch();

                    tableViewer.setInput(model);
                    tableViewer.add(result.getResultElements());

                } catch (GoogleSearchFault ex)
                {
                    MessageDialog.openWarning(e.display.getActiveShell(),
                            "Google Error", ex.getMessage());
                }

            }

            public void widgetDefaultSelected(SelectionEvent e)

            }
        });

        GridData gridData = new GridData();
        gridData.verticalAlignment = GridData.FILL;
        gridData.horizontalSpan = 3;
        gridData.grabExcessHorizontalSpace = true;
        gridData.grabExcessVerticalSpace = true;
        gridData.horizontalAlignment = GridData.FILL;

        tableViewer = new TableViewer(parent, SWT.FULL_SELECTION | SWT.BORDER);
        tableViewer.setLabelProvider(new SearchViewLabelProvider());
        tableViewer.setContentProvider(new ViewContentProvider());
        tableViewer.setInput(model);
        tableViewer.getControl().setLayoutData(gridData);
        tableViewer.addDoubleClickListener(this);

        Table table = tableViewer.getTable();
        table.setHeaderVisible(true);
        table.setLinesVisible(true);

        TableColumn titleColumn = new TableColumn(table, SWT.NONE);
        titleColumn.setText("Title");
        titleColumn.setWidth(250);

        TableColumn urlColumn = new TableColumn(table, SWT.NONE);
        urlColumn.setText("URL");
        urlColumn.setWidth(200);

    }

    public void setFocus()
    {
        searchText.setFocus();
    }

    public void doubleClick(DoubleClickEvent event)
    {
        if (!tableViewer.getSelection().isEmpty())
        {

            IStructuredSelection ss = (IStructuredSelection) tableViewer
                    .getSelection();
            GoogleSearchResultElement element = (GoogleSearchResultElement) ss
                    .getFirstElement();

            BrowserView.browser.setUrl(element.getURL());
        }
```

```
    }
}
```

Creating the **SearchViewLabelProvider** class

In the source code on the previous panel, the `TableViewer` object uses a class called `SearchViewLabelProvider`. In this instance, a label provider sets the column's text for each row of the table. To create the `SearchViewLabelProvider` class for the `SearchView` class within the Google project, complete the following steps:

1. Select **File > New > Class** from the menu bar to display the New Java Class wizard.
2. Type **com.ibm.developerworks.google.views** into the Package field.
3. Type **SearchViewLabelProvider** into the Name field.
4. Click on the **Browse** button to display the Superclass Selection dialog box.
5. Type **org.eclipse.jface.viewers.LabelProvider** into the Choose a Type field and click **OK**.
6. Click on the **Add** button to display the Implemented Interfaces Selection dialog box.
7. Type **org.eclipse.jface.viewers.ITableLabelProvider** into the Choose an interface field and click **OK**.
8. Click the **Finish** button to create the new class.

Implementing the **SearchViewLabelProvider** class

The `ITableLabelProvider` interface requires that the `getColumnImage` and `getColumnText` methods be implemented within the class. Since the results table does not include any images, the `getColumnImage` method simply returns null. The `getColumnText` uses the `GoogleSearchResultElement` class provided by the Google API to set the first and second columns of the SWT table. The first column contains the title of the search result, and the second column contains the search result's URL.

```
package com.ibm.developerworks.google.views;

...

public class SearchViewLabelProvider extends LabelProvider implements
        ITableLabelProvider
{

    public Image getColumnImage(Object element, int columnIndex)
    {
        return null;
    }

    public String getColumnText(Object element, int columnIndex)
    {
        switch (columnIndex)
```

```
            {
            case 0:
                return ((GoogleSearchResultElement) element).getTitle();
            case 1:
                return ((GoogleSearchResultElement) element).getURL();

            }
            return "";

        }

    }
```

Creating the **`BrowserView`** class

Now you need to create a view to display the URL that the user selects within the search result table. To create the `BrowserView` class within the Google project, complete the following steps:

1. Select **File > New > Class** from the menu bar to display the New Java Class wizard.
2. Type **com.ibm.developerworks.google.views** into the Package field.
3. Type **BrowserView** into the Name field.
4. Click on the **Browse** button to display the Superclass Selection dialog box.
5. Type **org.eclipse.ui.part.ViewPart** into the Choose a Type field and click **OK**.
6. Click the **Finish** button to create the new class.

Implementing the **`BrowserView`** class

As for the `SearchView` class, you must implement the `createPartControl` and `setFocus` methods in the `BrowserView` class. In this case, an SWT browser control is embedded within the view. This control displays the Web page that the user selects within the search results table.

```
package com.ibm.developerworks.google.views;

import org.eclipse.swt.SWT;
import org.eclipse.swt.browser.Browser;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.ui.part.ViewPart;

public class BrowserView extends ViewPart
{
    public static final String ID = "com.ibm.developerworks.google.views.BrowserView";

    public static Browser browser;

    public void createPartControl(Composite parent)
    {
        GridLayout gridLayout = new GridLayout();
        gridLayout.numColumns = 1;
        gridLayout.marginHeight = 5;
```

```
            gridLayout.marginWidth = 5;
            parent.setLayout(gridLayout);

            browser = new Browser(parent, SWT.NONE);

            browser.setLayoutData(new GridData(GridData.GRAB_HORIZONTAL
                    | GridData.GRAB_VERTICAL | GridData.FILL_HORIZONTAL
                    | GridData.FILL_VERTICAL));
            browser.setUrl("about:");

        }

        public void setFocus()
        {
            browser.setFocus();

        }
    }
```

Integrating the **`SearchView`** and **`BrowserView`** into a perspective

With the two views and supporting classes defined for your Google application, you need to integrate these components into the existing perspective you created in Part 1. Open the `GooglePerspective` class and modify the `createInitialLayout` method.

Find the complete source code for the `GooglePerspective` class below:

