

Building Eclipse plug-ins using templates

Skill Level: Intermediate

[Christopher Judd \(cjudd@juddsolutions.com\)](mailto:cjudd@juddsolutions.com)

Freelance Writer

Judd Solutions, LLC

28 Jun 2005

You may know that Eclipse is a framework meant for building other tools. You may also know that you can build your own plug-ins for Eclipse. But did you know that Eclipse comes with seven plug-in templates to get you started? This tutorial gives you a start-to-finish look at building a plug-in using the Hello World template, then introduces you to the other templates.

Section 1. Start here

Should I take this tutorial?

In this tutorial, you will learn how to build, package, and deploy different types of Eclipse plug-ins using the seven plug-in templates provided with Eclipse. Using these templates helps you by generating many of the necessary plug-in files and configurations, and will reduce your learning curve by providing simple, customized working examples.

What is this tutorial about?

This tutorial introduces you to Eclipse plug-in development. It walks you through the process of developing, packaging, and deploying a plug-in developed from the seven built-in plug-in templates. It discusses in detail each of the following plug-in templates to give you a broad understanding of the types of options available:

- Hello World
- Popup Menu
- Property Page
- View
- Perspective Extension
- Editor
- Multi-page Editor

You can begin writing Eclipse plug-ins by hand from scratch. However, using the templates can help you:

- Set up basic plug-in project structure
- Reduce coding
- Reduce your learning curve

Plug-in projects have a specific structure. Using the Plug-in Project wizard and templates helps to set up the required structure. In addition, it generates many of the necessary files and configurations. This project structure requires a `plugin.xml` file at the root of the project. Consider this file as a deployment descriptor for an Eclipse plug-in. It contains information about the plug-in's classpath, other plug-ins it depends on and which extension points the plug-in integrates into. In addition, the project must contain a class that extends `org.eclipse.ui.plugin.AbstractUIPlugin`. This class will be described in further detail later in this tutorial. And as with all Java™ technology projects, it must contain a source folder and output folder.

Using templates can significantly reduce the amount of typing. They generate many of the classes you need to get started and provide a simple implementation, usually a message dialog, to show you what an implementation might look like. The plug-in templates also update the `plugin.xml` file to include extension-points and required libraries.

The Eclipse platform itself has more than 1,900 classes and interfaces. Add the JDT, PDE, and J2SE and there is a lot to learn. The plug-in templates can reduce the learning curve and anxiety by providing simple, customized working examples in a few minutes.

Prerequisites

To gain the most from this tutorial, you should have a good working knowledge of

Eclipse. Before you start, make sure you have Eclipse downloaded and installed:
[Eclipse SDK V3.0.2](#).

Basic knowledge of Java is also helpful.

Section 2. Eclipse overview

Background

As described on the Eclipse Web site, Eclipse is a universal tool platform, an open, extensible, integrated development environment (IDE). Open and extensible are the operative words in this description. All development efforts do not involve the same types of applications, APIs, languages, or version control. So the Eclipse platform was architected from the beginning to enable individuals and organizations to extend the platform in any way they want using plug-ins. This is liberating since it enables us as developers to customize Eclipse through existing plug-ins or create our own to make ourselves productive.

Eclipse can boast of more than 850 commercial and open source plug-ins. These support just about every aspect of development. The Eclipse Web site even hosts many of its own popular plug-ins, including the new Web Tools Platform for developing J2EE and Web applications.

The [Resources](#) section contains a list of sites where you can find commercial and open source plug-ins.

Architecture

Eclipse itself has a layered architecture (see Figure 1), made up primarily of plug-ins. The platform does have a small micro-kernel used to bootstrap the plug-ins, but the rest of the platform is made up of plug-ins. This means anything Eclipse does you can do with the custom plug-ins you develop.

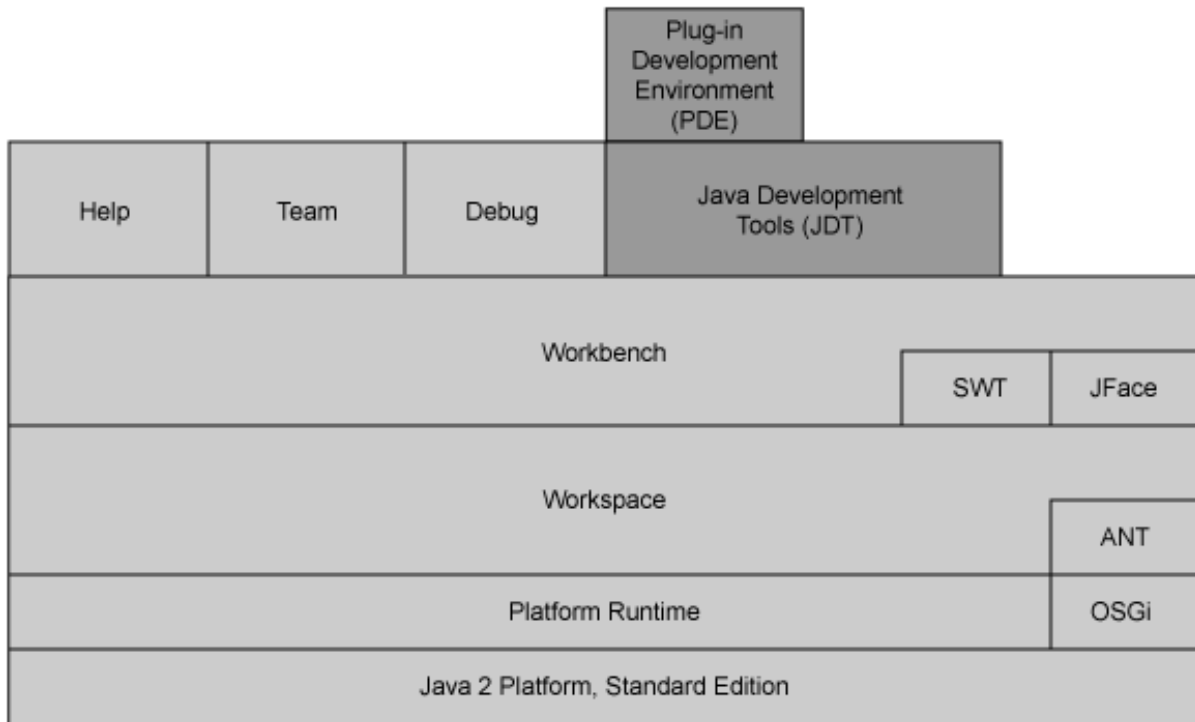
When you install the Eclipse SDK, you are installing three components that comprise the Eclipse project: the Eclipse Platform, Java Development Tools (JDT), and the Plug-in Development Environment (PDE). Each of these can be extended and have their own APIs. See [Resources](#) for links to each of the respective developer guides and APIs.

The focus of this tutorial is the tools found in the PDE, including the plug-in

templates. Knowing how to use these tools will enable you to extend Eclipse to meet your needs.

Figure 1 depicts the Eclipse Platform in the light gray. It is the foundation of Eclipse and the only required component. It sits on top of the Java 2 Standard Edition (J2SE) platform, allowing it to run on any operating system supported by the Java platform. Layered on top of the J2SE is the Eclipse platform run time, which includes the micro-kernel and the infrastructure for discovering and loading plug-ins. Until recently, this infrastructure was proprietary to Eclipse. However, Eclipse is in the process of switching to the standards-based OSGI Service Platform specification designed for dynamic services.

Figure 1. Eclipse architecture



The workspace acts as an abstract of the underlying file system. It is responsible for managing resources, such as projects, folders, and files. Also included within the workspace is the popular open source project: Apache Ant. The workbench is the graphical user interface used to interact with the workspace and run time. It also provides extensions for other visual plug-in integration. As you will read, SWT and JFace, Eclipse uses its own widget tool kit.

The Workbench includes three major extensions: Help, Debug, and Team. Help provides a mechanism for adding help to Eclipse's help system using HTML. Debug provides a debugger infrastructure that can be extended to support any language. Team is an extension for source code configuration management tools. It includes an implementation for CVS.

As shown in Figure 1, the JDT sits on top of the Eclipse Platform. The platform subproject provides an implementation for developing basic Java applications. It includes Java compiling and debugging. It also provides tools for editing, packaging, and unit-testing Java applications.

To make extending Eclipse easier, the SDK includes the PDE. The PDE includes the tool necessary for developing, packaging, and deploying Eclipse plug-ins. Because Eclipse is built on the J2SE, you need Java development tools to write plug-ins. Therefore, the PDE is built upon the JDT.

SWT and JFace

When the Eclipse Platform was originally developed, the architects felt the existing Java widget tool kits of Swing and Abstract Windows Toolkit (AWT) did not perform well enough or have the desired platform-native look and feel, so they created their own tool kit. This new widget tool kit and graphic library is the Standard Windows Toolkit (SWT).

Don't worry if you don't have any knowledge or experience with SWT or JFace. You won't need it to follow along. You will have to become familiar with both, however, as you get deeper into plug-in development.

In a lot of ways, SWT combines the best concepts of Swing and AWT. Where applicable, SWT uses native widgets of the underlying operating system much like AWT does. This provides the advantages of performance and a native application look and feel. Unfortunately, not all operating systems are created equal. So, rather than go with a least-common-denominator approach and only expose widgets common on all operating systems, SWT implements and renders its own version of the widgets if not supported by the underlying operating system. This is similar to how Swing implements all of its widgets.

To make SWT easier to use, a framework called JFace is layered on top of it. JFace provides a higher-level model, but at the same time does not hide SWT.

NOTE: SWT and JFace are frameworks independent of the Eclipse Platform and can be used as such to develop stand-alone desktop applications.

If you are familiar with Swing, the transition to SWT and JFace is a small learning curve. See [Resources](#) for links to articles and books about SWT and JFace.

Plug-in Development Environment tools

The PDE contains the tools necessary for developing plug-ins to extend Eclipse. You use these tools to develop, package, and deploy your plug-ins. These tools

include the plug-in templates that are the focus of this tutorial. They also include wizards, a Plug-in Development Perspective, views and special manifest editors.

The PDE includes seven wizards found in the Plug-in Development category of the New dialog. In this tutorial, we will use the Plug-in Project for setting up the project structure and running the first template, the Feature Project, for creating a plug-in feature to ease deployment, and the Update Site Project for creating an internal or external Web site for deployment.

Upon completing the Plug-in Project wizard, you will be prompted to switch to the Plug-in Development Perspective if you have not done so already. This perspective is helpful for developing plug-ins because it is similar to the Java Perspective, but includes two additional views specific to plug-in development: Plug-ins and Error Log. Additional PDE views can be found in the Show views dialog under the categories PDE and PDE Runtime.

The Plug-ins view contains a list of plug-ins available in your current instance of Eclipse. This view is helpful for archeology work. You can dig through other plug-ins, especially their plugin.xml files, to learn how other plug-ins accomplished what you want to accomplish. For example, you might ask how the Plug-ins view gets included in the Plug-in Development perspective.

The Error Log view displays the contents of the Eclipse log file found in the workspace's `.metadata` directory. This can be helpful to diagnose problems your new plug-in may be causing internal to the Eclipse platform.

Plug-ins require a manifest file with the name of plugin.xml. Later in this tutorial, you will learn more about this file. To make it easier to modify this file, the PDE includes a special multi-page editor called the Plug-in Manifest Editor (see Figure 2). It is associated with the plugin.xml file.

Figure 2. Plug-in Manifest Editor

The screenshot shows the Eclipse Plug-in Manifest Editor interface. The title bar indicates the project is 'com.ibm.tutorial.templates'. The main content is divided into four sections:

- General Information:** This section describes general information about the plug-in. It contains a table with the following fields:

ID:	com.ibm.tutorial.templates
Version:	1.0.0
Name:	Templates Plug-in
Provider:	IBM
Class:	com.ibm.tutorial.templa <input type="button" value="Browse..."/>
- Plug-in Content:** This section describes the content of the plug-in, which is made up of four sections:
 - Dependencies:** lists all the plug-ins required on this plug-in's classpath to compile and run.
 - Runtime:** lists the libraries that make up this plug-in's runtime.
 - Extensions:** declares contributions this plug-in makes to the platform.
 - Extension Points:** declares new function points this plug-in adds to the platform.
- Testing:** This section provides instructions on how to test the plug-in. It includes two links:
 - [Launch a runtime workbench](#)
 - [Launch a runtime workbench in Debug mode](#)
- Deploying:** This section provides instructions on how to deploy the plug-in:
 - Specify what needs to be packaged in the deployable plug-in on the [Build Configuration](#) page
 - Export the plug-in in a format suitable for deployment using the [Export Wizard](#)

At the bottom of the editor, there is a navigation bar with tabs for: Overview, Dependencies, Runtime, Extensions, Extension Points, Build, plugin.xml, and build.properties. The 'Overview' tab is currently selected.

We will use this editor throughout this tutorial. We will primarily focus on the Overview, Extensions, and plugin.xml pages.

The first page of the Plug-in Manifest Editor provides an overview of the plugin.xml file. It includes general information about your plug-in. It also contains links to the other pages from the Plug-in Content area. The Testing section provides an easy means to run and debug plug-ins. A shortcut to the export wizard for packaging the plug-in is provided in the Deploying section.

The Extension page can be used to run the plug-in templates on an existing plug-in project. The plugin.xml page is a text editor for the entire contents of the plugin.xml file.

The PDE also includes a large amount of documentation for extending the Eclipse Platform, JDT, and PDE. They are integrated into Eclipse's regular help. They

include developer guides and Javadoc.

Section 3. Create a plug-in with the Hello World template

Overview

In this section, we will learn how to write a plug-in that adds a menu and menu item to the Eclipse main menu, and a button to the toolbar. We will use the Hello World template as a starting point and modify the behavior to copy the active editor's absolute file name to the operating system's clipboard. In addition, since this is the first plug-in we are covering, we will walk through the process of developing an Eclipse plug-in. So by the end of this section, you should know how to write, debug, package, and deploy Eclipse plug-ins in general.

Building a plug-in

The process of creating a plug-in is not all that different from developing a regular Java application with Eclipse:

1. Create Plug-in Project
2. Create plug-in using template
3. Test and debug plug-in
4. Create a Feature Project
5. Package plug-in
6. Deploy plug-in

The first two steps can actually be done in a single step. The Plug-in Project wizard is one opportunity to create a plug-in from a template. This is how we will create the plug-in from the Hello World template. Later in the tutorial, we will see how to create a plug-in for an existing plug-in project.

Testing and debugging an Eclipse plug-in is rather fun. It is really not much different from debugging or testing a regular Java application in Eclipse, except that when you click the Run or Debug button, it launches another instance of Eclipse. Learning

how to use the debugger in Eclipse is out of scope here. See [Resources](#) for a debugging article if you are unfamiliar with debugging in Eclipse.

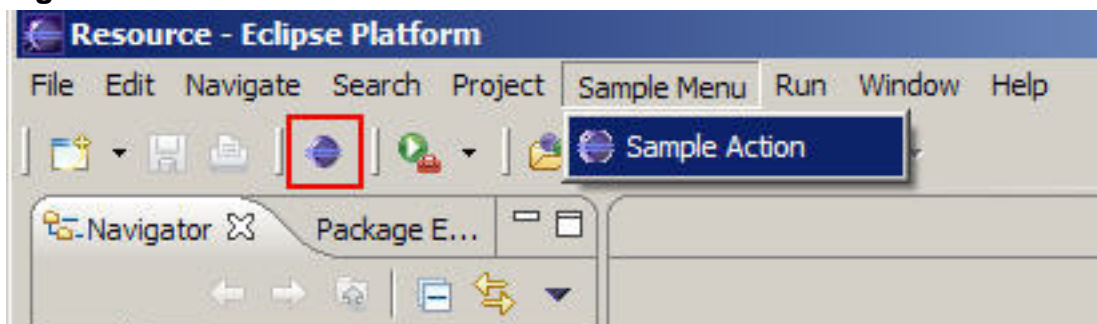
Ultimately, you will want to share your plug-in with your friends, co-workers, and possibly the rest of the Eclipse community. To do that, you will want to create a feature. A feature is a collection of one or more plug-ins and fragments. Features are not required, but can make installation and managing your plug-in easier for your users, and make packaging multiple plug-ins a breeze.

Packaging a plug-in or a collection of plug-ins as a feature follows the Eclipse export model and is just like exporting a JAR or Javadoc. Once you have packaged your plug-ins, you can deploy them as a zip file created from the packaging process or make them available on an internal or external Web site using Update Site Project.

Default Hello World template plug-in behavior

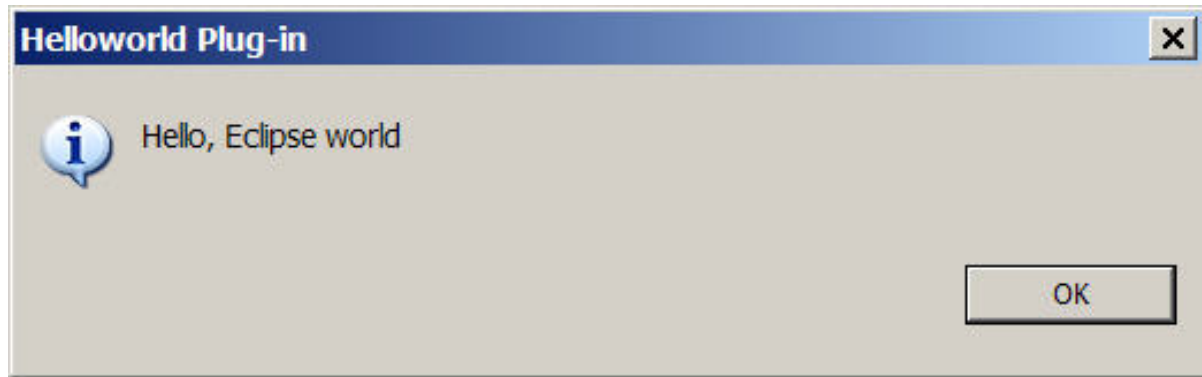
The default behavior of the plug-in that results from starting with the Hello World template is shown in Figure 3. The plug-in adds a main menu item to the Eclipse main menu and a menu item under that. In addition, a button is added to the toolbar.

Figure 3. Hello World menu and toolbar button



Notice in Figure 3, the default Hello World template adds a Sample Menu and underneath it a Sample Action. It also adds a button to the toolbar with the Eclipse icon on it like the Sample Action has. When the Sample Action or toolbar button is clicked, the modal dialog in Figure 4 is displayed.

Figure 4. Hello World message dialog



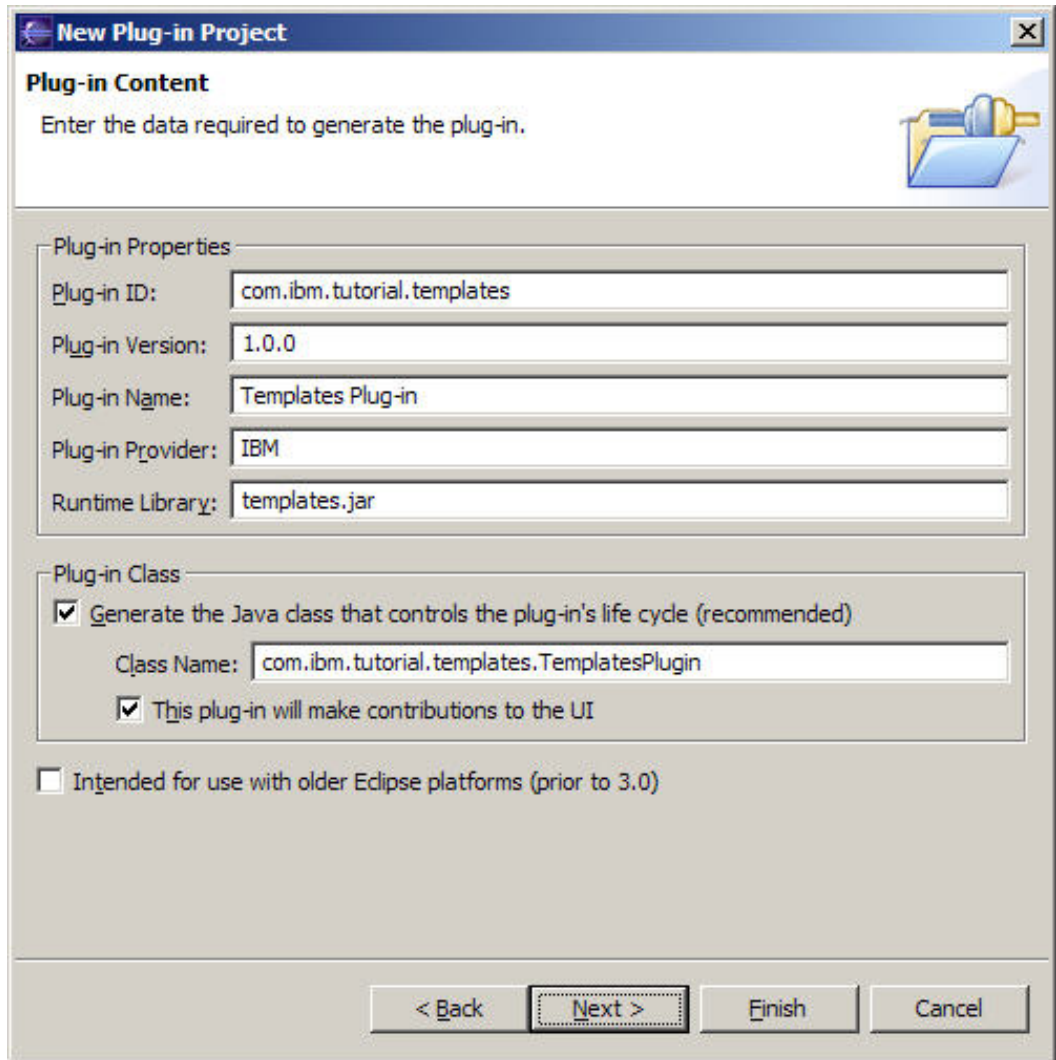
The dialog shown in Figure 4 is not very interesting, so we will modify the behavior to make it useful. We will change the behavior to locate the currently active editor, get its absolute file name, and copy it to the operating system clipboard.

