Plug a Swing-based development tool into Eclipse

How to integrate a Swing editor into the Eclipse Platform

Terry Chan (terrych@ca.ibm.com)
Software engineer, IBM Canada Ltd.
October 2002

Learn how to integrate a stand-alone Swing-based editor into the Eclipse Platform as a plug-in. Using simple techniques, you can share resources between the Swing tool, the Eclipse Platform, and various SWT widgets -- and these resources can communicate through mutual awareness. Tool vendors who want to bring Eclipse-based development tools to market with a minimal amount of re-coding will also find this article helpful.

Introduction

The Eclipse Platform provides a robust set of services and APIs for tool development. It smoothes the integration between tools from disparate vendors, creating a seamless environment for different types of development work.

One of the software components of the Eclipse Platform is SWT. Although not a core component set of the Platform, SWT is integral because it provides a set of Java-based GUI widgets to the product and plug-in developers. SWT is operating system independent and very portable, yet its underlying JNI interface delivers native Platform look-and-feel and performance.

Overall, SWT provides a good solution for developers and tool vendors who want to write plug-ins that operate well in the Platform's various frameworks and that are visually appealing. However, SWT suffers from its rather low level of interoperability with Java's Swing GUI widgets. For instance, Swing and SWT employ completely different event handling mechanisms. This difference often makes a GUI that is a hybrid of Swing and SWT unusable.

Some work has been done to provide an interface between Swing and SWT to give an acceptable level of compatibility, such as the `org.eclipse.swt.internal.swt.win32.SWT_AWT` utility class that enables a developer to embed a Swing widget into SWT. However, these methods are still experimental and not yet officially supported -- hence the "internal" moniker in the package name.

This poor interoperability presents an unfortunate obstacle for both the Eclipse project and tool vendors. Currently, a large number of software development and test tools provide a user interface written in Swing. Considerable time and investment from a vendor would be needed to port an existing tool with a sophisticated Swing GUI to SWT. Despite all the inherent advantages offered by the Eclipse Platform, the poor interoperability between Swing and SWT makes the development effort less attractive.

This article shows you how to:

- Launch a Swing-based editor to edit any Java file in the Eclipse Platform Workbench with the name "ThirdParty.java"
- Bring source code changes made in the Swing editor back into the Workbench
- Use the Preference Page framework to control the attributes of the Swing editor
- Make the Swing editor "Workbench-aware"
- Launch an SWT widget from the Swing editor

This article introduces some simple techniques to achieve the above, without using any unsupported APIs. No internal classes are referenced, and all general plug-in rules are adhered to. To get the most out of these techniques, you should have basic knowledge in writing plug-ins and using the Plug-in Development Environment, as well as access to the source code of the Swing-based editor.

Hypothetical Swing editor: Ed

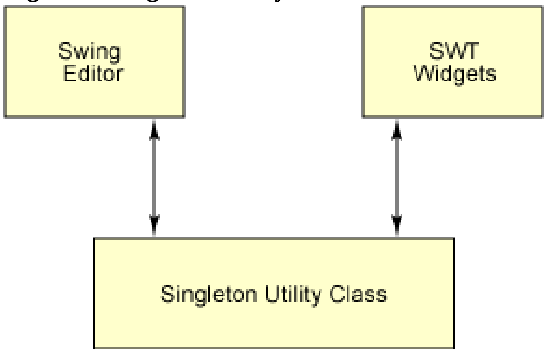
To simulate a real-world tools integration scenario, let's use a hypothetical Swing-based editor called "Ed". Here are some of Ed's characteristics:

- Ed is a Swing-based editor.
- Ed extends JFrame.
- Ed operates only on Java files with a specific name: ThirdParty.java.
- Ed has a **JEditorPane** and a **JButton** as private fields. The JEditorPane displays the source of ThirdParty.java. The **JButton** saves the source code.
- Ed is a stand-alone Java application with a `main()` method.
- Ed is designed to be run from a command prompt. It is not aware of the Eclipse Platform Workbench.