```
package com.ibm.developerworks.google;

import org.eclipse.ui.IPageLayout;
import org.eclipse.ui.IPerspectiveFactory;

import com.ibm.developerworks.google.views.BrowserView;
import com.ibm.developerworks.google.views.SearchView;

public class GooglePerspective implements IPerspectiveFactory
{
    public static final String ID = "com.ibm.developerworks.google.GooglePerspective";

    public void createInitialLayout(IPageLayout layout)
    {
        layout.setEditorAreaVisible(false);
        layout.addView(SearchView.ID, IPageLayout.BOTTOM, new Float(0.60)
                .floatValue(), IPageLayout.ID_EDITOR_AREA);
        layout.addView(BrowserView.ID, IPageLayout.TOP, new Float(0.40)
                .floatValue(), IPageLayout.ID_EDITOR_AREA);
    }
}
```

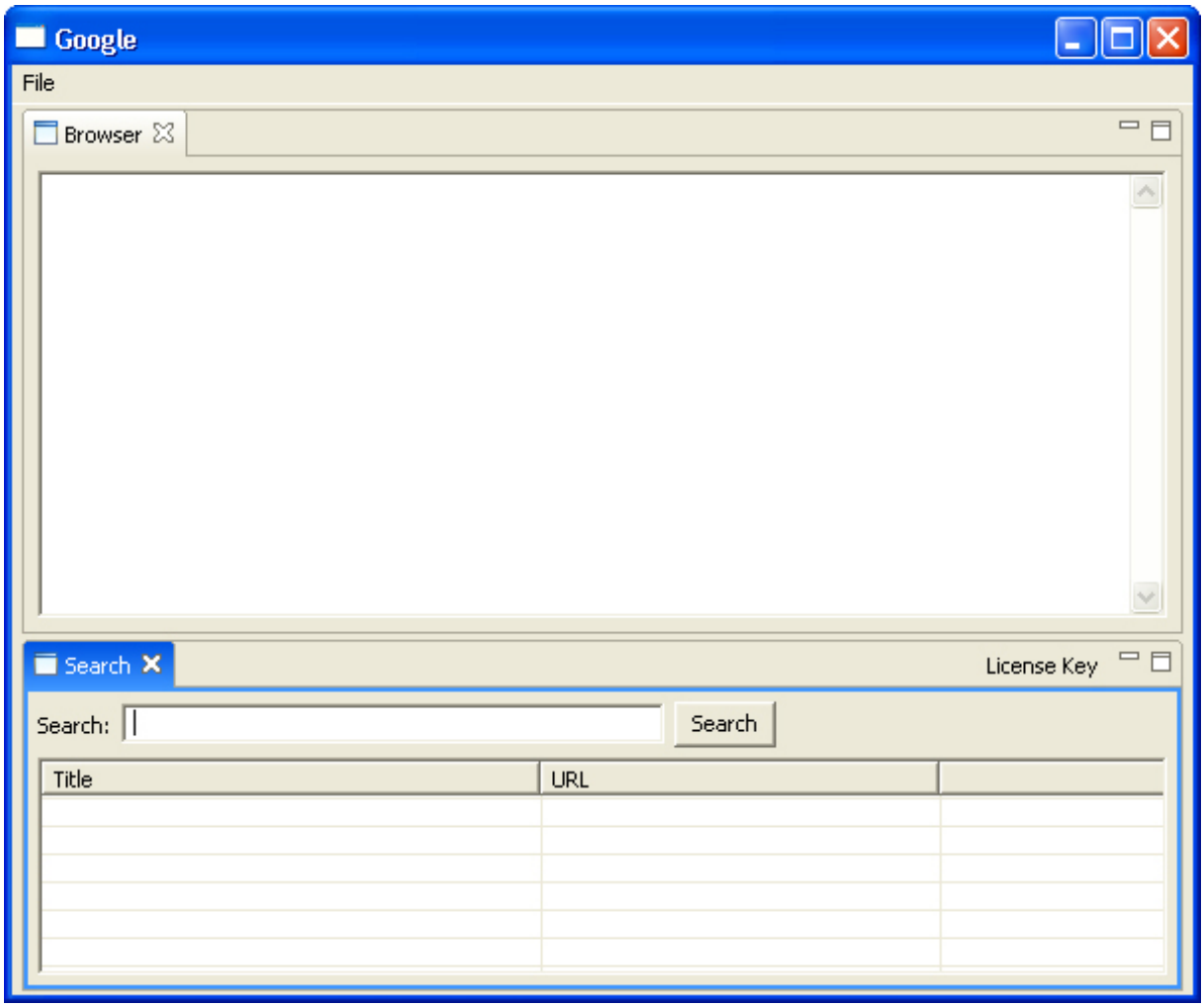
As Figure 3 shows, the last two lines within the `createInitialLayout` open the `SearchView` and `BrowserView` when the perspective is rendered. The `addView` method contains four parameters:

1. The first parameter contains the unique identifier for the view.
2. The second parameter defines the relationship to the workbench. Possible options

include Top, Bottom, Left, and Right.

3. The third parameter defines a ratio of how to divide the available space within the workbench. This parameter is a float value between 0.05 and 0.95
4. The final parameter contains the unique identifier reference for where the view should be displayed. In this case, the editor area is used.

Figure 3. The Search and Browser views of the Google application



The next section focuses on how to add menu bars, dialogs, action, and wizards to an RCP application.

Section 3. Integrating menu bars and dialog boxes

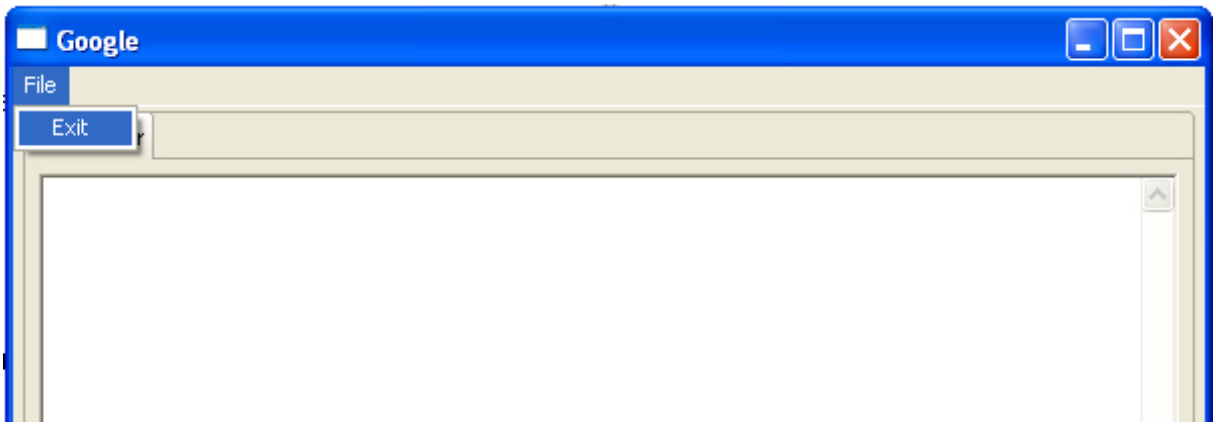
Adding menu bars to a perspective

Sometimes you'll want to add actions to an RCP application by creating a menu bar within the main window. To add new items to the menu bar, you need to override the `fillActionBars` method within the `WorkbenchAdvisor`.