Create Plug-in Project and run Hello World template

To create the Plug-in Project and the initial plug-in from the Hello World template, complete the following:

1. Select **File > New Project > Plug-in Development > Plug-in Project** to start the Plug-in Project wizard.
2. On the Plug-in Project page, enter a project name of `com.ibm.tutorial.templates`. By convention, plug-in project names have a similar format to a Java package name. This is helpful because when they are installed, they are mixed together with other plug-ins. This fully qualified name makes them easier to identify.
3. Click **Next**. The defaults for the Project Contents, Project Settings and Alternate Format are fine.
4. On the Plug-in Content page (see Figure 5), click **Next**. The defaults are fine unless you want to provide a different Plug-in Name, Provider, or Class Name.

Figure 5. Plug-in Content page



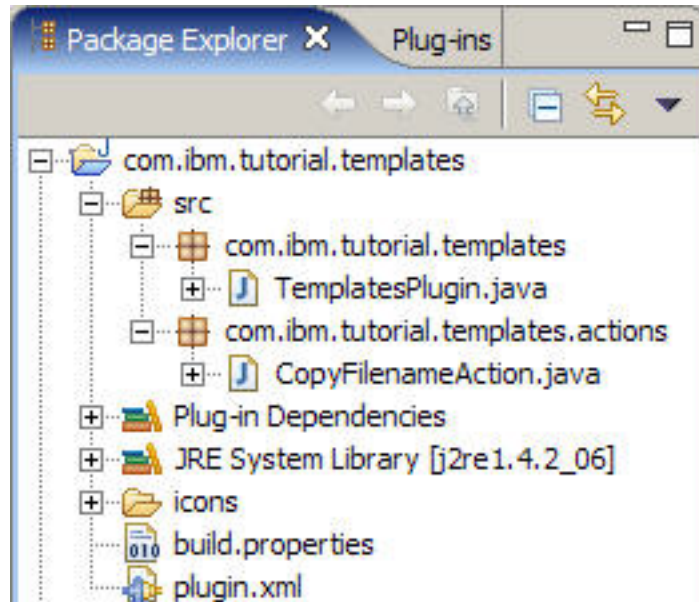
5. On the Templates page, check **Create a plug-in using one of the following templates** and select Hello World from the Available Templates.
6. Click **Next**.
7. On the Sample Action Set page, enter an Action Class Name of CopyFilenameAction.
8. Click **Finish** to complete the wizard. The rest of the defaults are fine.

Plug-in project structure overview

Upon completing the Plug-in Project wizard and Hello World template, you will have

a new Eclipse plug-in project. This project contains two classes and the plugin.xml file. It also contains an icon and build.properties file. See Figure 6 to review the contents of the newly created `com.ibm.tutorial.templates` plug-in project.

Figure 6. Results of the Plug-in Project and Hello World template



Plug-in class

As mentioned, each Eclipse plug-in must contain a class that extends `org.eclipse.ui.plugin.AbstractUIPlugin` (see Figure 7). Therefore, the Plug-in Project wizard generates this class based on the Plug-in Class Name. For this project, that class is `TemplatesPlugin`.

Figure 7. TemplatesPlugin inheritance hierarchy



Figure 7 shows the hierarchy and significance of the generated `Plugin` class to

your plug-in. Because of this class' inherited relationship with `org.eclipse.ui.plugin.AbstractUIPlugin` and `org.eclipse.core.runtime.Plugin`, it provides access to the Eclipse log, plug-in preferences, and the Workbench, as well as whether the plug-in is in debug mode or not. Also, since it implements the `org.osgi.framework.BundleActivator` interface, it has the plug-in's life-cycle methods of start and stop. Later, we will see where this class is configured in the `plugin.xml` so Eclipse can instantiate it.

```
package com.ibm.tutorial.templates;

import org.eclipse.ui.plugin.*;
import org.osgi.framework.BundleContext;
import java.util.*;

/**
 * The main plugin class to be used in the desktop.
 */
public class TemplatesPlugin extends AbstractUIPlugin {
    //The shared instance.
    private static TemplatesPlugin plugin;
    //Resource bundle.
    private ResourceBundle resourceBundle;

    /**
     * The constructor.
     */
    public TemplatesPlugin() {
        super();
        plugin = this;
        try {
            resourceBundle = ResourceBundle.getBundle(
"com.ibm.tutorial.templates.TemplatesPluginResources");
        } catch (MissingResourceException x) {
            resourceBundle = null;
        }
    }

    /**
     * This method is called upon plug-in activation
     */
    public void start(BundleContext context) throws Exception
    {
        super.start(context);
    }

    /**
     * This method is called when the plug-in is stopped
     */
    public void stop(BundleContext context) throws Exception
    {
        super.stop(context);
    }

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

    /**
     * Returns the string from the plugin's resource bundle,
```

```

    * or 'key' if not found.
    */
    public static String getResourceString(String key) {
        ResourceBundle bundle =
            TemplatesPlugin.getDefault().getResourceBundle();
        try {
            return (bundle != null) ? bundle.getString(key):key;
        } catch (MissingResourceException e) {
            return key;
        }
    }

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

```

Notice the `TemplatesPlugin` class follows the singleton pattern. When accessing this class, use the `getDefault` method. Eclipse will instantiate this class on your behalf, so you are guaranteed this will return a valid instance. Also notice this class provides access to a plug-in-specific resource bundle where you can store internationalized labels.

Pop-up action

In addition to the `TemplatesPlugin.java`, the Hello Wizard template generated a `CopyFilenameAction` class. This class is the action that will be fired when the menu item or button is chosen.

```

package com.ibm.tutorial.templates.actions;

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

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

    /**
     * The constructor.
     */
    public CopyFilenameAction() {}

    /**
     * The action has been activated. The argument of the
     * method represents the 'real' action sitting
     * in the workbench UI.
     */
}

```

```
* @see IWorkbenchWindowActionDelegate#run
*/
public void run(IAction action) {
    MessageDialog.openInformation(
        window.getShell(),
        "Templates Plug-in",
        "Hello, Eclipse world");
}

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

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

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

This code shows the default behavior of the `run` method that displays the Hello World message dialog. The `run` method is the method we will change to implement the new desired behavior of copying the absolute file name of the active editor to the clipboard.

Plug-in Manifest file

The Plug-in Project wizard also generates a `plugin.xml` file shown in the code below. This file is for all intents and purposes the plug-in deployment descriptor or manifest file. It is read by Eclipse at startup and used to declaratively configure different aspects of the plug-in, including descriptive information like its ID, name, version, and provider. It also includes its classpath through the run time and requires elements. Lastly, it identifies which points it wants to extend.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin
    id="com.ibm.tutorial.templates"
    name="Templates Plug-in"
    version="1.0.0"
```

```

provider-name="IBM"
class="com.ibm.tutorial.templates.TemplatesPlugin">

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

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

<extension
  point="org.eclipse.ui.actionSets">
  <actionSet
    label="Sample Action Set"
    visible="true"
    id="com.ibm.tutorial.templates.actionSet">
    <menu
      label="Sample &Menu"
      id="sampleMenu">
      <separator
        name="sampleGroup">
      </separator>
    </menu>
    <action
      label="&Sample Action"
      icon="icons/sample.gif"
      class="com.ibm.tutorial.templates.actions.CopyFilenameAction"
      tooltip="Hello, Eclipse world"
      menubarPath="sampleMenu/sampleGroup"
      toolbarPath="sampleGroup"
      id="com.ibm.tutorial.templates.actions.CopyFilenameAction">
    </action>
  </actionSet>
</extension>

</plugin>

```

There are several interesting items in this code. First, notice `com.ibm.tutorial.templates.TemplatesPlugin` is declared in the `class` attribute of the `plugin` element. This is how Eclipse knows to instantiate and call the life-cycle methods of the `TemplatePlugin` class. Next, notice the `extension` element includes an `actionSet` element. An `actionSet` adds menus, menu items, and toolbar items to the Workbench menus and toolbar. The `actionSet` includes a menu and action. The menu causes the Sample Menu to appear on the main menu. It requires one or more separators to group actions. The action contains declarative information about an `Action` class. This includes the label, icon, and tool tip that should be used when displaying the action as a menu item or toolbar button.

This is all configured declaratively so Eclipse can lazy-load the actual class that implements the behavior based on the class attribute. Lazy-loading enables Eclipse to start faster and use less memory. The lazy-loading pattern is common in Eclipse plug-ins. For a menu to appear it must contain a `menubarPath` attribute that contains the menu/group in which it belongs. For a toolbar button to appear, it must contain a `toolbarPath` attribute with a group name.

Modifying extensions

To make this plug-in copy the file name of the active editor, we are first going to change the label and tool tip of the Sample Action to something more appropriate for the copy file name behavior. We will then want to modify the behavior of the action to copy the active editor's file name to the clipboard. To conclude, we will want to add required plug-ins so classes can be found on the classpath.

To modify the Sample Action's label and tool tip, complete the following:

1. Open the plugin.xml file by double-clicking on it
2. Select the **Extensions** tab
3. Expand the `org.eclipse.ui.actionSets` in the All Extensions tree until you see the Sample Action
4. Select the Sample Action; the Sample Action's properties will appear to the right in the Extension Element Details area
5. Change the label to **Copy Filename**
6. Change the tool tip to **Copy Active Editor's Filename to Clipboard**
7. Save plugin.xml by pressing **Ctrl+S**

Modifying action behavior

To modify the behavior of the `CopyFilenameAction`, replace the run method with the code shown here:

```
public void run(IAction action) {
    // get the active editor
    if(window.getActivePage().getActiveEditor() != null)
    {
        IEditorPart editor =
            window.getActivePage().getActiveEditor();
        // get the input editor that has reference to a file
        if(editor.getEditorInput() instanceof IFileEditorInput)
        {
            IFileEditorInput fileEditorInput =
                (IFileEditorInput) editor.getEditorInput();
            IFile file = fileEditorInput.getFile();
            // use location so the path is relative
            // to the file system and not Eclipse workspace
            IPath osPath = file.getLocation();

            ClipboardUtil.copyTo(
                osPath.toFile().getAbsolutePath());
        }
    }
}
```

```

    }
}

```

The behavior of displaying the message dialog has been changed to copying the absolute file name of the active editor to the clipboard. This happens by first getting the active editor. To get the active editor, we must use the `org.eclipse.ui.IWorkbenchWindow` instance passed into the `init` method and stored in the private `window` field. Using the window reference, we can call `getActivePage` and `getActiveEditor` to return an instance of `org.eclipse.ui.IEditorPart`. `IEditorPart` is the window that contains all the editors. Once we have `IEditorPart`, we can make sure it is an instance of `IFileEditorInput` and typecast it as such. We need to use `IFileEditorInput` because it contains a `getFile` method that returns an `org.eclipse.core.resources.IFile`. `IFile` is relative to the current workspace, so in order to get the file name relative to the operating system, we must use `IFile`'s `getLocation`. Once we have the location, we can use it as a `java.io.File` with the `toFile` method and call its `getAbsolutePath`. The absolute path is then passed to a `copyTo` method on a utility class called `ClipboardUtil`, as shown below:

```

package com.ibm.tutorial.util;

import org.eclipse.swt.dnd.Clipboard;
import org.eclipse.swt.dnd.TextTransfer;
import org.eclipse.swt.dnd.Transfer;
import org.eclipse.swt.widgets.Display;

/**
 * Utility methods for working with the Clipboard.
 * @author juddcl
 */
public class ClipboardUtil {

    /**
     * Copies Text to the clipboard.
     * @param text
     */
    public static void copyTo(String text) {
        Clipboard clipboard =
            new Clipboard(Display.getCurrent());
        TextTransfer textTransfer = TextTransfer.getInstance();
        Transfer[] transfers = new Transfer[] { textTransfer };
        Object[] data = new Object[] { text };
        clipboard.setContents(data, transfers);
    }
}

```

This code is a simple utility class used to abstract the complexities of copying a `String` to the clipboard away from other code, such as the `CopyFilenameAction`.

Modifying dependencies

If you are following along and typing this example in your editor, you may have noticed that some of the classes are not found. Even if you try to add the imports, you still get a "class not resolved" error. This is because the classes are not included in the classpath. In a regular Java project, you would use the Java Build Path to include the JARs containing the appropriate classes to the project's libraries. This is not the case, however, for a plug-in project. Instead, you must add dependencies on other plug-ins using the Dependencies tab of the Plug-in Manifest Editor.

Complete the following steps to add dependencies on the `org.eclipse.core.resources` and `org.eclipse.ui.ide` plug-ins:

1. Open the Plug-in Manifest Editor by double-clicking the `plugin.xml` file in the Package Explorer
2. Select the **Dependencies** tab at the bottom of the editor window
3. Click **Add** in the Required Plug-ins area
4. Multi-select the `org.eclipse.core.resources` and `org.eclipse.ui.ide` plug-ins
5. Click **OK**
6. Save `plugin.xml` by pressing **Ctrl+S**

Run and debug

Now that the behavior has been modified and all the required plug-ins have been configured, we are ready to run and/or debug the plug-in. This process is really no different from running or debugging a regular Java application in Eclipse. You can use the Run and Debug buttons on the toolbar, launch a run-time workbench, or Launch a run-time workbench in Debug mode.

Under the covers, Eclipse creates a new launcher based on the Run-time Workbench launch configuration. This launcher starts another instance of Eclipse with your plug-in installed.

NOTE: Learning to use the debugger is outside the scope of this article. If you are new to the Eclipse debugger, see the article listed in [Resources](#).

To test the plug-in:

1. Click **Run** on the toolbar

2. When the second instance of Eclipse starts, create a new project, and add some classes to it
3. Open one of the classes in the editor
4. Click **Copy Active Editor's Filename to Clipboard** on the toolbar
5. In the editor, paste the contents of the Clipboard on a blank line; you should see the fully qualified name of the file
6. Close the second instance of Eclipse

Creating a feature

A feature is a collection of plug-ins. You may be asking yourself: "if we only have one plug-in, why create a feature?" Well, features can make installation easier and are required by an Update Site, which we will create in the next section. Features also provide a good place for including a description, URL, copyright, and license agreement for your plug-in.

To make a feature for this project:

1. Select **File > New Project > Plug-in Development > Feature Project** to start the Feature Project wizard
2. On the Feature Name page, enter a project name of `com.ibm.tutorial`
3. Click **Next**
4. On the Feature Properties, keep the defaults by clicking **Next**
5. On the Referenced Plug-ins and Fragments page, check the `com.ibm.tutorial.templates` plug-in
6. Click **Finish**

Now that you have a feature for your plug-in, you can export your feature using the Feature Manifest Editor's Overview. This will enable you to create a package for your friends and family. The export options are to export as a single zip file, a directory structure, or a JAR for an update site.

Creating an Update Site

Distributing an exported feature as a zip file or directory might work for a small amount of people, but for large organizations or to share with the Eclipse community at large, you will want to create an Update Site. An Update Site is basically a site.xml file and the packaged features and plug-ins on a Web server. The site.xml file tells Eclipse where to locate the feature and plug-in JARs.

For others to install your plug-in, they can use the Find and Install option under the **Help > Software Updates** menu.

To create an Update Site:

1. Select **File > New Project > Plug-in Development > Update Site Project** to start the Update Site Project wizard
2. On the Update Site Project page, enter a project name of `com.ibm.tutorial.site`
3. Check the "Generate a Web page listing all available features"
4. Click **Finished**
5. In the Site Manifest Editor's Features to Build area, click **Add**
6. Check the `com.ibm.tutorial` feature in the Available Features list and click **Finish**
7. Drag the `com.ibm.tutorial` from the Features to Build list to the Features to Publish list
8. Click **Build All** on the Features to Build list to build and package all the features in the list

You have now built an Update Site. To deploy it, copy the contents of your project to a Web server, such as the Apache Web Server or Apache Tomcat.

Now that we understand how to write, test, package, and deploy plug-ins, we will explore the remaining plug-in templates. Through the remainder of this tutorial, we will add the existing plug-in to our current project.

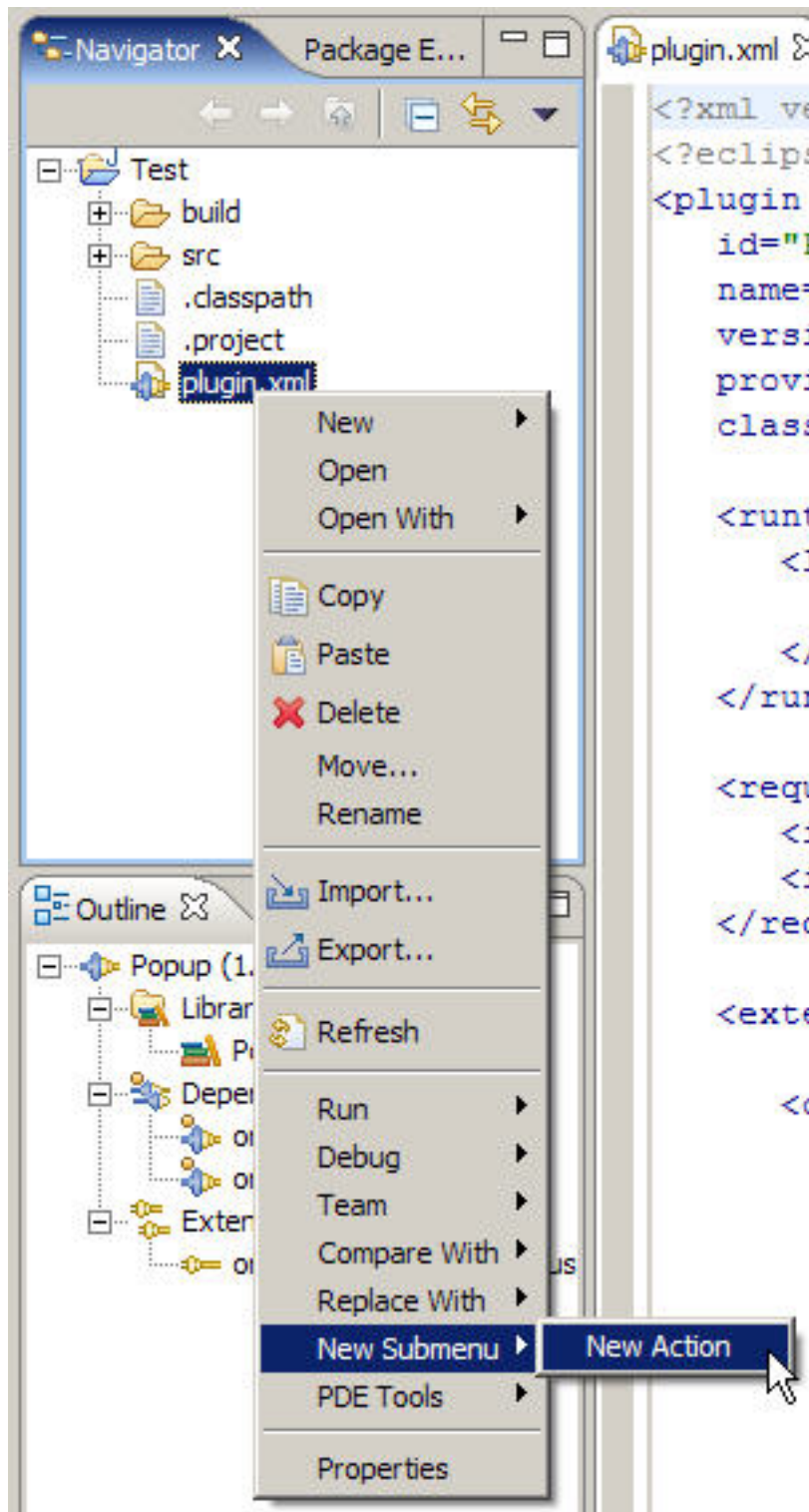
Section 4. Popup Menu template

Default Popup Menu template behavior

In this section, we will use the Popup Menu template as a starting point and modify the behavior to copy a file object's absolute file name to the operating system's clipboard.