The basic concept

Due to the limitations of Swing and SWT interoperability, direct communication (such as event handling) between the two is difficult to achieve. One way to achieve it, without using unsupported APIs or bending any plug-in development rules, is to avoid embedding them into each other, and instead let them have their separate Frames. The plug-in class, or a Singleton utility class, will handle the communication between them, as shown in Figure 1. For instance, a **JButton** on Ed could cause an SWT Shell to appear, displaying some Workbench-specific attributes (such as Project References) of the edited ThirdParty.java.

Figure 1. Singleton utility class



Editor integration

The primary goal of integration is to develop a plug-in that uses Ed as the default editor for any ThirdParty.java files found in the Workbench.

Preparing the plug-in project

In the Workbench, create a new plug-in project "org.eclipse.jumpstart.editorintegration", and select the "Create plug-in using a template wizard" option in the Create plug-in project wizard. Click **Next**. Check the option "Add default instance access" and click **Finish**. The Workbench switches to the Plug-in Development Perspective. A blank plug-in manifest file, as well as the plug-in class `EditorintegrationPlugin`, which extends `AbstractUIPlugin`, is created automatically. A private static instance of the plug-in class, as well as the getter method, is also be generated.

The plug-in manifest file editor should be open; if not, double-click plugin.xml to launch it.

The following libraries are needed for this plug-in. Add them to the plug-in project's Java Build Path:

- ECLIPSE_HOME/plugins/org.eclipse.core.resources/resources.jar
- ECLIPSE_HOME/plugins/org.eclipse.jdt.core/jdtcore.jar
- ECLIPSE_HOME/plugins/org.eclipse.jdt.ui/jdt.jar
- ECLIPSE_HOME/plugins/org.eclipse.swt/swt.jar
- ECLIPSE_HOME/plugins/org.eclipse.ui/workbench.jar

The plug-in manifest file

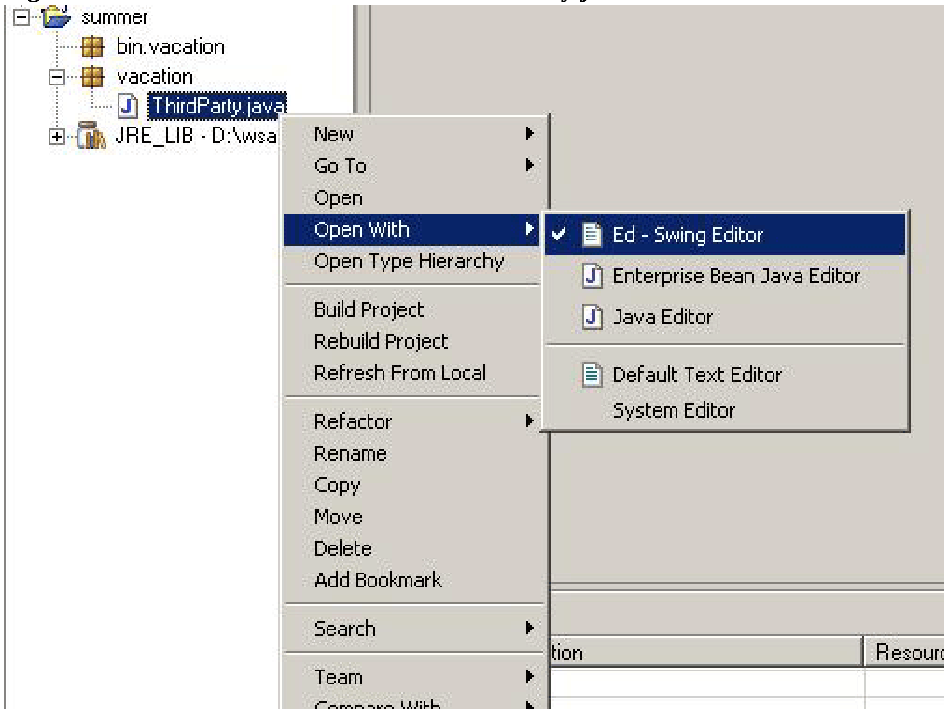
Because this plug-in operates only on Java files with the name ThirdParty.java, we need to specify an editor for them. In the plug-in manifest file editor, switch to the Extensions tab, and add the extension point "Internal and External Editors". Set default to "true", name to "Ed - Swing Editor", filenames to "ThirdParty.java", and launcher to "org.eclipse.jumpstart.editorintegration.EdLauncher". The source of the added extension point should look like Listing 1:

Listing 1. Adding an extension point

```
<extension point="org.eclipse.ui.editors">
<editor
   name="Ed - Swing Editor"
   default="true"
   icon="icons/thirdparty.gif"
   filenames="ThirdParty.java"
   launcher="org.eclipse.jumpstart.editorintegration.EdLauncher"
   id="org.eclipse.jumpstart.editorintegration.ed">
</editor>
</extension>
```

Ed is now the default editor for all ThirdParty.java files as shown in Figure 2.

Figure 2. Ed is the default editor for all ThirdParty.java files



Note: Be sure to include the icons/thirdparty.gif file, which is displayed as the default editor in the "Open With" menu for all ThirdParty.java files.

Integrating Ed source code
Import the source code of Ed into the plug-in project. How Ed is invoked is up to you. The plug-in class could contain an Ed as a private field with a public getter method:

Listing 2. Ed as a private field

```
private Ed ed = null;

public Ed getEd()
{
   if (ed == null)
   {
      ed = new Ed ();
   }
   return ed;
}
```

Alternatively, the plug-in class could return a separate instance of Ed for every ThirdParty.java file launched. You can implement either approach in a Singleton utility class maintained and provided by the plug-in.

The editor launcher
Because the plug-in uses the extension point `org.eclipse.ui.editors`, it must provide the Eclipse Platform with an editor launcher class as specified in the manifest file.

Create the class `org.eclipse.jumpstart.editorintegration.EdLauncher` to implement the interface IEditorLauncher (if this interface is not found, ensure the workbench.jar file is included in the Project Path; see Preparing the plug-in project). Be sure to check the "Inherited abstract methods" option in the Wizard.

Every time a ThirdParty.java file is double-clicked, the Eclipse Platform executes `EdLauncher.open(IFile)` to invoke the default editor of this file type. The Platform passes the clicked artifact as an IFile to the method. In this case, IFile is a Java source file, and so you could cast it as an ICompilationUnit.