```
...
    public void fillActionBars(IWorkbenchWindow window,
            IActionBarConfigurer configurer, int flags)
    {
        IMenuManager menuBar = configurer.getMenuManager();

        MenuManager fileMenu = new MenuManager("File",
                IWorkbenchActionConstants.M_FILE);
        fileMenu.add(new GroupMarker(IWorkbenchActionConstants.FILE_START));
        fileMenu.add(new GroupMarker(IWorkbenchActionConstants.MB_ADDITIONS));
        fileMenu.add(ActionFactory.QUIT.create(window));
        fileMenu.add(new GroupMarker(IWorkbenchActionConstants.FILE_END));

        menuBar.add(fileMenu);
    }
...
```

In the source code above, the `MenuManager` class adds a `fileMenu` to the workbench. Figure 4 shows the menu bar in action within the Google application.

Figure 4. Menu bars in the `WorkbenchAdvisor` class



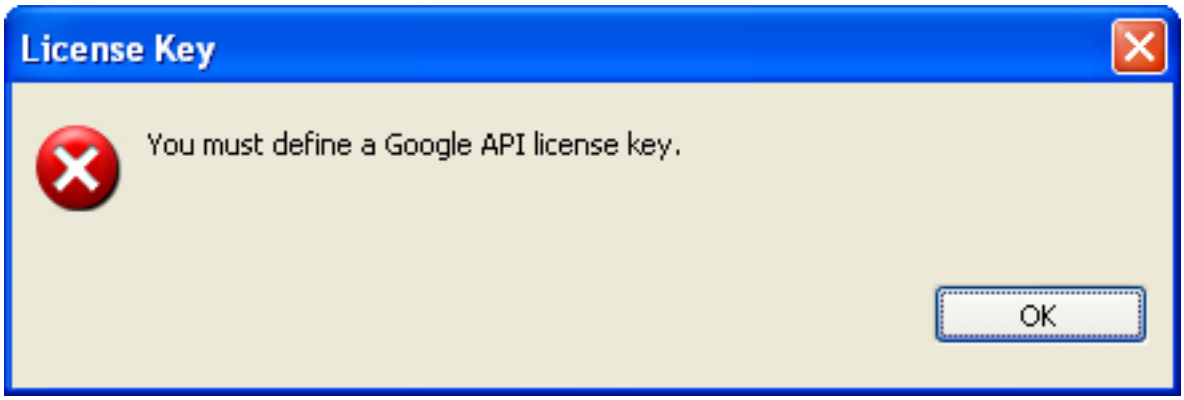
In addition to menu bars, the JFace toolkit provides some predefined dialog boxes that can enhance a user's experience with an RCP application. The next panel reviews the various dialog types the JFace package provides.

Various types of dialog boxes

The JFace toolkit includes a variety of dialogs that can display messages to your application's users. As Figure 5 demonstrates, the `SearchView` class uses the `MessageDialog` class to display an error to users if they don't provide a Google API

license key before executing a query.

Figure 5. Error message if the license key is not provided



In addition to the `MessageDialog`, the JFace package provides three other dialog types:

1. `ErrorDialog` uses an IStatus object and displays information about a particular error.
2. `InputDialog` allows the user to enter text into the dialog box.
3. `ProgressMonitorDialog` shows the execution progress of a process to the user.

The next section describes how you can add a wizard to your RCP application to gather data.

Section 4. Defining a wizard

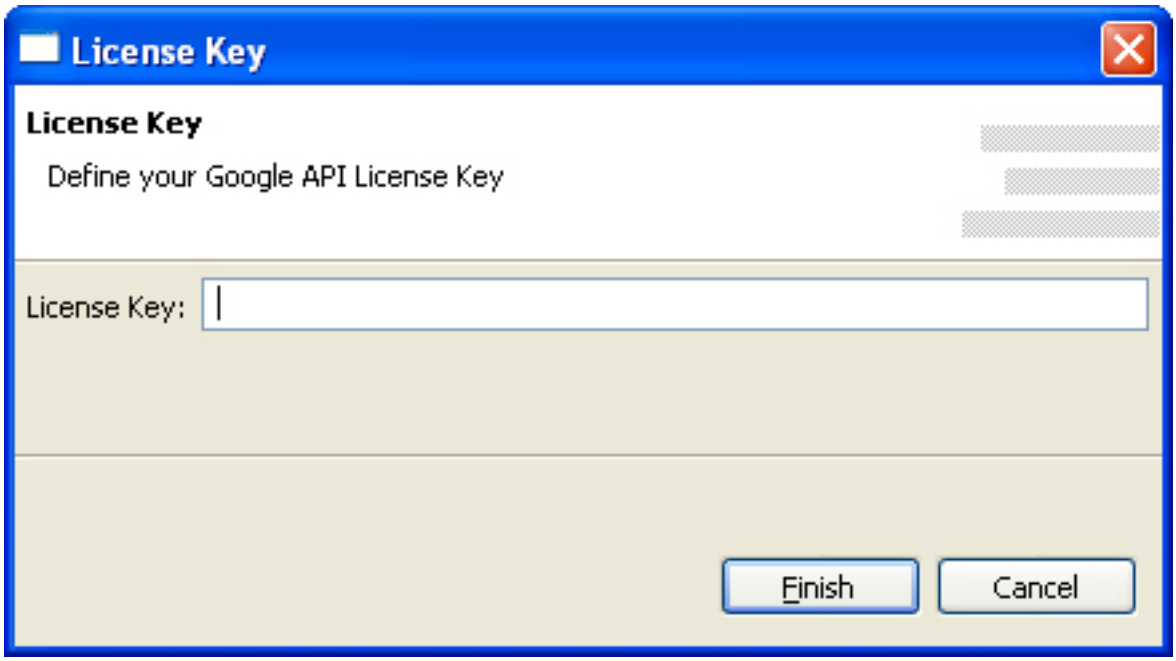
Overview of wizards

The JFace toolkit includes a powerful set of user-interface components that you can easily integrate into an RCP application. An interesting component of this toolkit is the support for wizards. A JFace wizard, coupled with other user-interface components within the Standard Widget Toolkit (SWT), provides a flexible mechanism to systematically gather user input and perform input validation.

Before reviewing the code and implementation details of how to create a wizard, review the purpose for a wizard within your Google application. To query the Google API, you must sign up for an account. Once your account has been activated, you'll be provided a license key. Google currently allows each account the ability to execute 1000 queries per day. Since you need to supply a license key as a parameter within the `GoogleSearch` object, you need a mechanism to gather the license key from the user.

As Figure 6 demonstrates, the application contains a JFace wizard consisting of one page that requests the license key.

Figure 6. The License Key wizard within the Google application



For more information on how to create an account to access the Google API, please refer to Resources on page29 .

Creating a `LicenseKeyWizard` class

To create a basic wizard, create a class that extends `org.eclipse.jface.wizard.Wizard`. In the Google application, a wizard will be