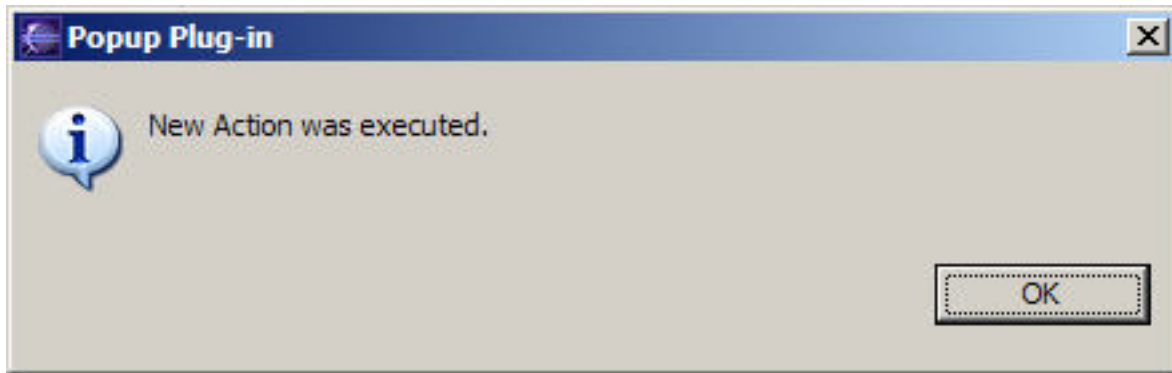
The default behavior of the Popup Menu template plug-in is to add a **New Submenu > New Action** menu to the plugin.xml files context menu (see Figure 8).

Figure 8. Popup Menu template menu



When New Action is selected, the dialog shown in Figure 9 is displayed.

Figure 9. Popup Menu message dialog



The dialog shown in Figure 9 is not very interesting, so we will modify the behavior to make it useful. We will change the behavior to copy the file's absolute file name to the operating system clipboard.

Run Popup Menu template

When we ran the Hello World template, we ran it as part of creating a new Plug-in Project. However, if we already have an existing project, we can still use the templates. For this template and the rest, we will use templates to extend our existing `com.ibm.tutorial.templates` project by running the template wizards from the Plug-in Manifest Editor.

To run the Popup Menu template:

1. Open the Plug-in Manifest Editor by double-clicking the `plugin.xml` file
2. Select the **Extensions** tab at the bottom of the editor
3. On the Extensions tab, click **Add**
4. On the New Extension dialog, select **Extension Wizards**
5. Select **Extension Templates** to display the seven templates covered in this tutorial, as well as the Help Content and Preference Page templates
6. Select the Popup Menu template and click **Next**
7. On the Sample Popup Menu page, set the Name Filter to `*`, the Action Label to `Copy Filename`, and the Action Class to `CopyFilenameAction`
8. Click **Finish**

Review Popup Menu template results

After running the Popup Menu template, a new `CopyFilenameAction.java` is added to the project, and a new extension point is added to `plugin.xml`.

The code for the `CopyFilenameAction` Popup Menu:

```
package com.ibm.tutorial.templates.popup.actions;

import org.eclipse.jface.action.IAction;
import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.ui.IObjectActionDelegate;
import org.eclipse.ui.IWorkbenchPart;

public class CopyFilenameAction
    implements IObjectActionDelegate {

    /**
     * Constructor for Action1.
     */
    public CopyFilenameAction() {
        super();
    }

    /**
     * @see IObjectActionDelegate#setActivePart(
     *      IAction, IWorkbenchPart)
     */
    public void setActivePart(IAction action,
        IWorkbenchPart targetPart) {}

    /**
     * @see IActionDelegate#run(IAction)
     */
    public void run(IAction action) {
        Shell shell = new Shell();
        MessageDialog.openInformation(
            shell,
            "Templates Plug-in",
            "Copy Filename was executed.");
    }

    /**
     * @see IActionDelegate#selectionChanged(
     *      IAction, ISelection)
     */
    public void selectionChanged(IAction action,
        ISelection selection) {}
}
```

The code is an action that implements the `org.eclipse.ui.IObjectActionDelegate` interface. Its behavior is similar to that of the default Hello World template. Next, we will modify this behavior to copy the absolute file name to the operating system clipboard.

The code for the Popup Menu extension point added to `plugin.xml`:

```
<extension point="org.eclipse.ui.popupMenus">
    <objectContribution
```

```

    objectClass="org.eclipse.core.resources.IFile"
    nameFilter="*"
    id="com.ibm.tutorial.templates.contribution1">
<menu
    label="New Submenu"
    path="additions"
    id="com.ibm.tutorial.templates.menu1">
    <separator name="group1"/>
</menu>
<action
    enablesFor="1"
    label="Copy Filename"
    class=
"com.ibm.tutorial.templates.popup.actions.CopyFilenameAction"
    menubarPath=
"com.ibm.tutorial.templates.menu1/group1"
    id="com.ibm.tutorial.templates.newAction"/>
</objectContribution>
</extension>

```

With this, CopyFilenameAction is added to any object of any view, including the Package Explorer and Resource Navigator that is an instance of the org.eclipse.core.resources.IFile interface. Just like the Hello World Template, it creates a menu and group. The action itself is then associated with the menu and group using the menubarPath.

Modify Popup Menu behavior

Make the following highlighted changes to modify the behavior of the Popup Menu to copy the absolute file name:

```

package com.ibm.tutorial.templates.popup.actions;

import org.eclipse.core.resources.IFile;
import org.eclipse.core.runtime.IPath;
import org.eclipse.jface.action.IAction;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.jface.viewers.IStructuredSelection;
import org.eclipse.ui.IObjectActionDelegate;
import org.eclipse.ui.IWorkbenchPart;

import com.ibm.tutorial.util.ClipboardUtil;

/**
 * Popup menu to copy the absolute filename to the
 * clipboard.
 */
public class CopyFilenameAction implements
    IObjectActionDelegate
{
    private ISelection selection;

    /**
     * @see IObjectActionDelegate#setActivePart(
     *      IAction, IWorkbenchPart)
     */
    public void setActivePart(
        IAction action, IWorkbenchPart targetPart) {}

```

```
/**
 * @see IActionDelegate#run(IAction)
 */
public void run(IAction action) {
    IFile file = getFile();
    IPath osPath = file.getLocation();
    ClipboardUtil.copyTo(
        osPath.toFile().getAbsolutePath());
}

/**
 * @see IActionDelegate#selectionChanged(
 *     IAction, ISelection)
 */
public void selectionChanged(
    IAction action, ISelection selection)
{
    this.selection = selection;
}

private IFile getFile() {
    return (IFile)
        ((IStructuredSelection)selection)
            .getFirstElement();
}
}
```

We have now added the private field of selection. This stores a collection of the currently selected items in the tree. It is set in the `selectionChanged` method called anytime a new item in the tree is selected.

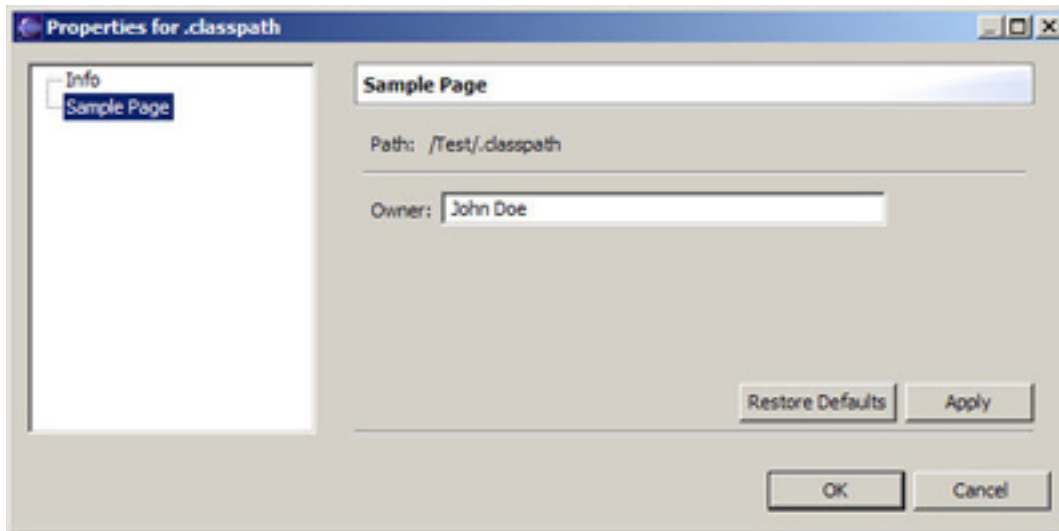
A `getFile` method was added to simplify typecasting the first item in the selection to the desired `IFile`.

The `run` method used above is similar to the implementation of the Hello World template example shown earlier. The only difference is that the file reference is obtained from the `getFile` method in this instance. Like before, it uses `getLocation` to get the file name relative to the operating system, rather than the workspace. Then we get a reference to `java.io.File` and pass its absolute file name to the `ClipboardUtil` class.

Section 5. Property Page template

Default Property Page template behavior

The default Property Page template creates the properties page shown in Figure 10. This page appears when you right-click on a file and choose Properties.

Figure 10. Default Property Page

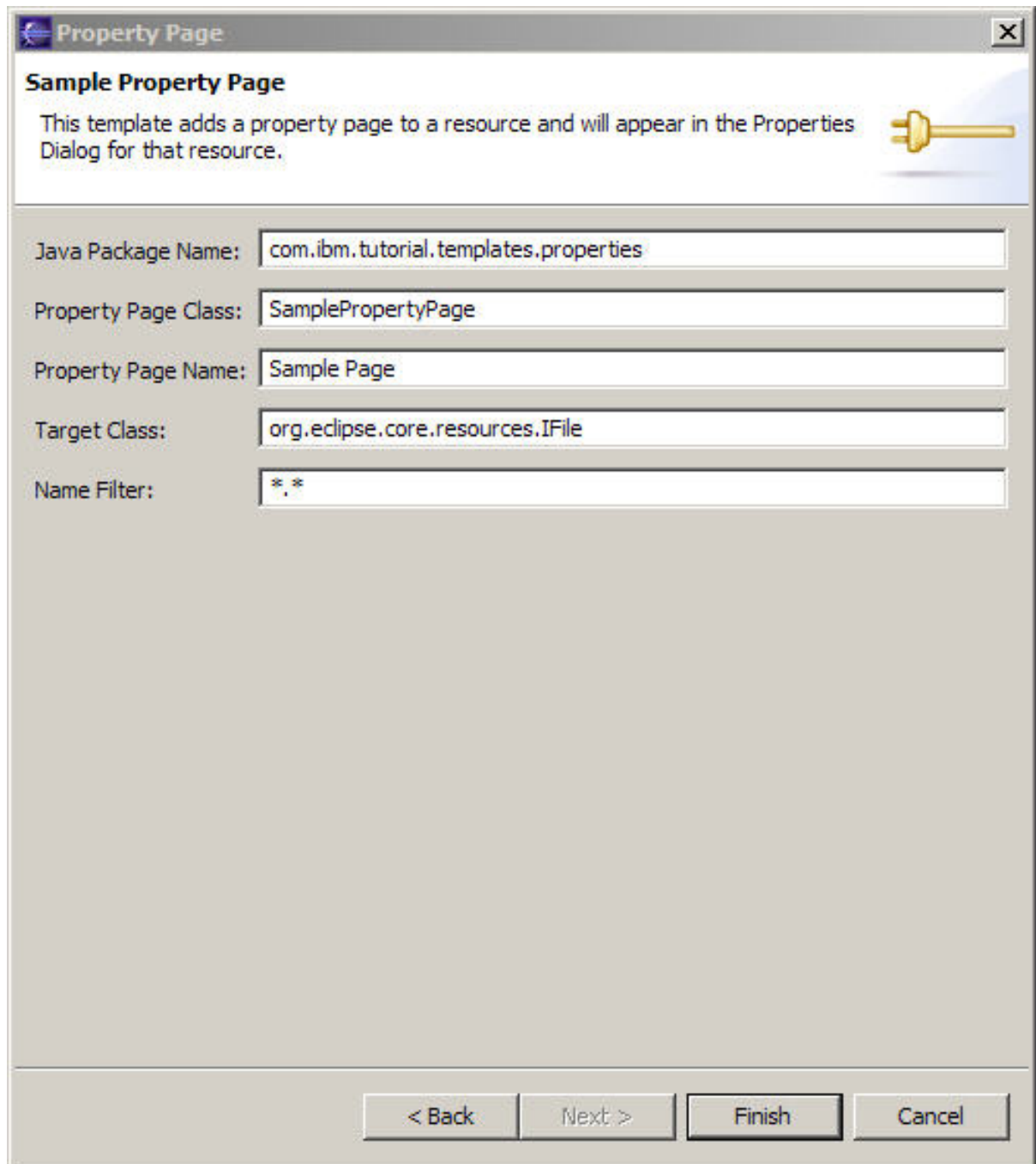
As you can see, the Property Page displays the path, as well as the owner. What you cannot see, however, is that the owner can be modified and will be saved if **Apply** or **OK** is clicked. The Restore Defaults will also set the value of Owner back to John Doe if it has been changed. So, this template provides a good example of how to persist properties on an object.

Run Property Page template

To create a default Properties Page from the template in the existing project:

1. Open the Plug-in Manifest Editor by double-clicking the plugin.xml file
2. Select the **Extensions** tab at the bottom of the editor
3. On the Extensions tab, click **Add**
4. On the New Extension dialog, select **Extension Wizards**
5. Select **Extension Templates**
6. Select the **Property Page** template and click **Next**
7. Click **Finish** to accept the defaults shown in Figure 11

Figure 11. Sample Property Page defaults



Property Page

Sample Property Page

This template adds a property page to a resource and will appear in the Properties Dialog for that resource.

Java Package Name:

Property Page Class:

Property Page Name:

Target Class:

Name Filter:

< Back Next > Finish Cancel

Review Property Page template results

After running the Property Page Template, a new `SamplePropertyPage.java` is added to the project, and a new extension point is added to `plugin.xml`.

Adding `SamplePropertyPage.java`:

```

package com.ibm.tutorial.templates.properties;

import org.eclipse.core.resources.IResource;
import org.eclipse.core.runtime.CoreException;
import org.eclipse.core.runtime.QualifiedName;
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.Control;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Text;
import org.eclipse.ui.dialogs.PropertyPage;

public class SamplePropertyPage extends PropertyPage {

    private static final String PATH_TITLE = "Path:";
    private static final String OWNER_TITLE = "&Owner:";
    private static final String OWNER_PROPERTY = "OWNER";
    private static final String DEFAULT_OWNER = "John Doe";

    private static final int TEXT_FIELD_WIDTH = 50;

    private Text ownerText;

    /**
     * Constructor for SamplePropertyPage.
     */
    public SamplePropertyPage() {
        super();
    }

    private void addFirstSection(Composite parent) {
        Composite composite = createDefaultComposite(parent);

        //Label for path field
        Label pathLabel = new Label(composite, SWT.NONE);
        pathLabel.setText(PATH_TITLE);

        // Path text field
        Text pathValueText =
            new Text(composite, SWT.WRAP | SWT.READ_ONLY);
        pathValueText.setText(
            ((IResource) getElement()).getFullPath().toString());
    }

    private void addSeparator(Composite parent) {
        Label separator =
            new Label(parent, SWT.SEPARATOR | SWT.HORIZONTAL);
        GridData gridData = new GridData();
        gridData.horizontalAlignment = GridData.FILL;
        gridData.grabExcessHorizontalSpace = true;
        separator.setLayoutData(gridData);
    }

    private void addSecondSection(Composite parent) {
        Composite composite = createDefaultComposite(parent);

        // Label for owner field
        Label ownerLabel = new Label(composite, SWT.NONE);
        ownerLabel.setText(OWNER_TITLE);

        // Owner text field
        ownerText =
            new Text(composite, SWT.SINGLE | SWT.BORDER);
        GridData gd = new GridData();
        gd.widthHint =
            convertWidthInCharsToPixels(TEXT_FIELD_WIDTH);
    }
}

```

```

ownerText.setLayoutData(gd);

// Populate owner text field
try {
    String owner =
        ((IResource) getElement()).getPersistentProperty(
            new QualifiedName("", OWNER_PROPERTY));
    ownerText.setText((owner != null) ?
        owner : DEFAULT_OWNER);
} catch (CoreException e) {
    ownerText.setText(DEFAULT_OWNER);
}
}

/**
 * @see PreferencePage#createContents(Composite)
 */
protected Control createContents(Composite parent) {
    Composite composite = new Composite(parent, SWT.NONE);
    GridLayout layout = new GridLayout();
    composite.setLayout(layout);
    GridData data = new GridData(GridData.FILL);
    data.grabExcessHorizontalSpace = true;
    composite.setLayoutData(data);

    addFirstSection(composite);
    addSeparator(composite);
    addSecondSection(composite);
    return composite;
}

private Composite createDefaultComposite(
    Composite parent)
{
    Composite composite = new Composite(parent, SWT.NULL);
    GridLayout layout = new GridLayout();
    layout.numColumns = 2;
    composite.setLayout(layout);

    GridData data = new GridData();
    data.verticalAlignment = GridData.FILL;
    data.horizontalAlignment = GridData.FILL;
    composite.setLayoutData(data);

    return composite;
}

protected void performDefaults() {
    // Populate the owner text field with the default value
    ownerText.setText(DEFAULT_OWNER);
}

public boolean performOk() {
    // store the value in the owner text field
    try {
        ((IResource) getElement()).setPersistentProperty(
            new QualifiedName("", OWNER_PROPERTY),
            ownerText.getText());
    } catch (CoreException e) {
        return false;
    }
    return true;
}
}

```

First, notice the generated `SamplePropertyPage` class extends

`org.eclipse.ui.dialogs.PropertyPage`. Actually, it has a long lineage since `PropertyPage` extends `org.eclipse.jface.preference.PreferencePage`, and that extends `org.eclipse.jface.dialogs.DialogPage`. It is through this lineage that `SamplePropertyPage` gets the methods `createContents`, `performDefaults`, and `performOk`, which it overrides.

The `createContents` method is called when the property page is displayed. It is overridden to create the Composite for holding all the components displayed on the page. Internally, it does some layout work, then delegates to the private methods of `addFirstSection`, `addSecondSection`, and `addSeparator` to add the actual components.

The `performDefaults` method is executed when the Restore Defaults button is clicked. It also overrides a super-class method. Its implementation populates the Text box to the default of John Doe.

The `performOk` method is executed when the OK or Apply button is clicked. It first calls `getElement` implemented by a super class. The `getElement` returns the file object originally selected. Using the file as a resource, it calls `setPersistentProperties`. This saves the value of the owner Text box in the workspace with a relationship to the file.