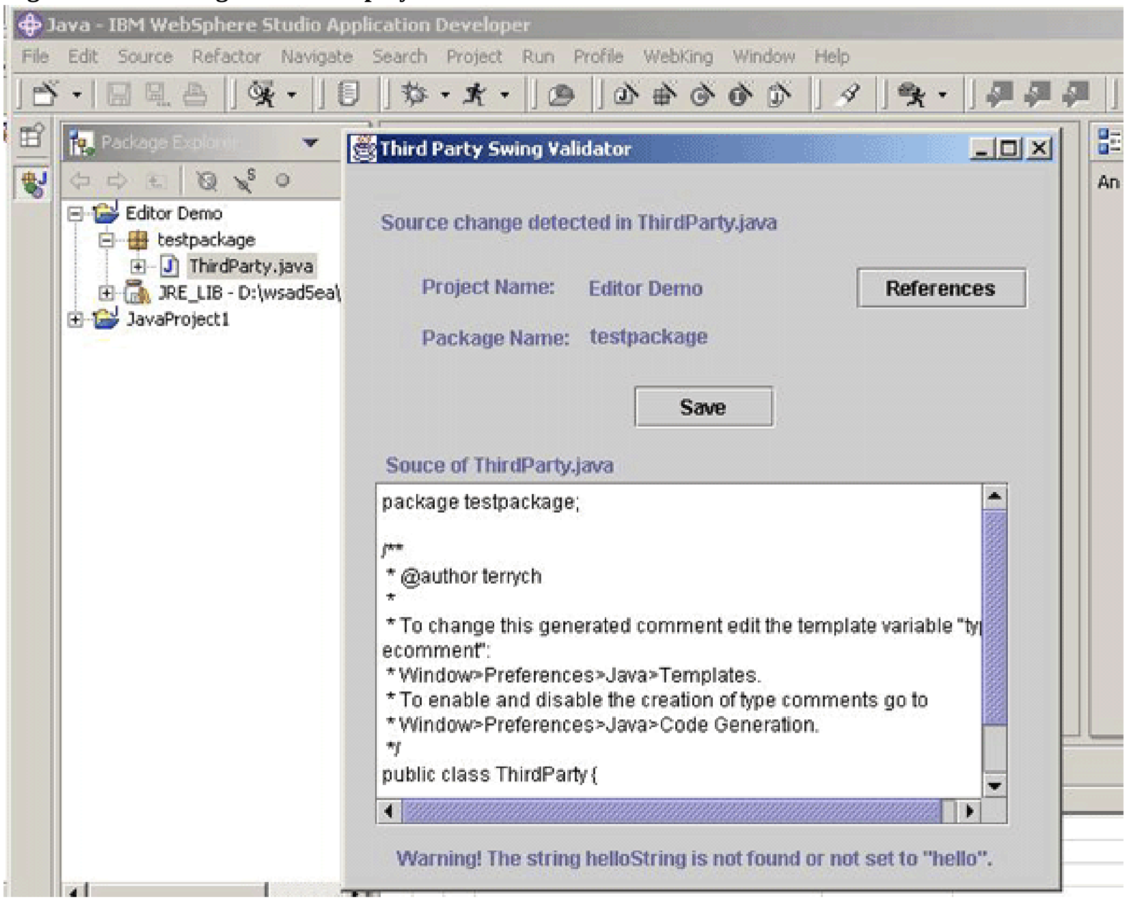
Because Ed is not designed to work with JDT objects, you must extract the source content from the ICompilationUnit and put it into Ed for viewing:

```
EditorintegrationPlugin.getDefault().getEd().getEditorPane().setText
   (ICompilationUnit.getSource());

EditorintegrationPlugin.getDefault().getEd().show();
```

Upon execution of the `show()` method, Ed is displayed as a JFrame outside of the main Workbench window (see Figure 3). The project name and package name of the edited ThirdParty.java are recorded by the plug-in. This information will be vital when you attempt to save changes made in Ed.

Figure 3. The Swing editor is displayed outside of the Workbench


Java - IBM WebSphere Studio Application Developer
File Edit Source Refactor Navigate Search Project Run Profile WebKing Window Help
Package Explorer
Editor Demo
testpackage
ThirdParty.java
JRE_LIB - D:\wsad5ea\
JavaProject1
Third Party Swing Validator
References
Save

Source change detected in ThirdParty.java

Project Name: Editor Demo

Package Name: testpackage

Souce of ThirdParty.java

```
package testpackage;

/**
 * @author terrych
 *
 * To change this generated comment edit the template variable "ty
ecomment":
 * Window>Preferences>Java>Templates.
 * To enable and disable the creation of type comments go to
 * Window>Preferences>Java>Code Generation.
 */
public class ThirdParty {
```

Warning! The string helloString is not found or not set to "hello".

Round-tripping: Bringing source changes back into the Workbench

A traditional editor would save source code in flat files, in a binary repository, or to a source code control system. As an editor, Ed needs some means of saving the changes to the source code that it displays.

Ed has a "Save" button (a **JButton**) as described in The Swing editor: Ed. When pressed, the `actionPerformed()` method is called, and the Save button fires an event. An object that implements an event listener receives the event and performs the source save operation.

You may use the Singleton utility class (see The editor launcher) as the object that implements the event listener. Upon receiving an event object from the Save button, the utility class extracts the source from Ed, and puts it into the corresponding Workbench object. The actual work of saving to the file system is delegated to the Eclipse Platform.