used to gather the user's Google API license key. To create the `LicenseKeyWizard` class within the Google project, complete the following steps:

1. Select **File > New > Class** from the menu bar to display the New Java Class wizard.
2. Type **com.ibm.developerworks.google.wizards** into the Package field.
3. Type **LicenseKeyWizard** into the Name field.
4. Click on the **Browse** button to display the Superclass Selection dialog box.
5. Type **org.eclipse.jface.wizard.Wizard** into the Choose a Type field and click **OK**.
6. Click the **Finish** button to create the new class.

Implementing the **`LicenseKeyWizard`** class

After creating the `LicenseKeyWizard` class, you need to override the `addPages` and `performFinish` methods. The `addPages` method adds pages to a wizard before it displays to the end user. The `performFinish` method executes when the user presses the **Finish** button within the wizard. The `LicenseKeyWizard` gathers the license key data and populates it to a static String variable in the class.

Find the complete source code for the `LicenseKeyWizard` class below:

```
package com.ibm.developerworks.google.wizards;

import org.eclipse.jface.wizard.Wizard;


public class LicenseKeyWizard extends Wizard
{
    private static String licenseKey;
    private LicenseKeyWizardPage page;

    public LicenseKeyWizard()
    {
        super();
        this.setWindowTitle("License Key");
    }

    public void addPages()
    {
        page = new LicenseKeyWizardPage("licenseKey");
        addPage(page);
    }
    public boolean performFinish()
    {
        if(page.getLicenseKeyText().getText().equalsIgnoreCase(""))
        {
            page.setErrorMessage("You must provide a license key.");
            page.setPageComplete(false);
            return false;
        }
        else
        {
            licenseKey = page.getLicenseKeyText().getText();
            return true;
```

```
            }

        }

        public static String getLicenseKey()
        {
            return licenseKey;
        }

        public static void setLicenseKey(String licenseKey)
        {
            LicenseKeyWizard.licenseKey = licenseKey;
        }
    }
```

Creating a **`LicenseKeyWizardPage`** class

In addition to the wizard class, each wizard must have at least one page that extends `org.eclipse.jface.wizard.WizardPage`. To create the `LicenseKeyWizardPage` class within the Google project, complete the following steps:

1. Select **File > New > Class** from the menu bar to display the New Java Class wizard.
2. Type **com.ibm.developerworks.google.wizards** into the Package field.
3. Type **LicenseKeyWizardPage** into the Name field.
4. Click on the **Browse** button to display the Superclass Selection dialog box.
5. Type **org.eclipse.jface.wizard.WizardPage** into the Choose a Type field and click **OK**.
6. Click the **Finish** button to create the new class.

Implementing the **`LicenseKeyWizardPage`** class

Without a class that implements a `WizardPage`, the `LicenseKeyWizard` wouldn't have any behavior. You can think of a wizard as a stack of cards, each with its own layout and design. Each `WizardPage` is responsible for the layout and behavior of a single page or card within a wizard. To create a `WizardPage`, you need to subclass the `WizardPage` base implementation and implement the `createControl` method to create the specific user-interface controls.

Find the complete source code for the `LicenseKeyWizardPage` class below:

```
package com.ibm.developerworks.google.wizards;

import org.eclipse.jface.wizard.WizardPage;
import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Text;
```

```
public class LicenseKeyWizardPage extends WizardPage
{
    private Text licenseKeyText;

    protected LicenseKeyWizardPage(String pageName)
    {
        super(pageName);
        setTitle("License Key");
        setDescription("Define your Google API License Key");
    }

    public void createControl(Composite parent)
    {
        GridLayout pageLayout = new GridLayout();
        pageLayout.numColumns = 2;
        parent.setLayout(pageLayout);
        parent.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));

        Label label = new Label(parent, SWT.NONE);
        label.setText("License Key:");

        licenseKeyText = new Text(parent, SWT.BORDER);
        licenseKeyText.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));

        setControl(parent);
    }

    public Text getLicenseKeyText()
    {
        return licenseKeyText;
    }

    public void setLicenseKeyText(Text licenseKeyText)
    {
        this.licenseKeyText = licenseKeyText;
    }
}
```

Section 5. Defining an action

Overview of actions

Actions within the Eclipse workbench are commands that the user of an application triggers. In general, actions fall into three distinct categories: buttons, items within the tool bar, or items within the menu bar. For example, when you select **File > New > Class** from the menu bar, you're executing an action that opens the New Java Class wizard. When you execute an action within the workbench, the action's run method performs its particular function within the application. In addition to an action's class, an action can have other properties that control how the action is rendered within the workbench. These properties include a text label, mouse over tool tip, and an icon. This tutorial's example Google application has two actions -- one that's used to exit the application and one that allows users to provide their Google API license key by clicking a button located on the Search view.

This section explores how to define actions with an extension point within the plug-in manifest. Specifically, it covers how to add an action to the pull-down menu of the Search view.

Defining the **`org.eclipse.ui.viewActions`** extension point

To add a new action to a view, you must define a new extension within the project's plug-in manifest. View actions are defined using the `org.eclipse.ui.viewActions` extension point. Each view has a pull-down menu that activates when you click on the top right triangle button. Using the plugin.xml tab of the plug-in manifest editor within the Google project, add the following content to begin the process of creating a view action:

```
...
      <extension point="org.eclipse.ui.viewActions">

            <viewContribution
                  id="com.ibm.developerworks.google.views.contribution"
                  targetID="com.ibm.developerworks.google.views.SearchView">

                        <action
                            id="com.ibm.developerworks.google.actions.LicenseKeyAction"
                            label="License Key"
                            toolbarPath="additions"
                            style="push"
                            state="false"
                            tooltip="Google License Key"
                 class="com.ibm.developerworks.google.actions.LicenseKeyAction" />

            </viewContribution>

      </extension>
...
```

The `LicenseKey` view action allows users to set the license key that will be used to query Google's API. The next few panels describe the

`org.eclipse.ui.viewActions` extension point and the steps necessary to create an `Action class`.

Stepping through the **`org.eclipse.ui.viewActions`** extension point

Beginning with the `<extension>` element, a simple `org.eclipse.ui.viewActions` extension contains a `<viewContribution>` and `<action>` element.

A `<viewContribution>` defines a group of view actions and menus. This element has the following attributes:

1. `id` -- This attribute defines a unique identifier for the view contribution.
2. `targetID` -- This attribute defines a registered view that is the target of the contribution.

The `<action>` element has the following attributes:

- `id` -- This attribute defines a unique identifier for the action.
- `label` -- This attribute defines a name for this action and, the workbench uses it to represent this action.
- `menubarPath` -- This optional attribute contains a slash-delimited path ('/') used to specify the location of this action in the pull-down menu.
- `toolbarPath` -- This optional attribute contains a named group within the toolbar of the target view. If this attribute is omitted, the action will not appear in the view's toolbar.
- `icon` -- This optional attribute contains the relative path of an icon used to visually represent the action within the view.
- `disableIcon` -- This optional attribute contains the relative path of an icon used to visually represent the action when it's disabled.
- `hoverIcon` -- This optional attribute contains the relative path of an icon used to visually represent the action when the mouse pointer is hovering over the action.
- `tooltip` -- This optional attribute defines the text for the action's tool tip.
- `helpContextId` -- This optional attribute defines a unique identifier indicating the action's help context.
- `style` -- This optional attribute defines the user-interface style type for the action. Options include push, radio, or toggle.
- `state` -- When the style attribute is toggled, this optional attribute defines the initial state of the action.
- `class` -- This attribute contains the fully qualified name of the class that implements the `org.eclipse.ui.IViewActionDelegate` interface.

Creating the **`LicenseKeyAction`** class

To create the `LicenseKeyAction` class for the `SearchView` class, complete the

following steps within the Google project:

1. Select **File > New > Class** from the menu bar to display the New Java Class wizard.
2. Type **com.ibm.developerworks.google.actions** into the Package field.
3. Type **LicenseKeyAction** into the Name field.
4. Click on the **Add** button to display the Implemented Interfaces Selection dialog box.
5. Type **org.eclipse.ui.IViewActionDelegate** into the Choose an interface field and click **OK**.
6. Click the **Finish** button to create the new class.

Implementing the `LicenseKeyAction` class

When an action is invoked, the action's run method executes. In the Google application, the `LicenseKeyAction` class launches a wizard to collect the user's Google API license key. In this case, this action is located in the upper-right corner of the search view.

Find the source code for the `LicenseKeyAction` class below:

```
package com.ibm.developerworks.google.actions;

import org.eclipse.jface.action.IAction;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.jface.wizard.WizardDialog;
import org.eclipse.ui.IViewActionDelegate;
import org.eclipse.ui.IViewPart;

import com.ibm.developerworks.google.views.SearchView;
import com.ibm.developerworks.google.wizards.LicenseKeyWizard;

public class LicenseKeyAction implements IViewActionDelegate
{
    private SearchView searchView;

    public void init(IViewPart view)
    {
        this.searchView = (SearchView) view;
    }

    public void run(IAction action)
    {
        LicenseKeyWizard wizard = new LicenseKeyWizard();
        WizardDialog dialog = new WizardDialog(searchView.getViewSite()
                .getShell(), wizard);
        dialog.open();