The extension point in the following code instructs Eclipse to add the `SamplePropertyPage` to every file property that has a filter of `*.*`. This would include all views, such as the Package Explorer and Resource Navigation.

```
<extension point="org.eclipse.ui.propertyPages">
  <page
    objectClass="org.eclipse.core.resources.IFile"
    nameFilter="*.*"
    class=
"com.ibm.tutorial.templates.properties.SamplePropertyPage"
    name="Sample Page"
    id=
"com.ibm.tutorial.templates.properties.samplePropertyPage"
  />
</extension>
```

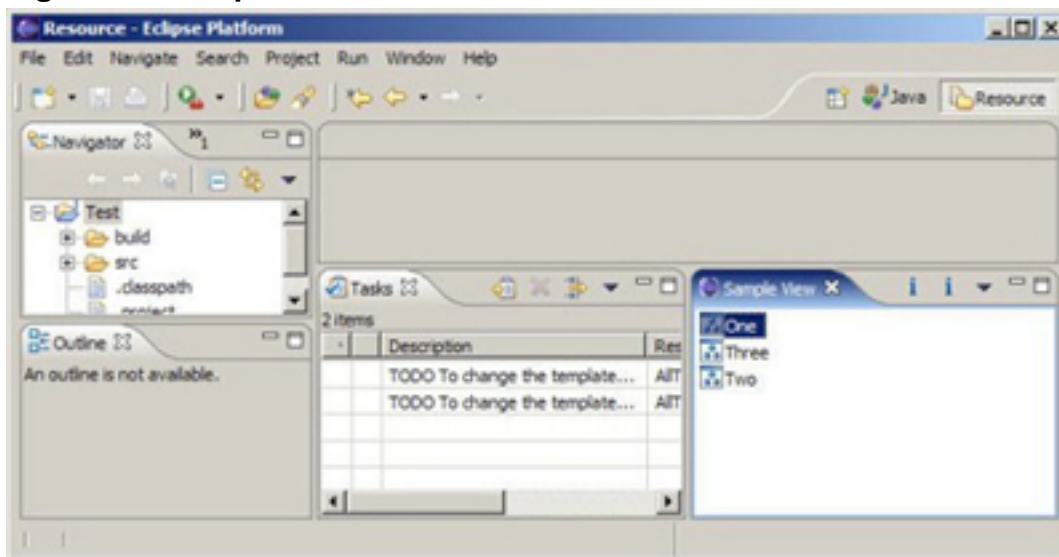
Section 6. View template

Default View template behavior

In this section, we will review the default View Template. By default, the View

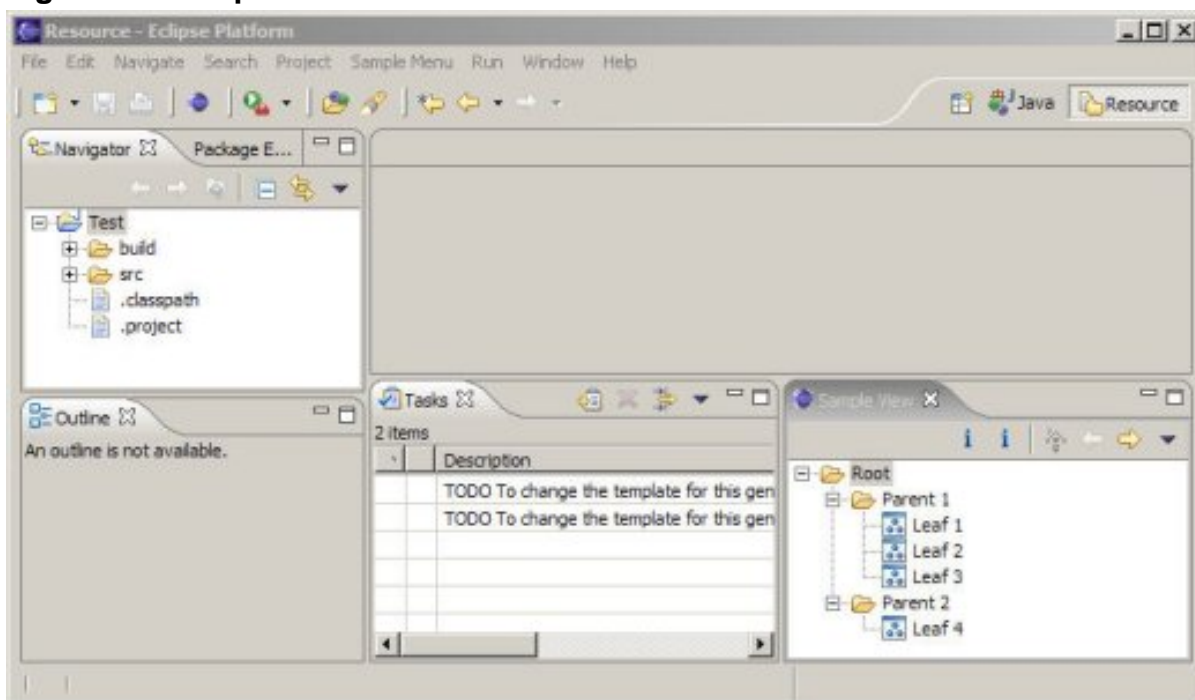
template creates a view, a dockable window inside Eclipse that contains a table and becomes apart of the Resource Perspective (shown in Figure 12).

Figure 12. Sample Table viewer



Optionally, the view can be generated to contain a tree, rather than a table (see Figure 13).

Figure 13. Sample Tree view



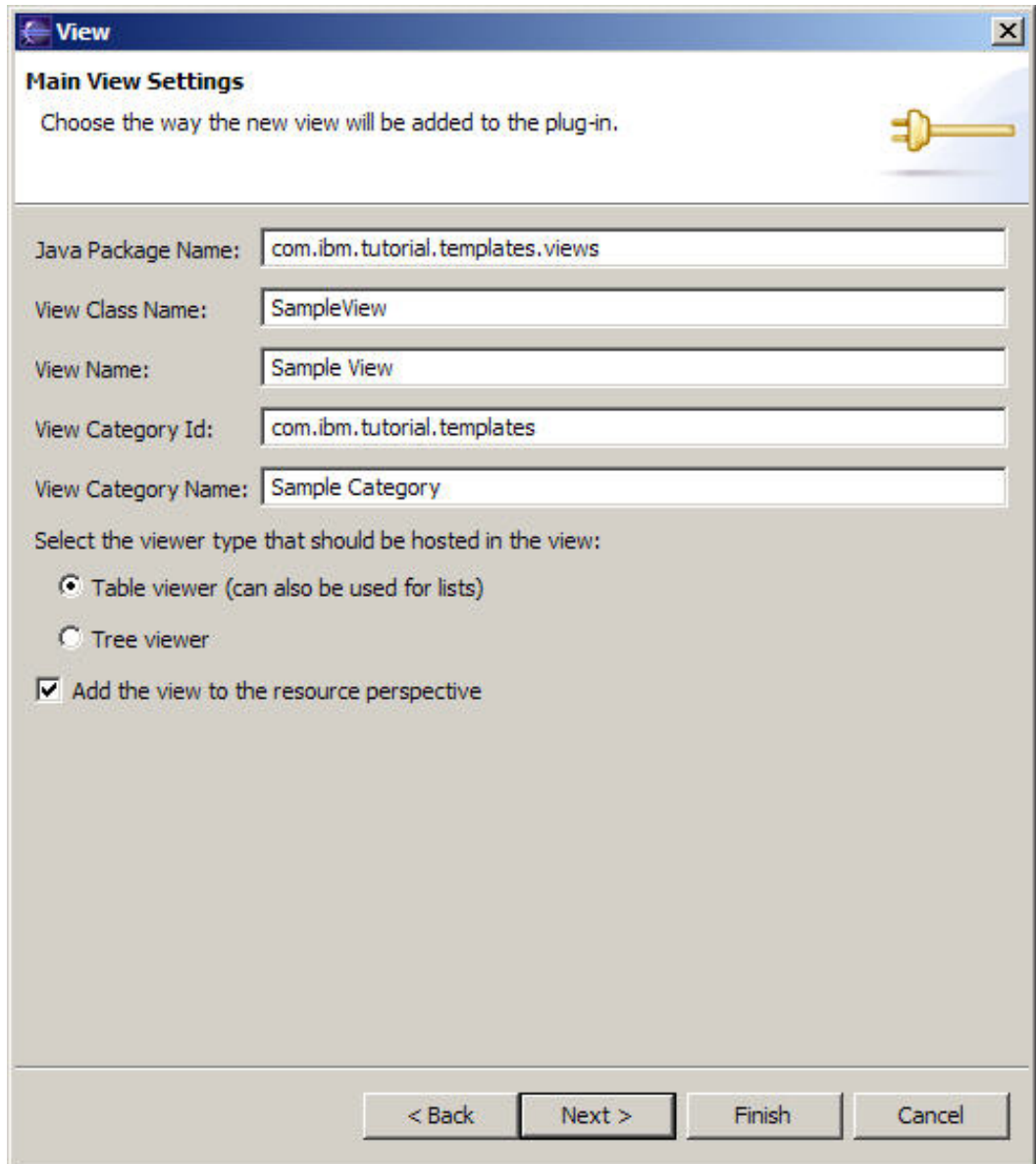
The View template wizard also provides a list of optional features that can be enabled and disabled, including double-click support, pop-up menus, toolbar menus, pull-down menus, sorting, and drill-down.

This view is also added to the Show View list under the Sample Category. It can be added to other Perspectives by selecting **Window > Show View > Other > Sample Category > Sample View**.

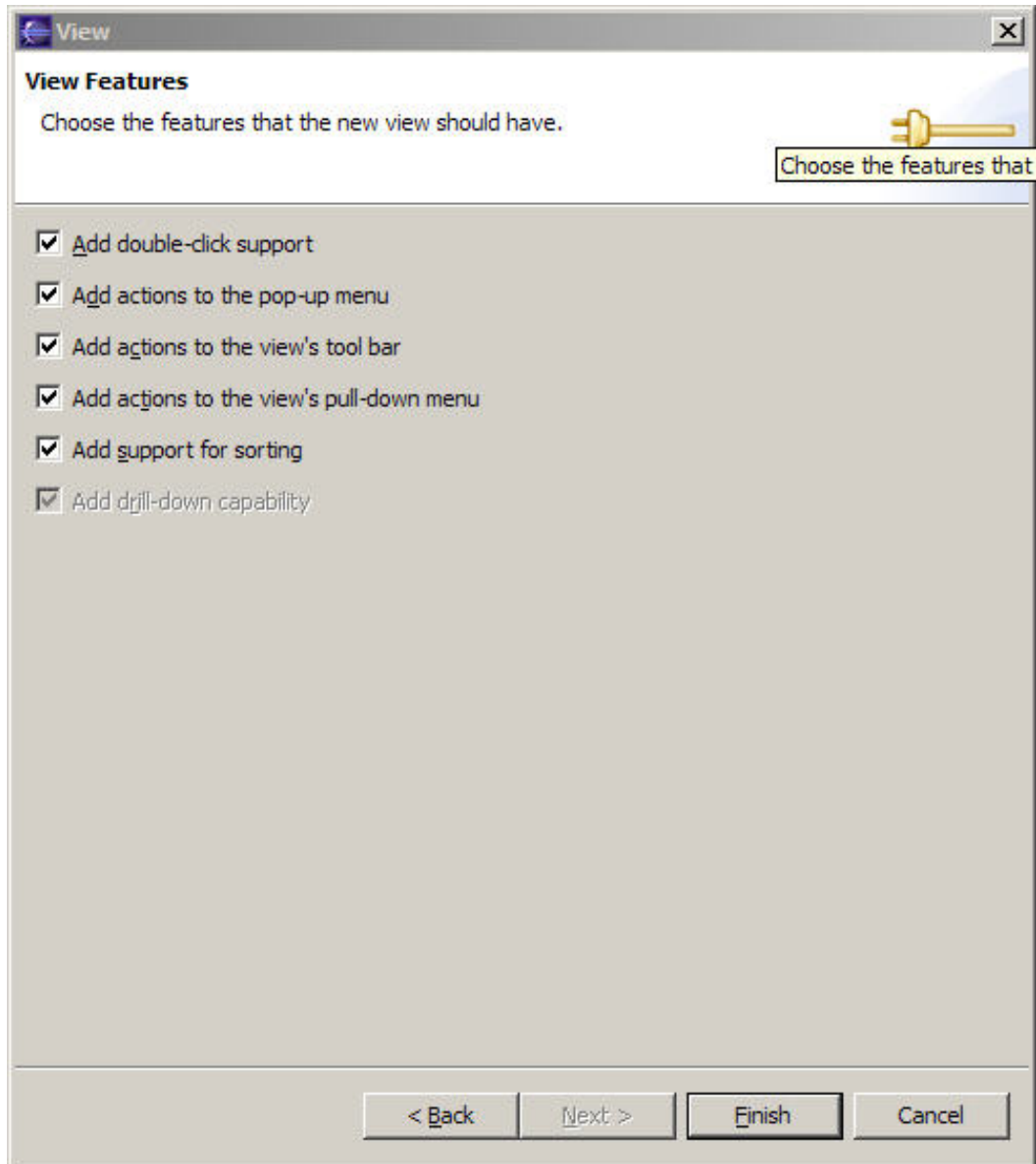
Run View template behavior

To create a default Sample View from the template in the existing project:

1. Open the Plug-in Manifest Editor by double-clicking the plugin.xml file
2. Select the Extensions tab at the bottom of the editor
3. On the Extensions tab, click **Add**
4. On the New Extension dialog, select Extension wizards
5. Select Extension Templates
6. Select the Sample View template and click **Next**
7. Select **Next** to accept the defaults shown in Figure 14
Figure 14. Main View setting defaults



8. Click **Finish** to accept the defaults shown in Figure 15
Figure 15. Main View setting defaults



Review View template behavior results

After running the View template, a new `SampleView.java` is added to the project, and two new extension points are added to `plugin.xml`.

The code for the `SampleView.java`, with some details removed for brevity:

```
package com.ibm.tutorial.templates.views;  
  
import org.eclipse.swt.widgets.Composite;  
import org.eclipse.ui.part.*;
```

```
import org.eclipse.jface.viewers.*;
import org.eclipse.swt.graphics.Image;
import org.eclipse.jface.action.*;
import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.ui.*;
import org.eclipse.swt.widgets.Menu;
import org.eclipse.swt.SWT;

public class SampleView extends ViewPart {

    class ViewContentProvider implements
        IStructuredContentProvider
    {
        public void inputChanged(
            Viewer v, Object oldInput, Object newInput) {}

        public void dispose() {}

        public Object[] getElements(Object parent) {
            return new String[] { "One", "Two", "Three" };
        }
    }

    class ViewLabelProvider extends LabelProvider
        implements ITableLabelProvider
    {
        public String getColumnText(Object obj, int index) {
            return getText(obj);
        }

        public Image getColumnImage(Object obj, int index) {
            return getImage(obj);
        }

        public Image getImage(Object obj) {
            return PlatformUI.getWorkbench().
                getSharedImages().getImage(
                    ISharedImages.IMG_OBJ_ELEMENT);
        }
    }

    class NameSorter extends ViewerSorter {}

    /**
     * The constructor.
     */
    public SampleView() {}

    /**
     * This is a callback that will allow us
     * to create the viewer and initialize it.
     */
    public void createPartControl(Composite parent) {
        viewer = new TableViewer(
            parent, SWT.MULTI | SWT.H_SCROLL | SWT.V_SCROLL);
        viewer.setContentProvider(new ViewContentProvider());
        viewer.setLabelProvider(new ViewLabelProvider());
        viewer.setSorter(new NameSorter());
        viewer.setInput(getViewSite());
        makeActions();
        hookContextMenu();
        hookDoubleClickAction();
        contributeToActionBars();
    }

    private void hookContextMenu() {
        // details removed
    }
}
```

```

    }

    private void contributeToActionBars() {
        // details removed
    }

    private void makeActions() {
        action1 = new Action() {
            public void run() {
                showMessage("Action 1 executed");
            }
        };

        action1.setText("Action 1");
        action1.setToolTipText("Action 1 tooltip");
        action1.setImageDescriptor(
            PlatformUI.getWorkbench().getSharedImages().
                getImageDescriptor(
                    ISharedImages.IMG_OBJS_INFO_TSK));

        // details removed
    }

    /**
     * Passing the focus request to the viewer's control.
     */
    public void setFocus() {
        viewer.getControl().setFocus();
    }
}

```

In this code, some details have been removed for brevity, but most of the major concepts still remain. First, notice `SampleView` extends `org.eclipse.ui.part.ViewPart`. This is the super class of all views. Next, notice this class has three inner classes: `ViewContentProvider`, `ViewLabelProvider`, and `NameSorter`. The combination of `ViewContentProvider` and `ViewLabelProvider` is similar to the Table Model concept in Swing. `ViewContentProvider` contains all the data. If you want your table to contain data other than One, Two, and Three, you should change the implementation of this class. The `ViewLabelProvider` determines how the data from `ViewContentProvider` is displayed. It can include text and images. In this case, it just uses the `toString` of the objects from the `ViewContentProvider` and displays an image. The `NameSorter` extends `ViewerSorter`, which implements a case-insensitive text comparison to determine order. This is the reason the order of the numbers are One, Three, and Two.

You will probably want to change the behavior of the generated actions that correspond to the pop-up menu, toolbar, and pull-down menu actions. These actions are created in the `makeActions` method. To change the actions' behavior, change their respective `run` method. Also feel free to change the text, tool tip, and other characteristics of the action.

The code for the View and Perspective Extension extension points added to `plugin.xml`:

```
<extension point="org.eclipse.ui.views">
  <category
    name="Sample Category"
    id="com.ibm.tutorial.templates" />
  <view
    class="com.ibm.tutorial.templates.views.SampleView"
    icon="icons/sample.gif"
    category="com.ibm.tutorial.templates"
    name="Sample View"
    id="com.ibm.tutorial.templates.views.SampleView" />
</extension>
<extension point="org.eclipse.ui.perspectiveExtensions">
  <perspectiveExtension
    targetID="org.eclipse.ui.resourcePerspective">
    <view
      ratio="0.5"
      relationship="right"
      relative="org.eclipse.ui.views.TaskList"
      id="com.ibm.tutorial.templates.views.SampleView" />
    </perspectiveExtension>
  </extension>
</extension>
```

The first extension point in the previous code adds a category called Sample Category. Next, it adds the Sample View and adds the Sample View to the Sample Category using the category attribute.

The second extension point in the previous code adds the Sample View to the Resource Perspective. Specifically, it adds it directly to the right of the TaskList and takes up 50 percent of the space, according to the ratio.

NOTE: To see a new view added to a perspective, you may have to reset the perspective by choosing **Window > Reset Perspective** from the main menu.

Section 7. Perspective Extension template

Default Perspective Extension behavior

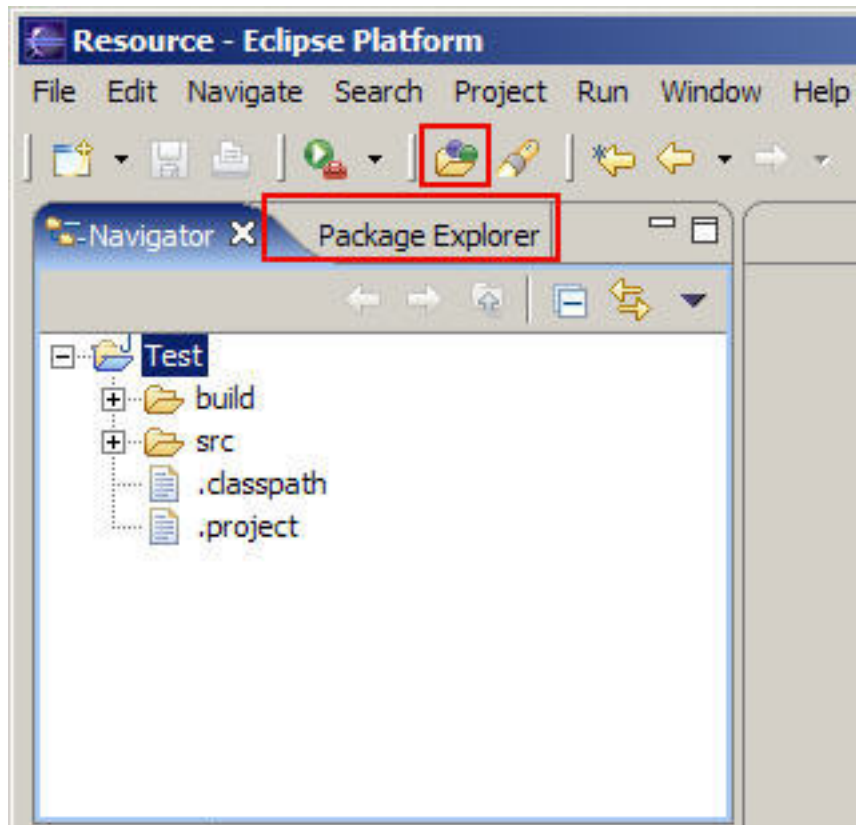
In this section, we will review the default Perspective Extension template.