Potentially, multiple files could have the same name in the Workbench. This is where the project and package name of ThirdParty.java are useful. This information is stored by the plug-in. The exact implementation approach is up to you. Assuming the editor stores the information, you could use the following code snippet in the utility class:

Listing 3. Managing file names

```
public void saveButtonPressed() {
   try {
      IWorkspaceRoot root = ResourcesPlugin.getWorkspace().getRoot();

      IProject myProj = root.getProject(getEd().getProjectname());

      IFolder myFolder = myProj.getFolder(getEd().getPackageName());

      IJavaElement myPackageFragment = JavaCore.create(myFolder);

      if (myPackageFragment != null) {
         IPackageFragment packageFrag = (IPackageFragment)myPackageFragment;

         String sourceFromEd = getEd().getJEditorPane1().getText();

         ICompilationUnit icu = packageFrag.getCompilationUnit("ThirdParty.java");

         icu.getBuffer().setContents(sourceFromValidator);

         icu.save(null, true);
      }
      else {
         System.out.println("myPackageFragment is null.");
      }
   } catch (Exception e) {
      e.printStackTrace();
   }
}
```

Reverse round-tripping
Listing 3 takes care of the "forward" round-tripping. "Backward" round-tripping is also needed to bring into Ed any changes made in ThirdParty.java with the Eclipse Platform's JDT Java editor.

The utility class may implement the interface `org.eclipse.jdt.core.IElementChangedListener`, which you can use to track changes made to any IElements, including ICompilationUnit. The method `elementChanged(ElementChangedEvent)` is called when a source change is introduced to a Java file in the Workbench.

You need to selectively filter out the IElement changes that do not concern the Ed plug-in. One way of filtering is to extract and examine the `IJavaElementDelta` object from the `IElementChangedEvent` argument. For instance, the following statement filters irrelevant source changes in the context of the Ed plug-in:

Listing 4. Filtering irrelevant source changes

```
IJavaElementDelta delta = event.getDelta();

if (delta != null) {
   if(delta.getElement().getElementName().equalsIgnoreCase("ThirdParty.java")) {
      //code to update Ed's editor panel.
   }
}
```

For editors of non-Java artifacts, the IElementChangedListener cannot capture changes made in the Workbench. The Eclipse Platform provides the interface `org.eclipse.core.resources.IResourceChangeListener` to handle changes made to non-Java resources.

Preference pages
To provide users with rich, easy-to-use features, a tool should provide configurable options accessible via startup parameters, or via a GUI that is not part of the editor's core graphical interface. In the case of a plug-in for the Eclipse Platform, it is highly recommended that these options be configured via the Platform's Preference Page framework (Window -> Preferences).

For example purposes, let's control the color of Ed as a configurable option using a Platform preference page.

Adding a preference page extension point in the plug-in manifest file
A preference page is defined as an extension point in the Eclipse Platform. To use it, add it in the plug-in manifest file editor, or put in the following code into the plugin.xml:

Listing 5. Adding a preference page to the plugin.xml

```
<extension
   id="org.eclipse.jumpstart.editorintegration.pref"
   name="Ed Preference"
   point="org.eclipse.ui.preferencePages">
<page
   name="Swing Editor Preference Page"
   class="org.eclipse.jumpstart.editorintegration.EdPreferencePage1"
   id="Swing Editor Preference Page"
</page>
</extension>
```

The preference page class
Preference pages extend `org.eclipse.jface.preference.PreferencePage`. In this example, the simple preference page consists of three slider bars at the maximum value of 255, representing the colors (red, green, and blue) of the `java.awt.Color` object of Ed.

Create the class `org.eclipse.jumpstart.editorintegration.EdPreferencePage1` in the plug-in project as specified in the manifest file. It must extend `org.eclipse.jface.preference.PreferencePage` and implement the interface `org.eclipse.ui.IWorkbenchPreferencePage`.