    }

    public void selectionChanged(IAction action, ISelection selection)
    {

    }
```

```
}
```

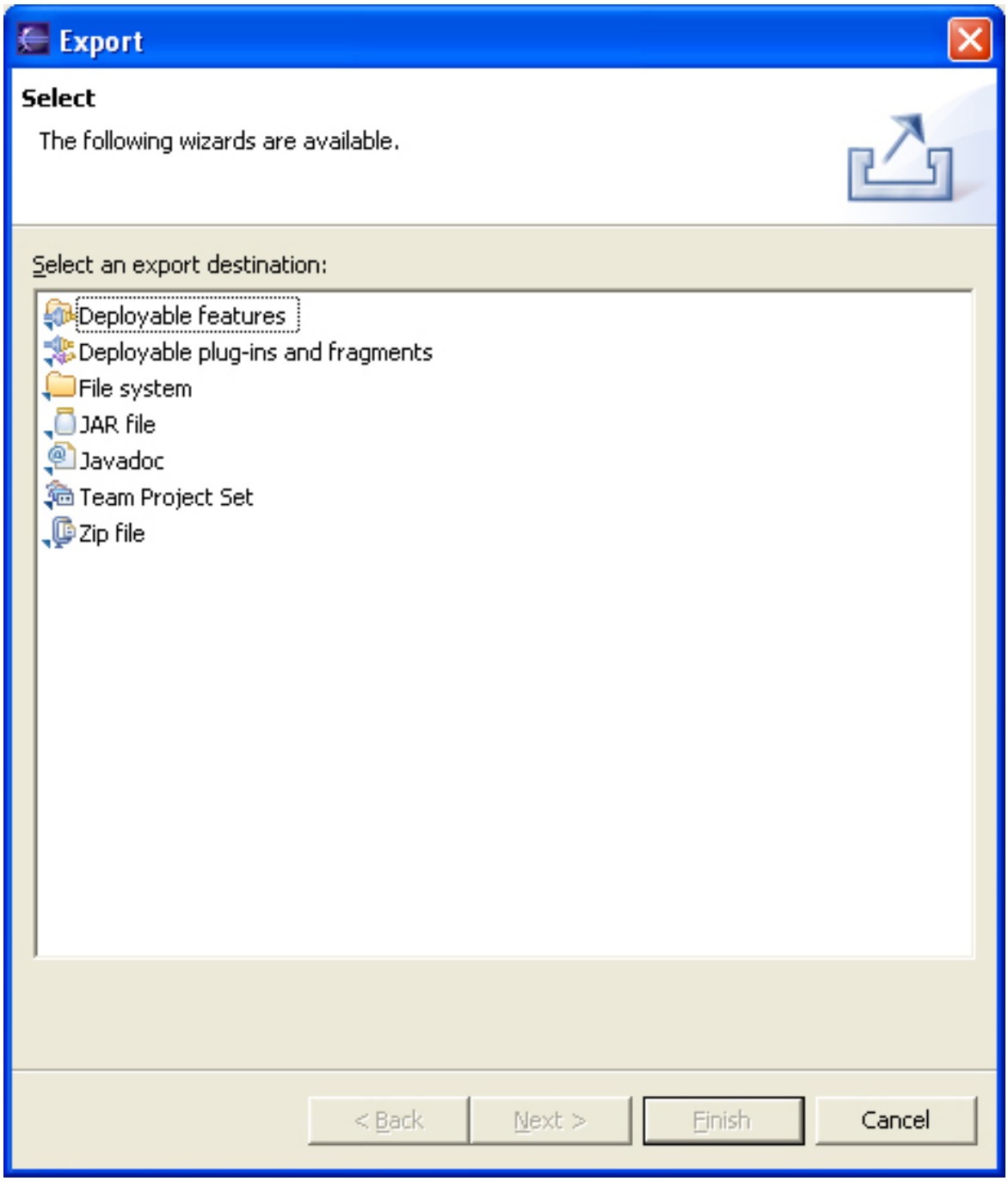
Before you run the Google application, verify that the project builds successfully and that you've received a license key to use Google's API.

Section 6. Launching the application

Exporting the application

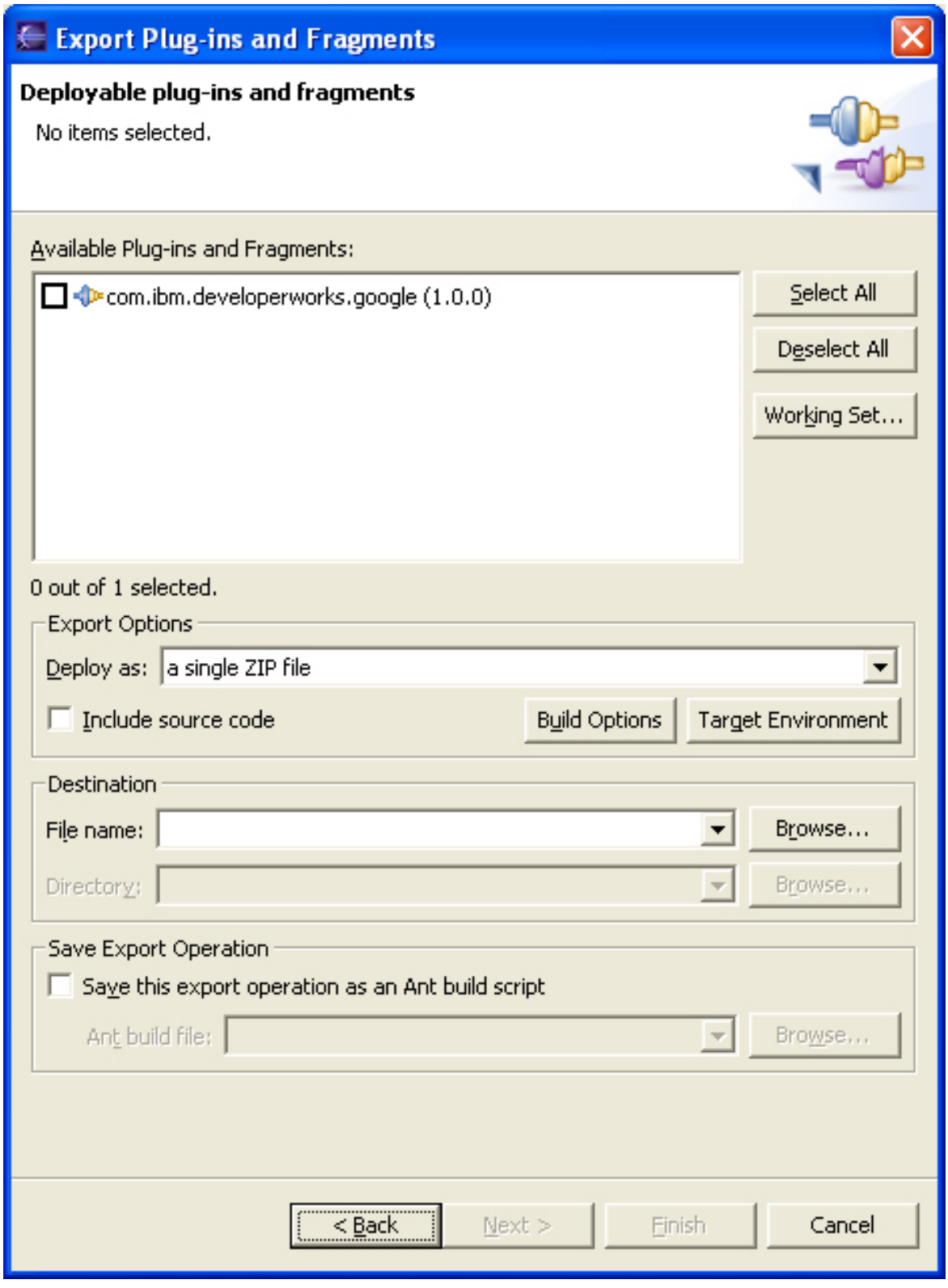
To create a stand-alone version of the Google application, complete the following steps within the Plug-in Development Environment:

1. Select **File > Export** from the menu bar to display the Export dialog.
   Figure 7. The Export dialog



2. Select **Deployable plug-ins and fragments** from the list of export options.

3. Click **Next** to display the Export Plug-ins and Fragments page of the Export wizard.
Figure 8. Export Plug-ins and Fragments page of the Export wizard



4. Verify that the **com.ibm.developerworks.google** plug-in is checked.

5. Select **a directory structure** under the Deploy as field.
6. Click the **Browse** button and choose an export location.
7. Click **Finish** to build the project.

Preparing the directory structure

To complete the stand-alone application, you need to copy some files from the Eclipse IDE directory into Google's export directory. Unfortunately, Eclipse 3.0 doesn't provide a tool to copy all the necessary dependent plug-ins and JAR files into the export directory.

Complete the following steps to prepare the directory structure:

1. Copy startup.jar from the root directory of the Eclipse 3.0 IDE to the root of the Google application's export directory.
2. Copy the following directories from the Eclipse 3.0 IDE plugin directory to the plugin directory of the Google application's export directory:

```
org.eclipse.core.expressions_3.0.0
org.eclipse.core.runtime_3.0.0
org.eclipse.help_3.0.0
org.eclipse.jface_3.0.0
org.eclipse.osgi.services_3.0.0
org.eclipse.osgi.util_3.0.0
org.eclipse.osgi_3.0.0
org.eclipse.swt.win32_3.0.0 (Windows only)
org.eclipse.swt.gtk_3.0.0 (Linux only)
org.eclipse.swt_3.0.0
org.eclipse.ui.workbench_3.0.0
org.eclipse.ui_3.0.0
org.eclipse.update.configurator_3.0.0
```

Testing and executing the application

After you've completed the task of preparing the directory, your export directory should have the following structure:

```
- google.bat (Windows only)
- google.sh (Linux only)
- startup.jar
+ ----- plugins
        + ----- org.eclipse.core.expressions_3.0
        + ----- org.eclipse.core.runtime_3.0.0
        + ----- org.eclipse.help_3.0.0
        + ----- org.eclipse.jface_3.0.0
        + ----- org.eclipse.osgi.services_3.0.0
        + ----- org.eclipse.osgi.util_3.0.0
        + ----- org.eclipse.osgi_3.0.0
        + ----- org.eclipse.swt.win32_3.0.0 (Windows only)
        + ----- org.eclipse.swt.gtk_3.0.0 (Linux only)
        + ----- org.eclipse.swt_3.0.0
```

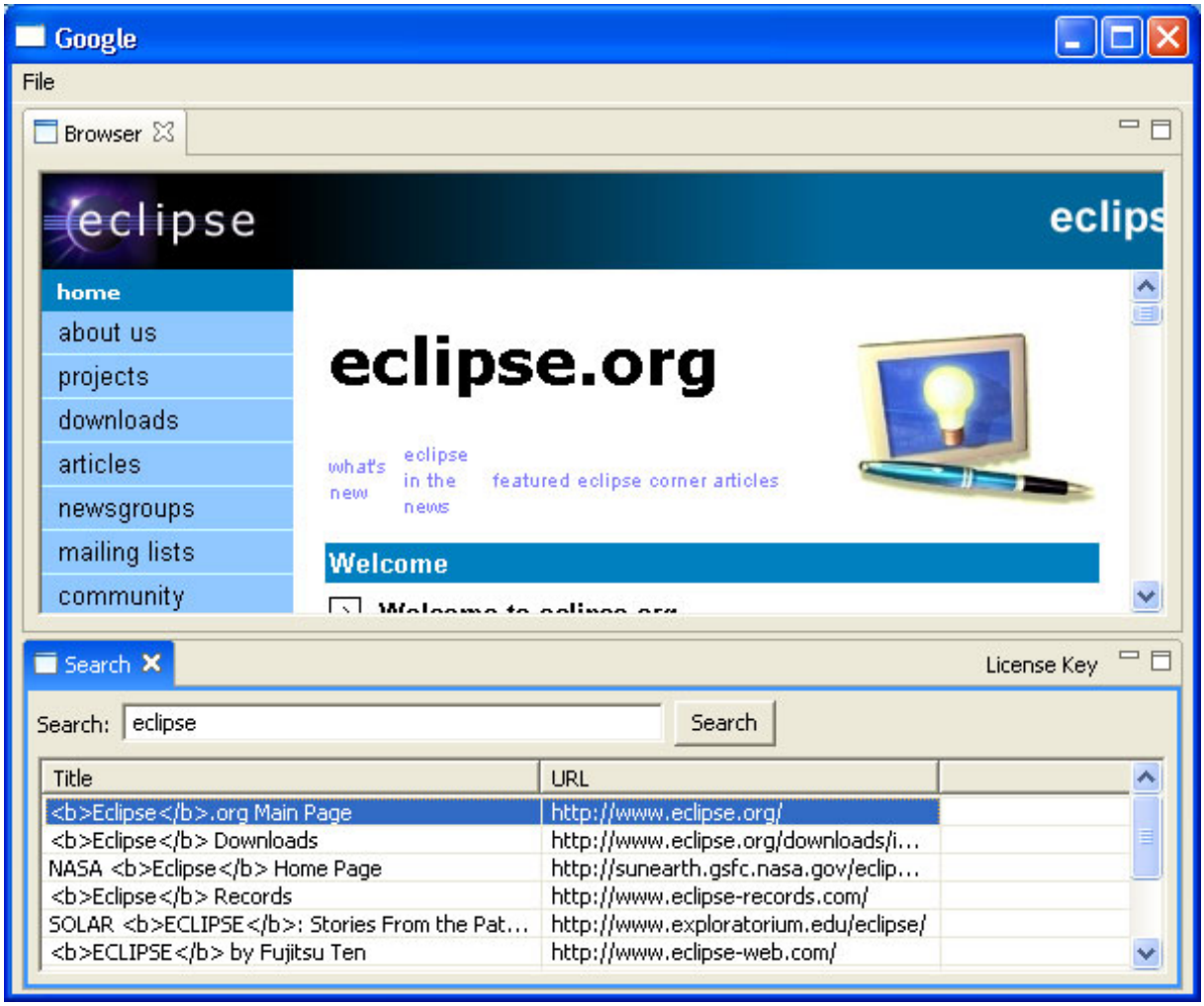
```
+ ----- org.eclipse.ui.workbench_3.0.0
+ ----- org.eclipse.ui_3.0.0
+ ----- org.eclipse.update.configurator_3.0.0
```

To test the application, you need to create a launch script. Using your favorite text editor, create a file named google.bat (Windows) or google.sh (Linux) with the following content:

```
java -cp startup.jar org.eclipse.core.launcher.Main
-application com.ibm.developerworks.google.GoogleApplication
```

With all the classes created, the plug-in manifest defined, and all of the necessary dependencies in place, you can launch the Google application and perform a search. Figure 9 illustrates how you can use the Google API to search for the term "eclipse" and how the Eclipse project Web site is displayed.

Figure 9. Google RCP application with search results



Section 7. Summary and resources

Summary

As the Eclipse development team begins to establish the RCP within the development landscape, it's going to be exciting to see how the platform's strategy and technology evolves. Although the RCP concept is very new, the 3.0 release of Eclipse delivers developers a framework they can start using today. The example Google application used in this tutorial demonstrates the generic workbench and explores how you can integrate various user-interface elements to create an elegant, cross-platform solution.

This series presented the following topics:

- An introduction to the core components that can make up an RCP application including Perspectives, Views, Actions, and Wizards.
- An exploration of how to develop an RCP through the use of extensions.
- A sample RCP application that you can use to query and display search results from Google.

Resources

- *Download* (part2-src.zip) the companion source code package for the sample RCP application demonstrated in this tutorial.
- Download *Eclipse 3.0* (http://www.eclipse.org/downloads/index.php) from the Eclipse Foundation.
- Download *Java 2 SDK, Standard Edition 1.4.2* (http://java.sun.com/j2se/1.4.2/download.html) from Sun Microsystems.
- Download *Ant 1.6.1* (http://ant.apache.org/) or higher from the Apache Software Foundation.
- Find more resources for how to use the Standard Widget Toolkit and JFace on the Eclipse Web site, including:
  - *Understanding Layouts in SWT* (Eclipse Website)
  - *Building and delivering a table editor with SWT/JFace* (Eclipse Website)
- Find more resources for how to use the Standard Widget Toolkit and JFace on *developerWorks*, including:
  - *Integrate ActiveX controls into SWT applications* (*developerWorks*, June 2003)
  - *Developing JFace wizards* (*developerWorks*, May 2003)
- Find more resources for how to use Eclipse on *developerWorks*, including:
  - *Developing Eclipse plug-ins* (*developerWorks*, December 2002)
  - *XML development with the Eclipse Platform* (*developerWorks*, April 2003)

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT

extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

For more information about the Toot-O-Matic, visit www-106.ibm.com/developerworks/xml/library/x-toot/ .