By default, the Perspective Extension template modifies the configuration of the Resource perspective by adding the JavaActionSet (see figures 16 and 17), Debug Perspective shortcut, Type Hierarchy View shortcut, New Project Wizard menu, and the Package Explorer View.

Figure 16 shows that the Package Explorer is added to the Resource perspective behind the Resource Navigator. It also shows that the JavaActionSet adds the Open

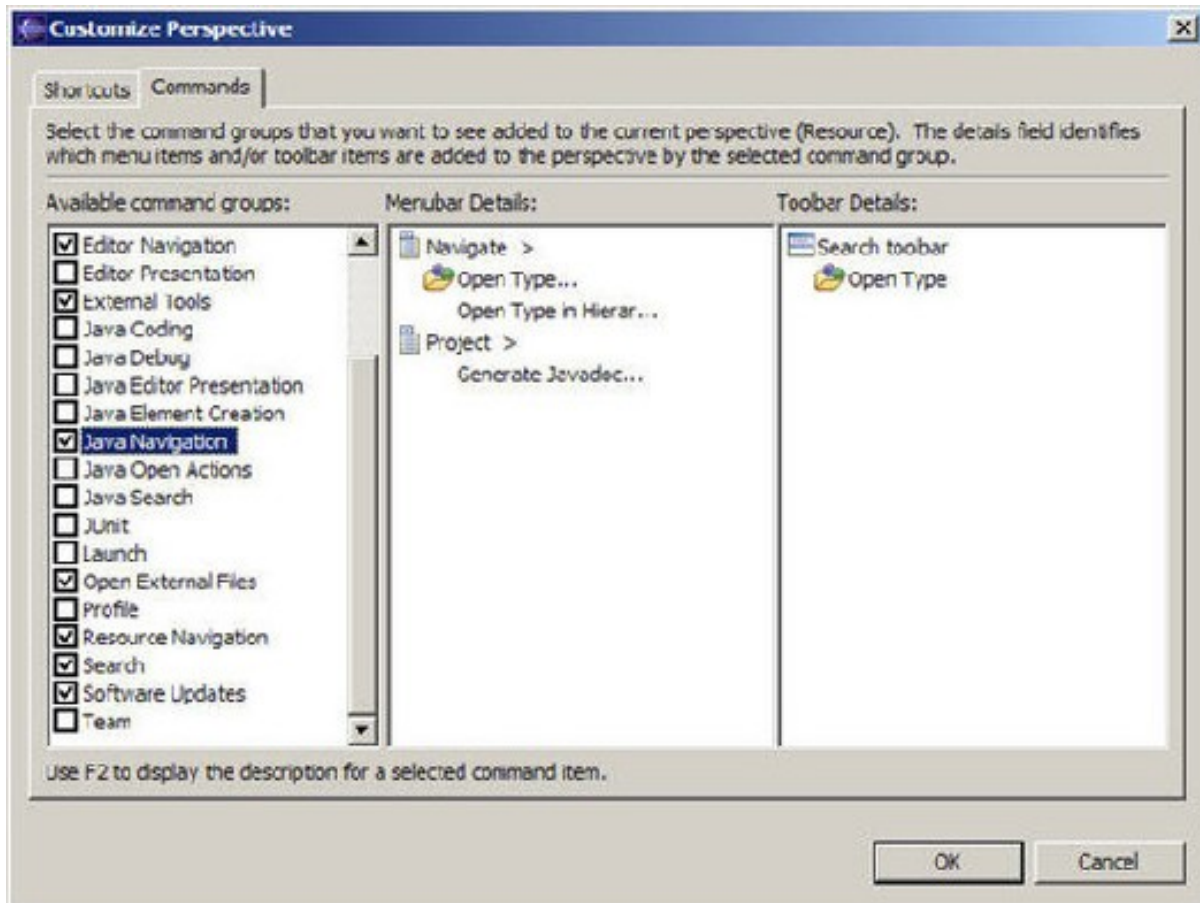
Type button to the toolbar.

Figure 16. Highlight of Package Explorer View and JavaActionSet



By default, the Resource Perspective does not have the `JavaActionSet` enabled. Figure 17 -- the Customize Perspective dialog available from **Window > Customize Perspective > Commands** -- shows the Perspective Extension with the `JavaActionSet` enabled, which has a label of Java Navigation. This means the menu bar of the Resource Perspective will have an Open Type and Open Type in Hierarchy menu items on the Navigate menu and the Generate Javadoc on the Project menu. Figure 17 also shows that the Open Type is added to the toolbar, which we saw in Figure 16.

Figure 17. JavaActionSet enabled in the Customize Perspective



Debug Perspective

Figure 18 shows the Debug Perspective added to the Perspective shortcut list. The Perspective shortcut list is under the Open a perspective button, as shown in Figure 18, and on the **Window > Open Perspective** menu.

Figure 18. Debug Perspective shortcut

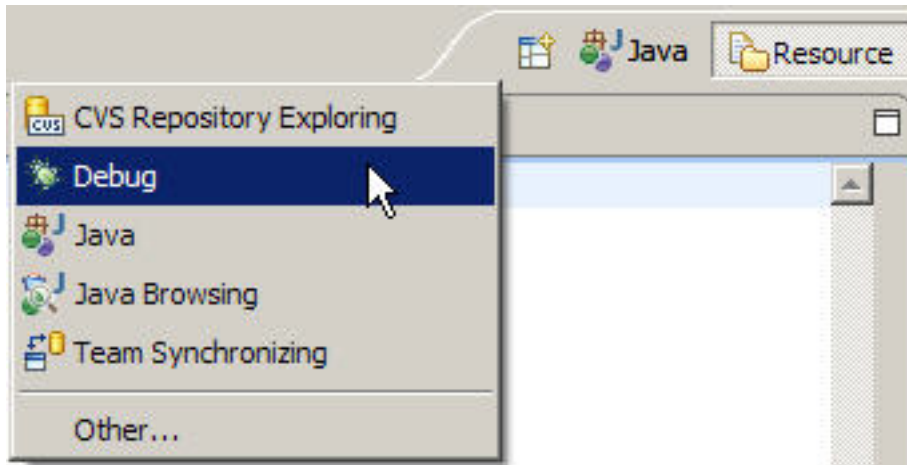
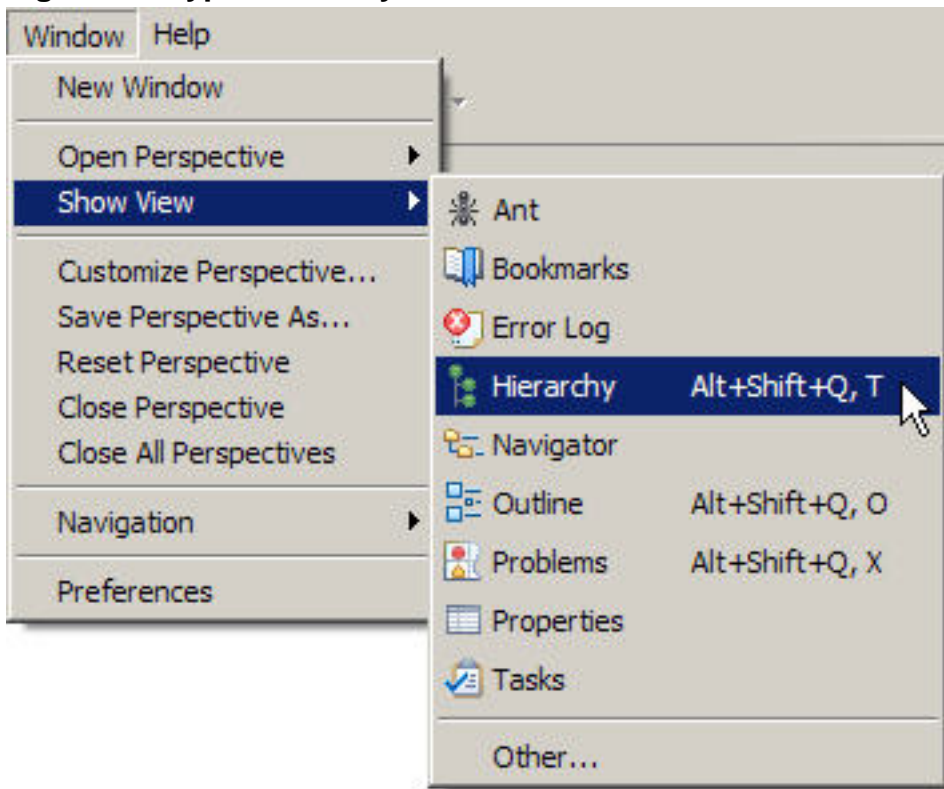


Figure 19 shows the Type Hierarchy added to the shortcut list on the **Window > Show View** menu.

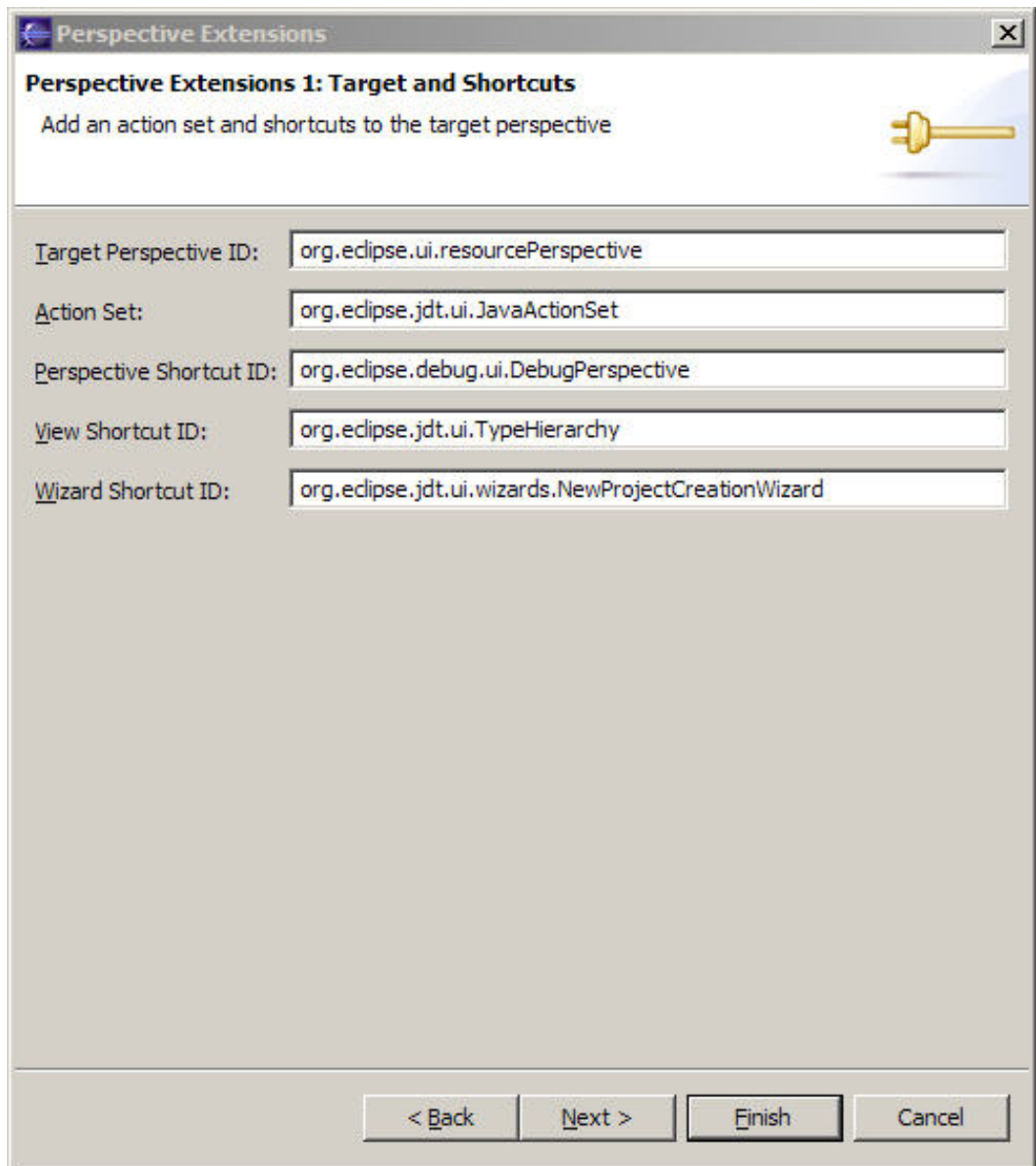
Figure 19. Type Hierarchy View shortcut



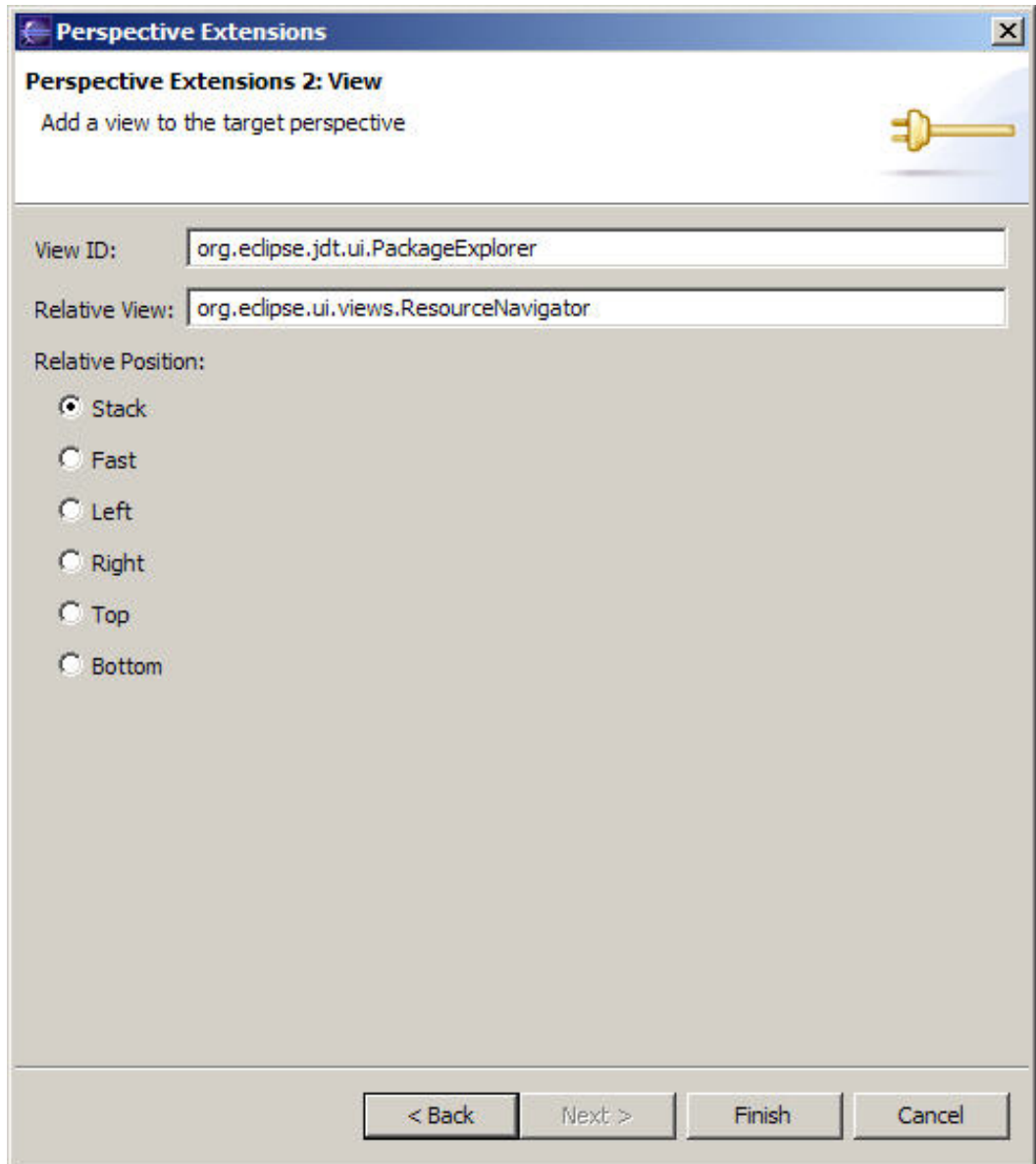
Run Perspective Extension behavior

To create an extension to the Resource Perspective with the Perspective Extension template in an existing project:

1. Open the Plug-in Manifest Editor by double-clicking the plugin.xml file
2. Select the Extensions tab at the bottom of the editor
3. On the Extensions tab, click **Add**
4. On the New Extension dialog, select **Extension Wizards**
5. Select **Extension Templates**
6. Select the Perspective Extensions template and click **Next**
7. Click **Next** to accept the defaults shown in Figure 20
Figure 20. Perspective Extensions Target and Shortcuts page



8. Click **Finish** to accept the defaults shown in Figure 21
Figure 21. Perspective Extensions View page



Review Perspective Extension behavior results

The Perspective Extension template is interesting because it does not generate any code. Instead, it adds configurations to the plugin.xml file, as shown below:

```
<extension
  point="org.eclipse.ui.perspectiveExtensions">
  <perspectiveExtension
    targetID="org.eclipse.ui.resourcePerspective">
    <view
      ratio="0.5"
      relationship="right"
```

```
        relative="org.eclipse.ui.views.TaskList"
        id="com.ibm.tutorial.templates.views.SampleView" />
</perspectiveExtension>
<perspectiveExtension
    targetID="org.eclipse.ui.resourcePerspective">
    <newWizardShortcut
        id=
        "org.eclipse.jdt.ui.wizards.NewProjectCreationWizard" />
    <viewShortcut id="org.eclipse.jdt.ui.TypeHierarchy" />
    <perspectiveShortcut
        id="org.eclipse.debug.ui.DebugPerspective" />
    <actionSet id="org.eclipse.jdt.ui.JavaActionSet" />
    <view
        relationship="stack"
        relative="org.eclipse.ui.views.ResourceNavigator"
        id="org.eclipse.jdt.ui.PackageExplorer" />
</perspectiveExtension>
```

The first `perspectiveExtension` was added by the View template. The second `perspectiveExtension` was added by the Perspective Extension.

The Perspective Extension template configuration adds the New Project Creation Wizard as a New Wizard shortcut. Then it adds the Type Hierarchy view to the View shortcuts, followed by adding the Debug Perspective to the Perspective shortcut. Next, the `JavaActionSet` is enabled with an Action Set. Last, the Package Explorer view is stacked under the Resource Navigator view.

NOTE: You may have to reset the perspective by choosing **Window > Reset Perspective** from the main menu to see changes to the perspective.

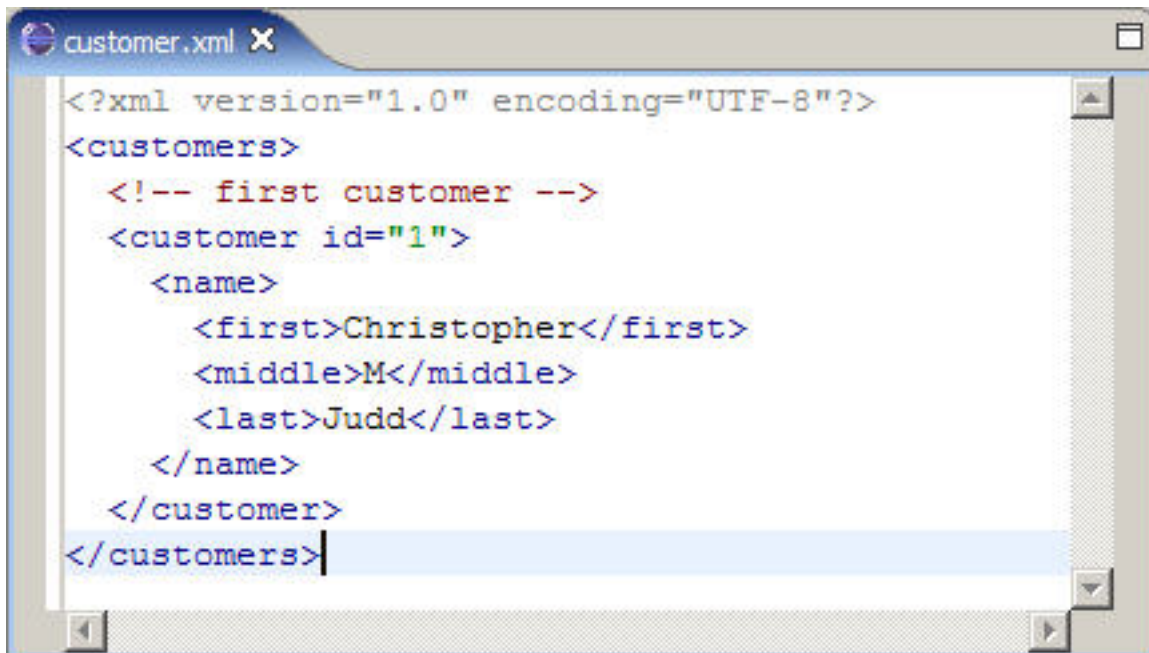
Section 8. Editor template

Default Editor template behavior

By default, the Editor template generates a fully functional XML Editor (see Figure 22) and associates it with the `*.xml` file type (see Figure 23).

Figure 22 shows an example of the XML Editor at work on a `customer.xml` file. Notice the editor has syntax highlighting for processing instructions, comments, elements, and attributes.

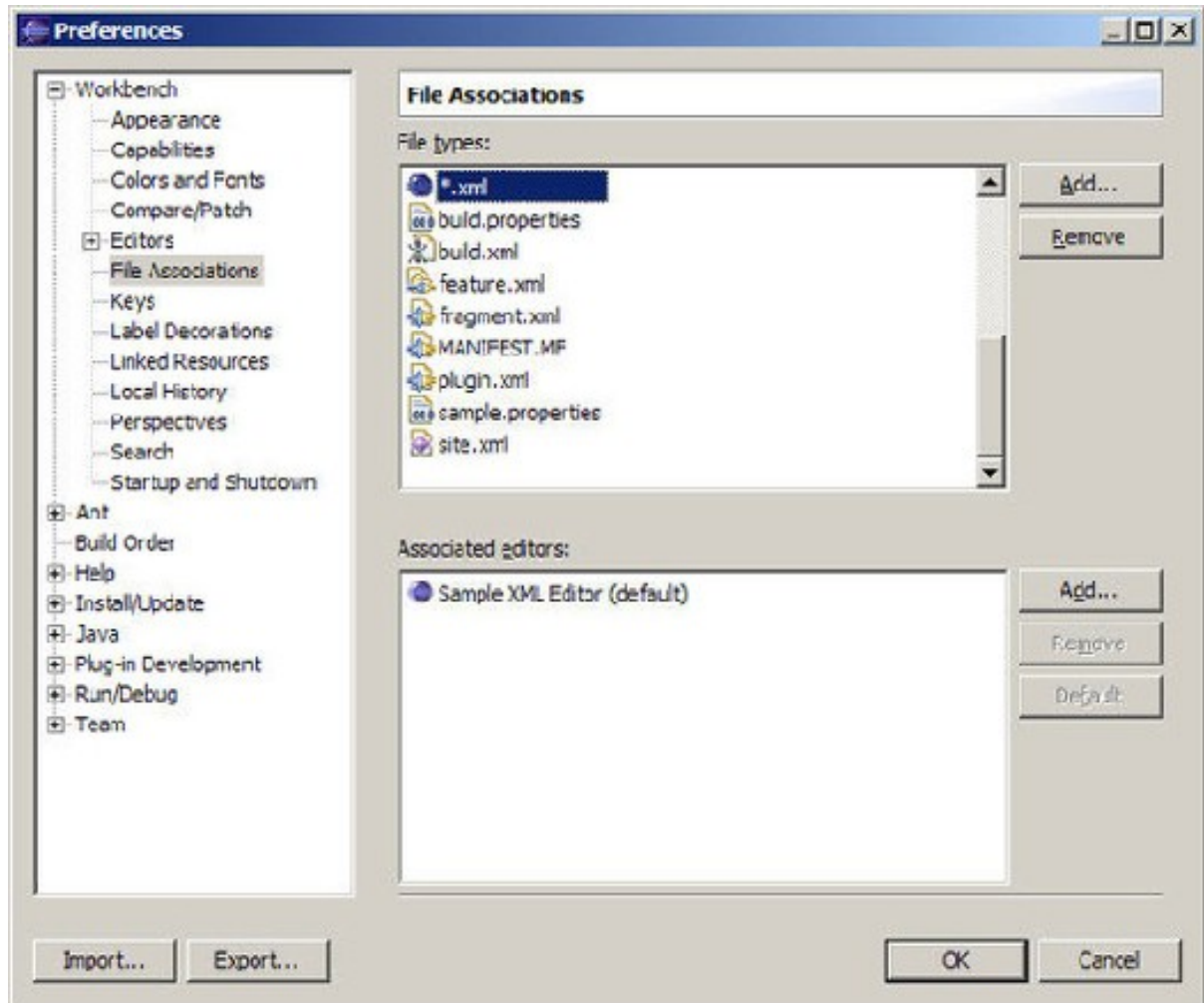
Figure 22. XML Editor generated by Editor template

A screenshot of an Eclipse IDE window titled 'customer.xml'. The window contains XML code for a customer record. The code is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<customers>
  <!-- first customer -->
  <customer id="1">
    <name>
      <first>Christopher</first>
      <middle>M</middle>
      <last>Judd</last>
    </name>
  </customer>
</customers>
```

Figure 23 shows that the *.xml file type is associated with the Sample XML Editor in the Workbench properties found under the **Windows > Preferences** menu.

Figure 23. XML Editor file association

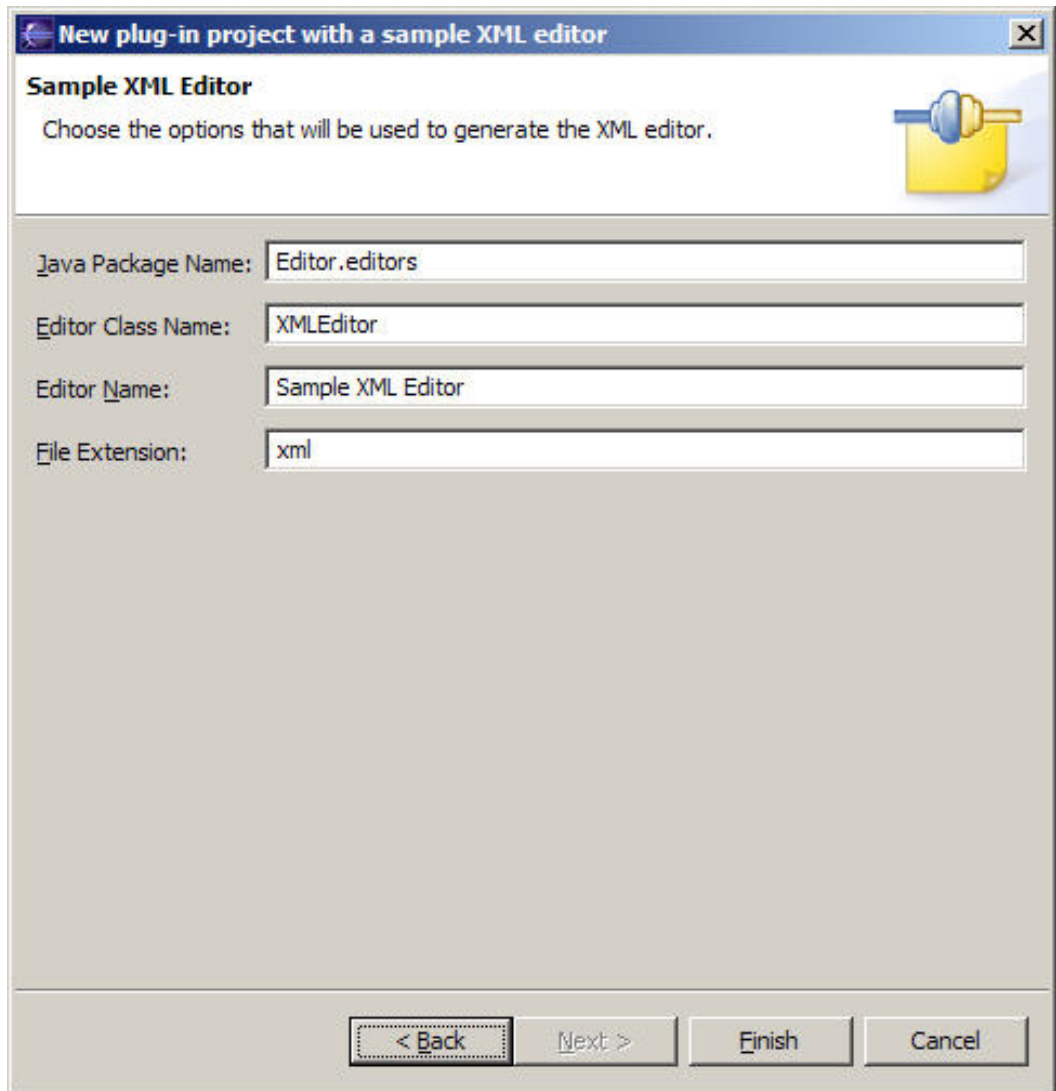


Run Editor template behavior

To create the XML Editor with the Editor template in an existing project:

1. Open the Plug-in Manifest Editor by double-clicking the plugin.xml file
2. Select the **Extensions** tab at the bottom of the editor
3. On the Extensions tab, click **Add**
4. On the New Extension dialog, select **Extension Wizards**
5. Select **Extension Templates**
6. Select the Editor template and click **Next**

7. Click **Next** to accept the defaults shown in Figure 24
Figure 24. Sample XML Editor defaults



New plug-in project with a sample XML editor

Sample XML Editor

Choose the options that will be used to generate the XML editor.

Java Package Name:

Editor Class Name:

Editor Name:

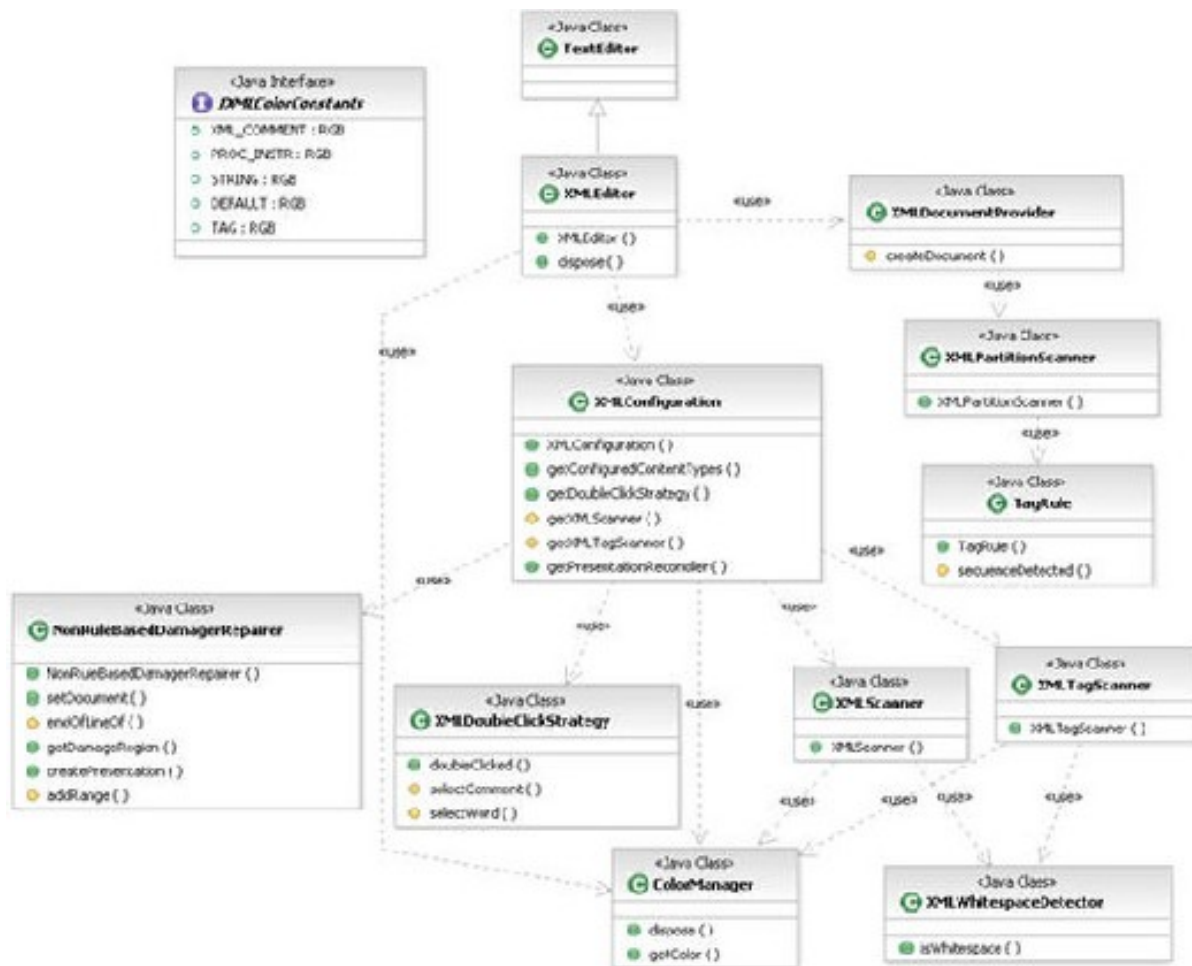
File Extension:

< Back Next > Finish Cancel

Review Editor template behavior results

The Editor template is the most complex of the plug-in templates since it generates the most code (12 Java files in all) and most working parts. Rather than explain each line of code, we will use the UML diagram in Figure 25 to explain the classes. As always, we will review the plugin.xml changes in the code below.

Figure 25. XML Editor UML



The XMLEditor is the primary class, even though it does not do much except associate itself with the ColorManager, XMLDocumentProvider, and XMLConfiguration, which are used for mapping and caching colors, acting as a controller, and configuring the XML parsing, respectively. The rest of the classes -- XMLDoubleClickStrategy, XMLScanner, XMLTagScanner, XMLWhitespaceDetector, XMLPartitionScanner, and TagRule -- are used to support the XML parsing.

The lone interface, IXMLColorConstants, contains a list of RGB color constant mappings to XML types:

```

<extension point="org.eclipse.ui.editors">
  <editor
    class="com.ibm.tutorial.templates.editors.XMLEditor"
    icon="icons/sample.gif"
    contributorClass=
"org.eclipse.ui.texteditor.BasicTextEditorActionContributor"
    name="Sample XML Editor"
    id="com.ibm.tutorial.templates.editors.XMLEditor"
    extensions="xml" />
</extension>

```

This code shows the extension point that adds the XML Editor. The class attribute uses the generated `XMLEditor` class. The icon is the default Eclipse icon. Figure 23 shows the icon being used in the file association. The icon is also used in the Open With context menu on XML files. The `contributorClass` attribute tells Eclipse to include basic editor menus. The name identifies the editor name. You can see how the editor name is used in the file association preferences page in Figure 23. Last, the editor is associated with `*.xml` files with the `extensions` attribute.

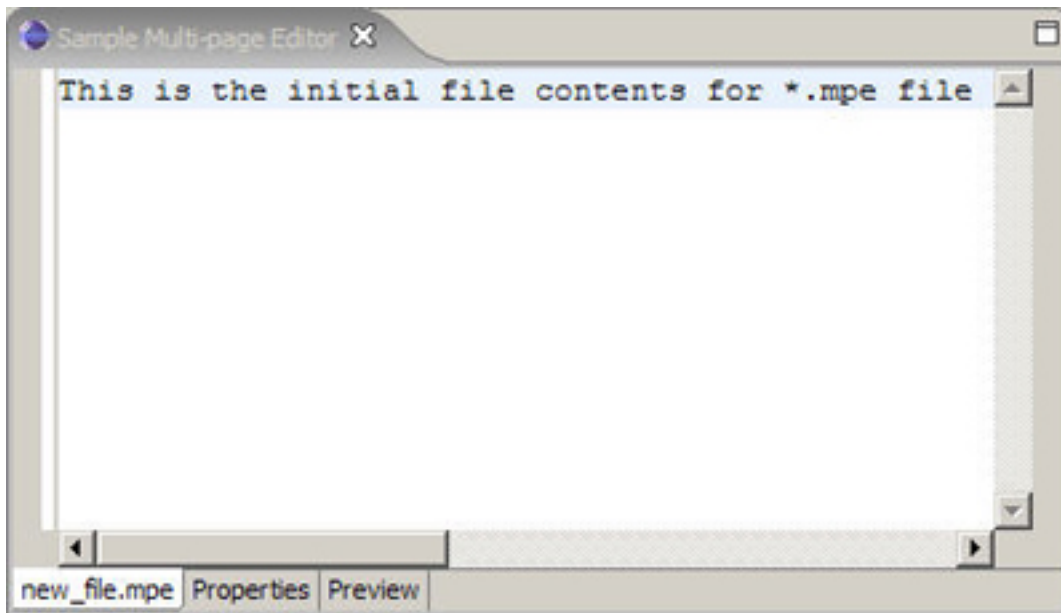
Section 9. Multi-page Editor template

Default Multi-page Editor template behavior

By default, the Multi-page Editor template generates a trivial editor with three pages and a wizard for generating files associated with the editor. The editor enables text to be typed in a text editor on the first page (see Figure 26), previewed on the third in a read-only format with each word on a separate line and ordered alphabetically (see Figure 27), and the ability to control the font of the third page from the second (see Figure 28).

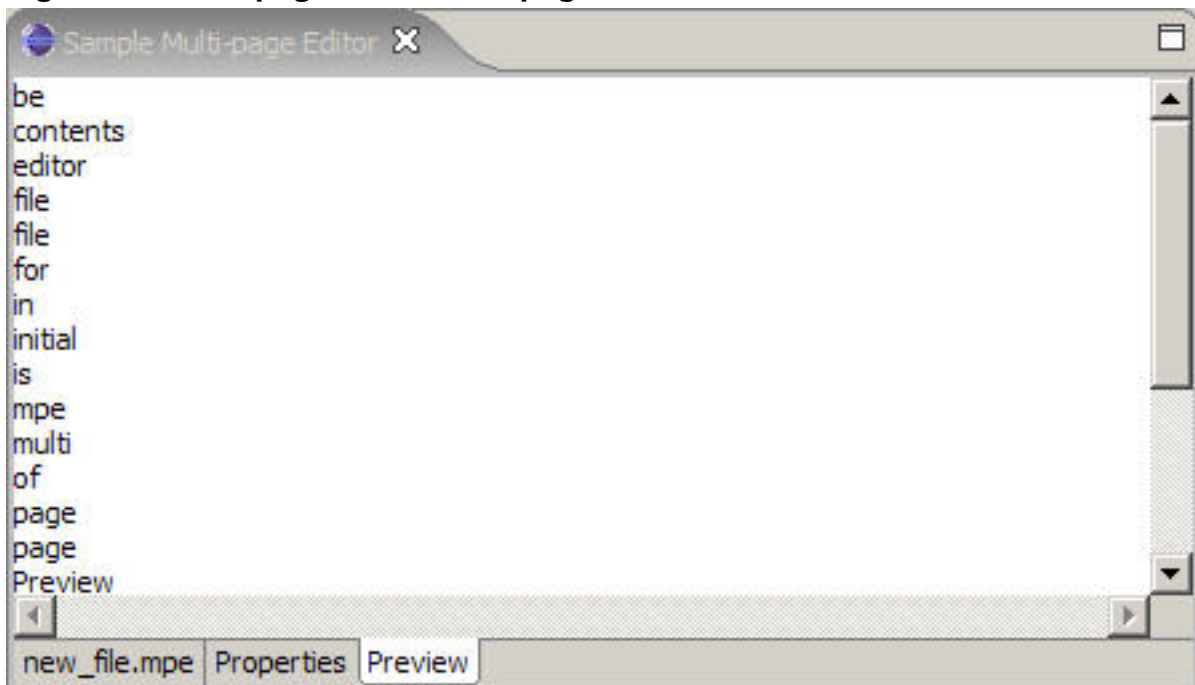
The first page of the Multi-page Editor shown in Figure 26 is a simple text editor. You can use it to type anything you want. The text can then be previewed on the third, or Preview, page shown in Figure 27.

Figure 26. First page of the Multi-page Editor



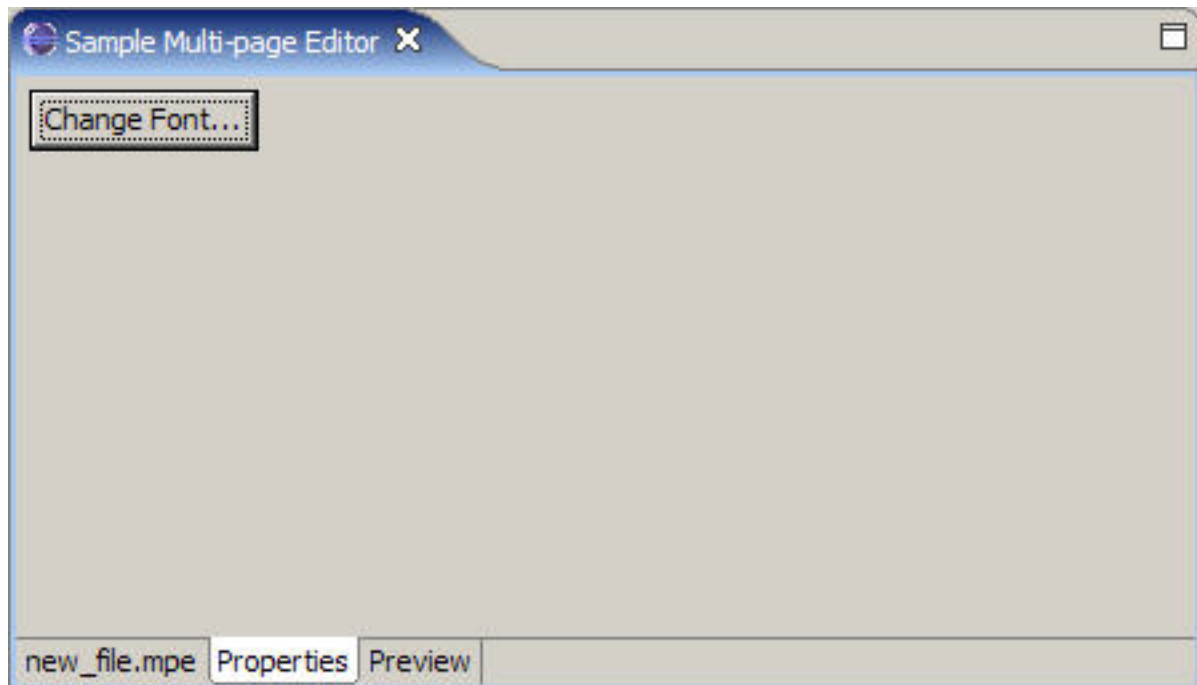
Notice in Figure 27, the sentence from Figure 26 has been reordered so the sentence is now in alphabetical order. In addition, each word is separated on its own line.

Figure 27. Third page of the Multi-page Editor



The second, or Properties, page shown in Figure 28 only has as a single property to set -- the font of the third or Preview page.

Figure 28. Second page of the Multi-page Editor



Default Multi-page Editor template behavior, continued

This Multi-page Editor is associated with a file extension of `*.mpe`, which stands for Multi-page Editor. To create a new file of this type, the Multi-page Editor generates the Multi-page Editor file wizard, shown in figures 29 and 30.

Figure 29 shows the new Multi-page Editor file wizard located in the Sample Wizards category.

Figure 29. Multi-page Editor File wizard

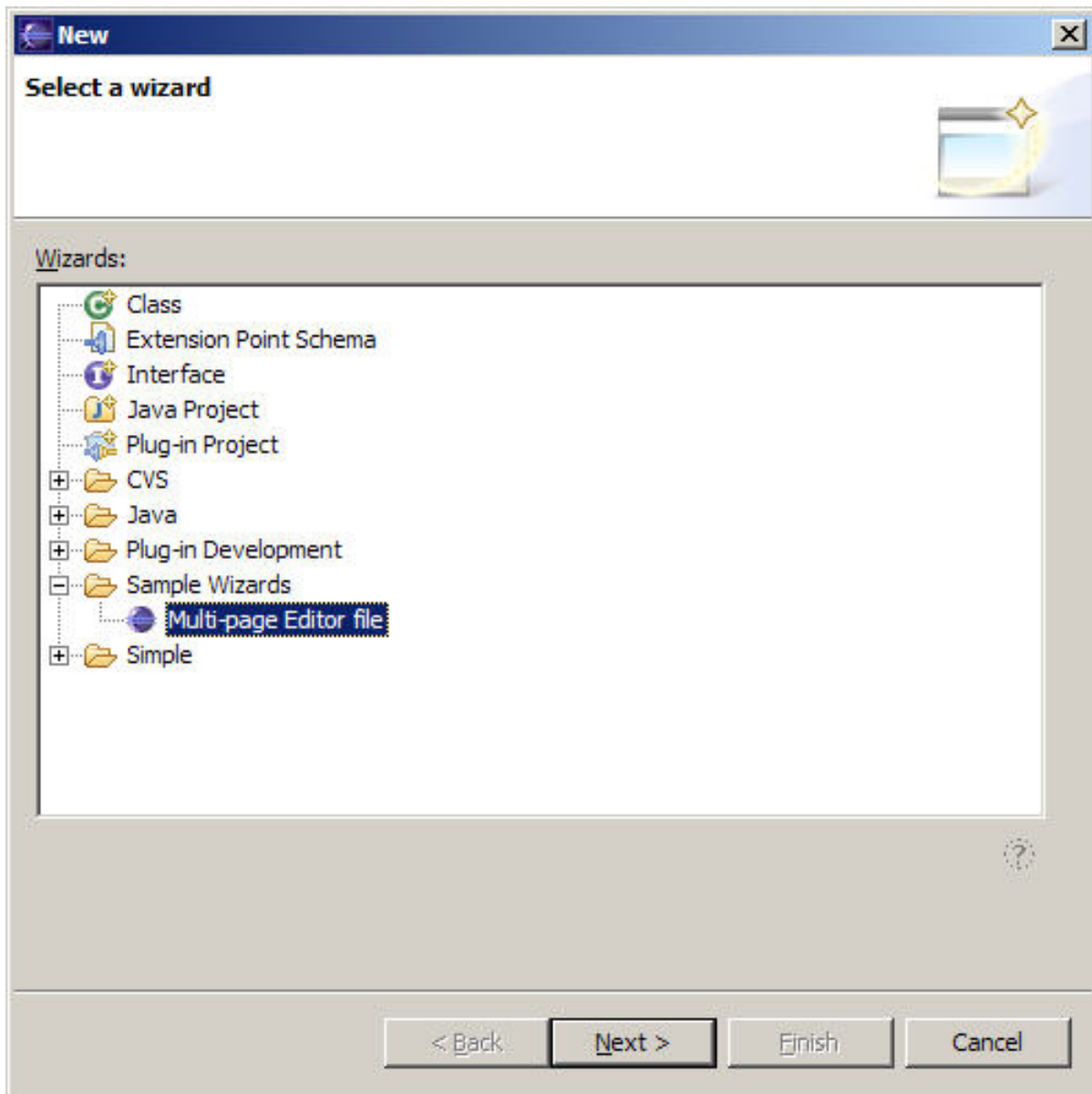
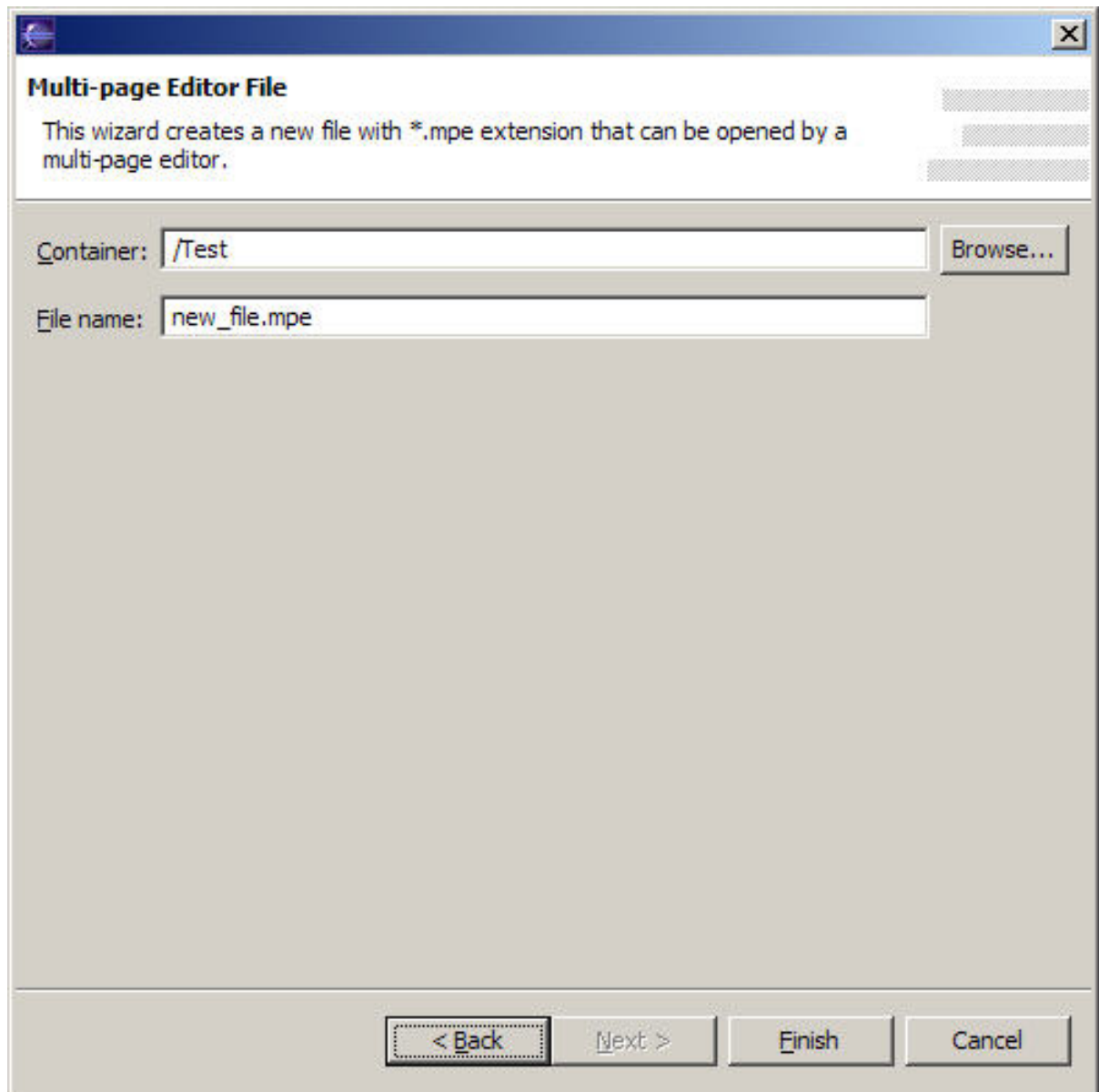


Figure 30 shows that the Multi-page editor collects two pieces of information. First is the Container, or Folder. The second is the name of the file. By default, the new file will contain the sentence in Figure 26.

Figure 30. Multi-page Editor File wizard



Run Multi-page Editor template behavior

To add the Multi-page Editor and Multi-page Editor File wizard to an existing project:

1. Open the Plug-in Manifest Editor by double-clicking the plugin.xml file
2. Select the **Extensions** tab at the bottom of the editor
3. On the Extensions tab, click **Add**

4. On the New Extension dialog, select **Extension Wizards**
5. Select **Extension Templates**
6. Select the Multi-page Editor template and click **Next**
7. Click Finish to accept the defaults shown in Figure 31
Figure 31. Sample Multi-Page Editor

Multi-page Editor

Sample Multi-Page Editor

Choose the options that will be used to generate the multi-page editor.

Java Package Name:

Editor Class Name:

Editor Contributor Class Name:

Editor Name:

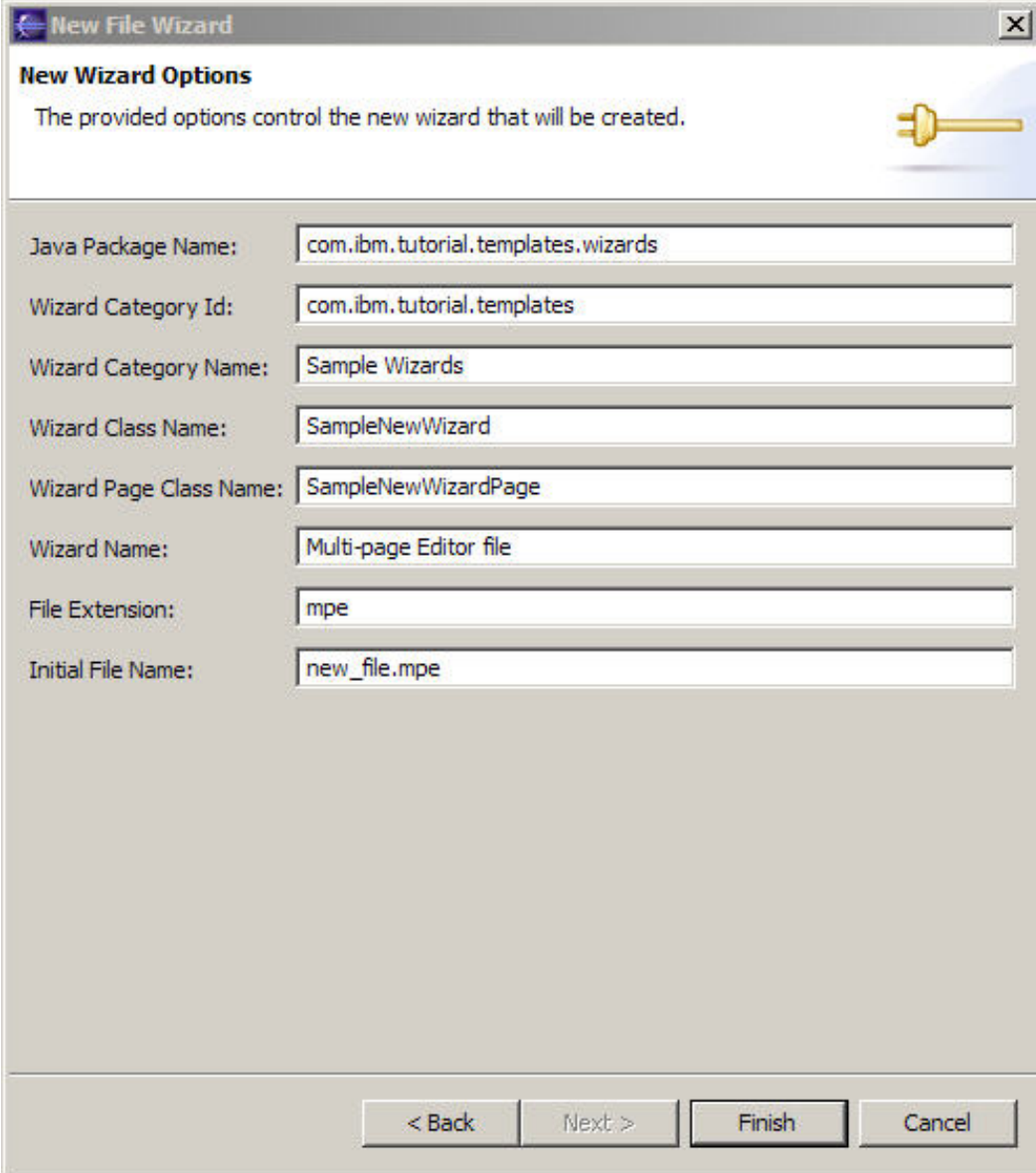
File Extension:

< Back Next > Finish Cancel

8. Click **Add** again
9. On the New Extension dialog, select **Extension Wizards**

10. Select **Extension Templates**
11. Select the New File Wizard template and click **Next**
12. Click Finish to accept the defaults shown in Figure 32

Figure 32. New wizard options



The screenshot shows the 'New File Wizard' dialog box with the 'New Wizard Options' tab selected. The dialog contains several text input fields for configuring a new wizard. The fields and their values are as follows:

Field Name	Value
Java Package Name:	com.ibm.tutorial.templates.wizards
Wizard Category Id:	com.ibm.tutorial.templates
Wizard Category Name:	Sample Wizards
Wizard Class Name:	SampleNewWizard
Wizard Page Class Name:	SampleNewWizardPage
Wizard Name:	Multi-page Editor file
File Extension:	mpe
Initial File Name:	new_file.mpe

At the bottom of the dialog, there are four buttons: '< Back', 'Next >', 'Finish', and 'Cancel'. The 'Finish' button is highlighted with a dark border.

Review Multi-page Editor template behavior results

The Multi-page Editor generates four Java files. Two are related to the editor and the other two are responsible for the wizard. The two editor classes are the

MultiPageEditor and the MultiPageEditorContributor, which are the editor itself and the class responsible for managing global actions for the editor. The two wizard classes are the SampleNewWizard and a page in that wizard, SampleNewWizardPage.

The code for the MultiPageEditor class:

```
package com.ibm.tutorial.templates.editors;

import java.io.StringWriter;
import java.text.Collator;
import java.util.ArrayList;
import java.util.Collections;
import java.util.StringTokenizer;

import org.eclipse.core.resources.IMarker;
import org.eclipse.core.resources.IResourceChangeEvent;
import org.eclipse.core.resources.IResourceChangeListener;
import org.eclipse.core.resources.ResourcesPlugin;
import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.jface.dialogs.ErrorDialog;
import org.eclipse.swt.SWT;
import org.eclipse.swt.custom.StyledText;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.graphics.Font;
import org.eclipse.swt.graphics.FontData;
import org.eclipse.swt.layout.FillLayout;
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.Display;
import org.eclipse.swt.widgets.FontDialog;
import org.eclipse.ui.*;
import org.eclipse.ui.editors.text.TextEditor;
import org.eclipse.ui.part.FileEditorInput;
import org.eclipse.ui.part.MultiPageEditorPart;
import org.eclipse.ui.ide.IDE;

/**
 * An example showing how to create a multi-page editor.
 * This example has 3 pages:
 * <ul>
 * <li>page 0 contains a nested text editor.
 * <li>page 1 allows you to change the font used in page 2
 * <li>page 2 shows the words in page 0 in sorted order
 * </ul>
 */
public class MultiPageEditor
    extends MultiPageEditorPart
    implements IResourceChangeListener
{
    /** The text editor used in page 0. */
    private TextEditor editor;

    /** The font chosen in page 1. */
    private Font font;

    /** The text widget used in page 2. */
    private StyledText text;
}
```

```

    * Creates a multi-page editor example.
    */
public MultiPageEditor() {
    super();
    ResourcesPlugin.getWorkspace().
        addResourceChangeListener(this);
}

/**
 * Creates page 0 of the multi-page editor,
 * which contains a text editor.
 */
void createPage0() {
    try {
        editor = new TextEditor();
        int index = addPage(editor, getEditorInput());
        setPageText(index, editor.getTitle());
    } catch (PartInitException e) {
        ErrorDialog.openError(
            getSite().getShell(),
            "Error creating nested text editor",
            null,
            e.getStatus());
    }
}

/**
 * Creates page 1 of the multi-page editor,
 * which allows you to change the font used in page 2.
 */
void createPage1() {

    Composite composite =
        new Composite(getContainer(), SWT.NONE);
    GridLayout layout = new GridLayout();
    composite.setLayout(layout);
    layout.numColumns = 2;

    Button fontButton = new Button(composite, SWT.NONE);
    GridData gd = new GridData(GridData.BEGINNING);
    gd.horizontalSpan = 2;
    fontButton.setLayoutData(gd);
    fontButton.setText("Change Font...");

    fontButton.addSelectionListener(new SelectionAdapter() {
        public void widgetSelected(SelectionEvent event) {
            setFont();
        }
    });

    int index = addPage(composite);
    setPageText(index, "Properties");
}

/**
 * Creates page 2 of the multi-page editor,
 * which shows the sorted text.
 */
void createPage2() {
    Composite composite =
        new Composite(getContainer(), SWT.NONE);
    FillLayout layout = new FillLayout();
    composite.setLayout(layout);
    text = new StyledText(
        composite, SWT.H_SCROLL | SWT.V_SCROLL);
    text.setEditable(false);

    int index = addPage(composite);
    setPageText(index, "Preview");
}

```

```

}

/**
 * Creates the pages of the multi-page editor.
 */
protected void createPages() {
    createPage0();
    createPage1();
    createPage2();
}

/**
 * The <code>MultiPageEditorPart</code> implementation of
 * this <code>IWorkbenchPart</code> method disposes all
 * nested editors.
 * Subclasses may extend.
 */
public void dispose() {
    ResourcesPlugin.getWorkspace().
        removeResourceChangeListener(this);
    super.dispose();
}

/**
 * Saves the multi-page editor's document.
 */
public void doSave(IProgressMonitor monitor) {
    getEditor(0).doSave(monitor);
}

/**
 * Saves the multi-page editor's document as another
 * file.
 * Also updates the text for page 0's tab, and updates
 * this multi-page editor's input
 * to correspond to the nested editor's.
 */
public void doSaveAs() {
    IEditorPart editor = getEditor(0);
    editor.doSaveAs();
    setPageText(0, editor.getTitle());
    setInput(editor.getEditorInput());
}

/* (non-Javadoc)
 * Method declared on IEditorPart
 */
public void gotoMarker(IMarker marker) {
    setActivePage(0);
    IDE.gotoMarker(getEditor(0), marker);
}

/**
 * The <code>MultiPageEditorExample</code> implementation
 * of this method
 * checks that the input is an instance of
 * <code>IFileEditorInput</code>.
 */
public void init(
    IEditorSite site, IEditorInput editorInput)
    throws PartInitException
{
    if (!(editorInput instanceof IFileEditorInput))
        throw new PartInitException(
            "Invalid Input: Must be IFileEditorInput");
    super.init(site, editorInput);
}

/* (non-Javadoc)

```

```

    * Method declared on IEditorPart.
    */
public boolean isSaveAsAllowed() {
    return true;
}

/**
 * Calculates the contents of page 2 when the it is
 * activated.
 */
protected void pageChange(int newPageIndex) {
    super.pageChange(newPageIndex);
    if (newPageIndex == 2) {
        sortWords();
    }
}

/**
 * Closes all project files on project close.
 */
public void resourceChanged(
    final IResourceChangeEvent event)
{
    if(event.getType() == IResourceChangeEvent.PRE_CLOSE){
        Display.getDefault().asyncExec(new Runnable(){
            public void run(){
                IWorkbenchPage[] pages =
                    getSite().getWorkbenchWindow().getPages();
                for (int i = 0; i<pages.length; i++){
                    if(((FileEditorInput)editor.getEditorInput())
                        .getFile().getProject().equals(
                            event.getResource()))
                    {
                        IEditorPart editorPart =
                            pages[i].findEditor(
                                editor.getEditorInput());
                        pages[i].closeEditor(editorPart,true);
                    }
                }
            }
        });
    }
}

/**
 * Sets the font related data to be applied to the text
 * in page 2.
 */
void setFont() {
    FontDialog fontDialog =
        new FontDialog(getSite().getShell());

    fontDialog.setFontList(text.getFont().getFontData());
    FontData fontData = fontDialog.open();
    if (fontData != null) {
        if (font != null)
            font.dispose();
        font = new Font(text.getDisplay(), fontData);
        text.setFont(font);
    }
}

/**
 * Sorts the words in page 0, and shows them in page 2.
 */
void sortWords() {
    String editorText =
        editor.getDocumentProvider().getDocument(
            editor.getEditorInput()).get();
}

```

```

StringTokenizer tokenizer =
    new StringTokenizer(editorText,
        "\t\n\r\f!@#\u0024%&*()-" +
        "_=+`~[]{};:\'\" ,.<>/?|\\");
ArrayList editorWords = new ArrayList();
while (tokenizer.hasMoreTokens()) {
    editorWords.add(tokenizer.nextToken());
}

Collections.sort(editorWords, Collator.getInstance());
StringWriter displayText = new StringWriter();
for (int i = 0; i < editorWords.size(); i++) {
    displayText.write(((String) editorWords.get(i)));
    displayText.write(
        System.getProperty("line.separator"));
}
text.setText(displayText.toString());
}
}

```

The code for the `MultiPageEditorContributor.java` class:

```

package com.ibm.tutorial.templates.editors;

import org.eclipse.jface.action.*;
import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.ui.IActionBars;
import org.eclipse.ui.IEditorPart;
import org.eclipse.ui.IWorkbenchActionConstants;
import org.eclipse.ui.PlatformUI;
import org.eclipse.ui.actions.ActionFactory;
import org.eclipse.ui.ide.IDE;
import org.eclipse.ui.ide.IDEActionFactory;
import org.eclipse.ui.part.MultiPageEditorActionBarContributor;
import org.eclipse.ui.texteditor.ITextEditor;
import org.eclipse.ui.texteditor.ITextEditorActionConstants;

/**
 * Manages the installation/deinstallation of global
 * actions for multi-page editors.
 * Responsible for the redirection of global actions to
 * the active editor.
 * Multi-page contributor replaces the contributors for the
 * individual editors in the multi-page editor.
 */
public class MultiPageEditorContributor
    extends MultiPageEditorActionBarContributor
{
    private IEditorPart activeEditorPart;
    private Action sampleAction;
    /**
     * Creates a multi-page contributor.
     */
    public MultiPageEditorContributor() {
        super();
        createActions();
    }

    /**
     * Returns the action registered with the given
     * text editor.
     * @return IAction or null if editor is null.
     */
    protected IAction getAction(
        ITextEditor editor, String actionID)

```

```

    {
        return (editor == null ?
            null : editor.getAction(actionID));
    }

    /* (non-Javadoc)
     * Method declared in
     * AbstractMultiPageEditorActionBarContributor.
     */
    public void setActivePage(IEditorPart part) {
        if (activeEditorPart == part)
            return;

        activeEditorPart = part;

        IActionBars actionBars = getActionBars();
        if (actionBars != null) {
            ITextEditor editor = (part instanceof ITextEditor) ?
                (ITextEditor) part : null;

            actionBars.setGlobalActionHandler(
                ActionFactory.DELETE.getId(),
                getAction(
                    editor, ITextEditorActionConstants.DELETE));
            actionBars.setGlobalActionHandler(
                ActionFactory.UNDO.getId(),
                getAction(
                    editor, ITextEditorActionConstants.UNDO));
            actionBars.setGlobalActionHandler(
                ActionFactory.REDO.getId(),
                getAction(
                    editor, ITextEditorActionConstants.REDO));
            actionBars.setGlobalActionHandler(
                ActionFactory.CUT.getId(),
                getAction(
                    editor, ITextEditorActionConstants.CUT));
            actionBars.setGlobalActionHandler(
                ActionFactory.COPY.getId(),
                getAction(
                    editor, ITextEditorActionConstants.COPY));
            actionBars.setGlobalActionHandler(
                ActionFactory.PASTE.getId(),
                getAction(
                    editor, ITextEditorActionConstants.PASTE));
            actionBars.setGlobalActionHandler(
                ActionFactory.SELECT_ALL.getId(),
                getAction(
                    editor,
                    ITextEditorActionConstants.SELECT_ALL));
            actionBars.setGlobalActionHandler(
                ActionFactory.FIND.getId(),
                getAction(
                    editor, ITextEditorActionConstants.FIND));
            actionBars.setGlobalActionHandler(
                IDEActionFactory.BOOKMARK.getId(),
                getAction(
                    editor, IDEActionFactory.BOOKMARK.getId()));
            actionBars.updateActionBars();
        }
    }

    private void createActions() {
        sampleAction = new Action() {
            public void run() {
                MessageDialog.openInformation(
                    null,
                    "Templates Plug-in",
                    "Sample Action Executed");
            }
        }
    }

```

```

    };
    sampleAction.setText("Sample Action");
    sampleAction.setToolTipText("Sample Action tool tip");
    sampleAction.setImageDescriptor(
        PlatformUI.getWorkbench().getSharedImages().
            getImageDescriptor(
                IDE.SharedImages.IMG_OBJS_TASK_TSK));
}

public void contributeToMenu(IMenuManager manager) {
    IMenuManager menu = new MenuManager("Editor &Menu");
    manager.prependToGroup(
        IWorkbenchActionConstants.MB_ADDITIONS, menu);
    menu.add(sampleAction);
}

public void contributeToToolBar(IToolBarManager manager)
{
    manager.add(new Separator());
    manager.add(sampleAction);
}
}

```

This code is a contributor, which demonstrates how to customize the menu and toolbar based on a particular editor in the methods `createActions`, `contributeToMenu`, and `contributeToToolBar`. In this example, it adds a `Sample Action` to the toolbar and to a new Editor Menu. It also demonstrates in `setActionPage` and `getAction` methods how to disable global actions based on the editor's state. In this example, global actions related to copy, paste, undo, etc. are disabled for the second and third page of the wizard.

The code for the `SampleNewWizard.java` class:

```

package com.ibm.tutorial.templates.wizards;

import org.eclipse.jface.viewers.IStructuredSelection;
import org.eclipse.jface.wizard.Wizard;
import org.eclipse.ui.INewWizard;
import org.eclipse.ui.IWorkbench;
import org.eclipse.core.runtime.*;
import org.eclipse.jface.operation.*;
import java.lang.reflect.InvocationTargetException;
import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.core.resources.*;
import org.eclipse.core.runtime.CoreException;
import java.io.*;
import org.eclipse.ui.*;
import org.eclipse.ui.ide.IDE;

/**
 * This is a sample new wizard. Its role is to create a new
 * file resource in the provided container. If the
 * container resource (a folder or a project) is selected
 * in the workspace when the wizard is opened, it will
 * accept it as the target container. The wizard creates
 * one file with the extension "mpe". If a sample multi-
 * page editor (also available as a template) is registered
 * for the same extension, it will be able to open it.
 */

```

```
public class SampleNewWizard extends Wizard
    implements INewWizard
{
    private SampleNewWizardPage page;
    private ISelection selection;

    /**
     * Constructor for SampleNewWizard.
     */
    public SampleNewWizard() {
        super();
        setNeedsProgressMonitor(true);
    }

    /**
     * Adding the page to the wizard.
     */

    public void addPages() {
        page = new SampleNewWizardPage(selection);
        addPage(page);
    }

    /**
     * This method is called when 'Finish' button is pressed
     * in the wizard. We will create an operation and run it
     * using wizard as execution context.
     */
    public boolean performFinish() {
        final String containerName = page.getContainerName();
        final String fileName = page.getFileName();
        IRunnableWithProgress op = new IRunnableWithProgress()
        {
            public void run(IProgressMonitor monitor)
                throws InvocationTargetException
            {
                try {
                    doFinish(containerName, fileName, monitor);
                } catch (CoreException e) {
                    throw new InvocationTargetException(e);
                } finally {
                    monitor.done();
                }
            }
        };
        try {
            getContainer().run(true, false, op);
        } catch (InterruptedException e) {
            return false;
        } catch (InvocationTargetException e) {
            Throwable realException = e.getTargetException();
            MessageDialog.openError(
                getShell(),
                "Error", realException.getMessage());
            return false;
        }
        return true;
    }

    /**
     * The worker method. It will find the container, create
     * the file if missing or just replace its contents, and
     * open the editor on the newly created file.
     */

    private void doFinish(
        String containerName,
        String fileName,
        IProgressMonitor monitor)
```

```

        throws CoreException
    {
        // create a sample file
        monitor.beginTask("Creating " + fileName, 2);
        IWorkspaceRoot root =
            ResourcesPlugin.getWorkspace().getRoot();
        IResource resource = root.findMember(
            new Path(containerName));
        if (!resource.exists() ||
            !(resource instanceof IContainer))
        {
            throwCoreException("Container \"" +
                containerName + "\" does not exist.");
        }
        IContainer container = (IContainer) resource;
        final IFile file =
            container.getFile(new Path(fileName));
        try {
            InputStream stream = openContentStream();
            if (file.exists()) {
                file.setContents(stream, true, true, monitor);
            } else {
                file.create(stream, true, monitor);
            }
            stream.close();
        } catch (IOException e) {
        }
        monitor.worked(1);
        monitor.setTaskName("Opening file for editing...");
        getShell().getDisplay().asyncExec(new Runnable() {
            public void run() {
                IWorkbenchPage page =
                    PlatformUI.getWorkbench().
                        getActiveWorkbenchWindow().getActivePage();
                try {
                    IDE.openEditor(page, file, true);
                } catch (PartInitException e) {
                }
            }
        });
        monitor.worked(1);
    }

    /**
     * We will initialize file contents with a sample text.
     */
    private InputStream openContentStream() {
        String contents =
            "This is the initial file contents for *.mpe " +
            "file that should be word-sorted in the " +
            "Preview page of the multi-page editor";
        return new ByteArrayInputStream(contents.getBytes());
    }

    private void throwCoreException(String message)
        throws CoreException
    {
        IStatus status =
            new Status(IStatus.ERROR,
                "com.ibm.tutorial.templates",
                IStatus.OK, message, null);
        throw new CoreException(status);
    }

    /**
     * We will accept the selection in the workbench to see
     * if we can initialize from it.
     * @see IWorkbenchWizard#init(
     *     IWorkbench, IStructuredSelection)
     */

```

```
    */
    public void init(IWorkbench workbench,
        IStructuredSelection selection)
    {
        this.selection = selection;
    }
}
```

The wizard in this code is a standard wizard that displays pages, then generates code in the `doFinish` method. What is most interesting about this class is that in `performFinish`, it demonstrates how to interact with Eclipse's process monitor for tasks that take a long time.

SampleNewWizardPage.java class

The code for the `SampleNewWizardPage.java` class:

```
package com.ibm.tutorial.templates.wizards;

import org.eclipse.jface.wizard.WizardPage;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.SWT;
import org.eclipse.core.resources.*;
import org.eclipse.core.runtime.Path;
import org.eclipse.swt.events.*;
import org.eclipse.ui.dialogs.ContainerSelectionDialog;
import org.eclipse.jface.viewers.*;

/**
 * The "New" wizard page allows setting the container for
 * the new file as well as the file name. The page
 * will only accept file name without the extension OR
 * with the extension that matches the expected one (mpe).
 */

public class SampleNewWizardPage extends WizardPage {
    private Text containerText;
    private Text fileText;
    private ISelection selection;

    /**
     * Constructor for SampleNewWizardPage.
     * @param pageName
     */
    public SampleNewWizardPage(ISelection selection) {
        super("wizardPage");
        setTitle("Multi-page Editor File");
        setDescription("This wizard creates a new file with " +
            "*.mpe extension that can be opened by a " +
            "multi-page editor.");
        this.selection = selection;
    }

    /**
     * @see IDialogPage#createControl(Composite)
     */
    public void createControl(Composite parent) {
        Composite container = new Composite(parent, SWT.NULL);
        GridLayout layout = new GridLayout();
    }
}
```

```

container.setLayout(layout);
layout.numColumns = 3;
layout.verticalSpacing = 9;
Label label = new Label(container, SWT.NULL);
label.setText("&Container:");

containerText =
    new Text(container, SWT.BORDER | SWT.SINGLE);
GridData gd =
    new GridData(GridData.FILL_HORIZONTAL);
containerText.setLayoutData(gd);
containerText.addModifyListener(new ModifyListener() {
    public void modifyText(ModifyEvent e) {
        dialogChanged();
    }
});

Button button = new Button(container, SWT.PUSH);
button.setText("Browse...");
button.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent e) {
        handleBrowse();
    }
});
label = new Label(container, SWT.NULL);
label.setText("&File name:");

fileText =
    new Text(container, SWT.BORDER | SWT.SINGLE);
gd = new GridData(GridData.FILL_HORIZONTAL);
fileText.setLayoutData(gd);
fileText.addModifyListener(new ModifyListener() {
    public void modifyText(ModifyEvent e) {
        dialogChanged();
    }
});
initialize();
dialogChanged();
setControl(container);
}

/**
 * Tests if the current workbench selection is a suitable
 * container to use.
 */
private void initialize() {
    if (selection!=null
        && selection.isEmpty()==false
        && selection instanceof IStructuredSelection)
    {
        IStructuredSelection ssel =
            (IStructuredSelection)selection;
        if (ssel.size()>1) return;
        Object obj = ssel.getFirstElement();
        if (obj instanceof IResource) {
            IContainer container;
            if (obj instanceof IContainer)
                container = (IContainer)obj;
            else
                container = ((IResource)obj).getParent();

            containerText.setText(
                container.getFullPath().toString());
        }
        fileText.setText("new_file.mpe");
    }
}

/**

```

```
* Uses the standard container selection dialog to
* choose the new value for the container field.
*/
private void handleBrowse() {
    ContainerSelectionDialog dialog =
        new ContainerSelectionDialog(
            getShell(),
            ResourcesPlugin.getWorkspace().getRoot(),
            false,
            "Select new file container");
    if (dialog.open() == ContainerSelectionDialog.OK) {
        Object[] result = dialog.getResult();
        if (result.length == 1) {
            containerText.setText(
                ((Path)result[0]).toOSString());
        }
    }
}

/**
 * Ensures that both text fields are set.
 */
private void dialogChanged() {
    String container = getContainerName();
    String fileName = getFileName();

    if (container.length() == 0) {
        updateStatus("File container must be specified");
        return;
    }
    if (fileName.length() == 0) {
        updateStatus("File name must be specified");
        return;
    }
    int dotLoc = fileName.lastIndexOf('.');
    if (dotLoc != -1) {
        String ext = fileName.substring(dotLoc + 1);
        if (ext.equalsIgnoreCase("mpe") == false) {
            updateStatus("File extension must be \"mpe\"");
            return;
        }
    }
    updateStatus(null);
}

private void updateStatus(String message) {
    setErrorMessage(message);
    setPageComplete(message == null);
}

public String getContainerName() {
    return containerText.getText();
}

public String getFileName() {
    return fileText.getText();
}
}
```

This code is a simple SWT wizard page. The most interesting characteristic of this class is in the `handleBrowse` method. This method uses Eclipse's `ContainerSelectionDialog` for displaying the folders in the workspace.

Section 10. Summary

This tutorial just scratched the surface of Eclipse plug-in development. It demonstrated how to create, test, debug, and package a basic plug-in. Then it showed how to start developing your own plug-ins using the seven plug-in templates as a starting point:

- Hello World
- Popup Menu
- Property Page
- View
- Perspective Extension
- Editor
- Multi-page Editor

You should now have a good understanding of how to take advantage of the plug-in templates in Eclipse to reduce the coding and learning curve of generating your own custom plug-ins.

Downloads

Description	Name	Size	Download method
Complete source	os-eclplgnssource.zip	37KB	HTTP

[Information about download methods](#)

Resources

Learn

- Visit Eclipse.org for the latest downloads, news, articles, and announcements of Eclipse.
- The [Platform Plug-in Developers Guide](#) provides documentation on extending the base Eclipse Platform.
- The [Eclipse Platform API Specification](#) contains the Eclipse Platform Javadoc.
- The [JDT Programmers Guide](#) provides documentation on extending the Java Development Tools.
- The [Eclipse JDT API Specification](#) contains the JDT Javadoc.
- The [PDE Plug-in Guide](#) provides documentation on extending the Plug-in Development Environment.
- The [PDE API Reference](#) contains the PDE Javadoc.
- *The Definitive Guide to SWT and JFace*, by Rob Warner with Robert Harris (Apress, 2004), provides thorough coverage of these two newer technologies.
- "[Migrate your Swing applications to SWT](#)" is a tutorial that contains a comparison and mapping between SWT and Swing.
- "[Debugging with the Eclipse Platform](#)" explains how to use the Eclipse debugger.
- Find out more about [Eclipse plug-ins](#).
- Find out more about Eclipse at the [Eclipse Plug-in Resource Center and Marketplace](#).
- Visit the developerWorks [Open source zone](#) for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.

Get products and technologies

- Innovate your next open source development project with [IBM trial software](#), available for download or on DVD.

Discuss

- Get involved in the developerWorks community by participating in [developerWorks blogs](#).

About the author

Christopher Judd

Christopher Judd is the president and primary consultant for [Judd Solutions, LLC](#), international speaker, open source evangelist, [Central Ohio Java Users Group](#) coordinator, and co-author of *Enterprise Java Development on a Budget* and *Pro Eclipse JST*. He has spent eight years developing software in the insurance, retail, government, manufacturing, service, and transportation industries. His current focus is consulting, mentoring, and training with Java, J2EE, J2ME, Web services and related technologies.