The preference page presents a similar coding challenge as the editor launcher: How would JFace/SWT communicate with Swing? Fortunately, the same approach applies. For instance, the `performApply()` method may look like this:

Listing 6. The performApply() method

```
protected void performApply() {
   int red = redSWTSlider.getSelection();
   int green = greenSWTSlider.getSelection();
   int blue = blueSWTSlider.getSelection();

   java.awt.Color newColor = new java.awt.Color(red, green, blue);
   EditorintegrationPlugin.getDefault().getEd().getContentPane().setBackground(
      newColor);
}
```

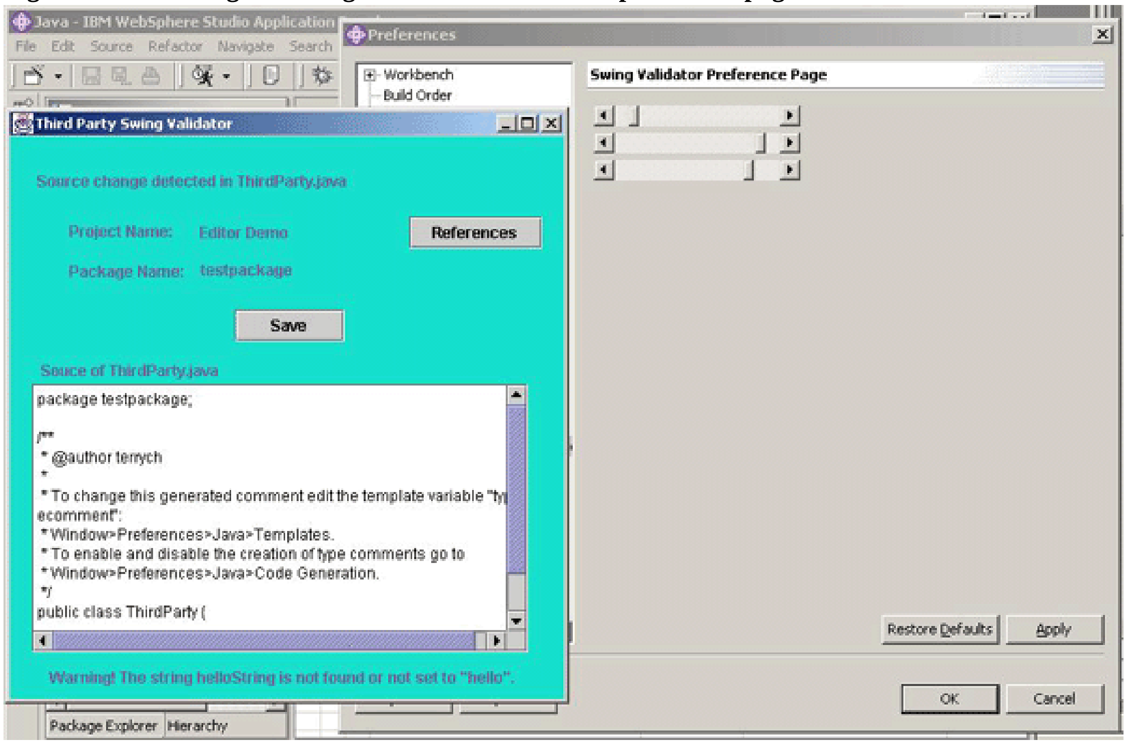
The plug-in should use the Platform's Preference Store mechanism to store the configured value, as would any other plug-ins. The `performOk()` method may look like this:

Listing 7. The performOk() method

```
public boolean performOk() {
   getPreferenceStore().setValue("redValue", redSWTSlider.getSelection();
   getPreferenceStore().setValue("greenValue", greenSWTSlider.getSelection());
   getPreferenceStore().setValue("blueValue", blueSWTSlider.getSelection());
   return true;
}
```

Controlling the Swing editor's color from the preference pages is shown in Figure 4.

Figure 4. Controlling the Swing editor's color from the preference pages



Source change detected in ThirdParty.java

Project Name: Editor Demo

Package Name: testpackage

Souce of ThirdParty.java

```
package testpackage;

/**
 * @author terrych
 *
 * To change this generated comment edit the template variable "ty
ecomment":
 * Window>Preferences>Java>Templates.
 * To enable and disable the creation of type comments go to
 * Window>Preferences>Java>Code Generation.
 */
public class ThirdParty {
```

Warning! The string helloString is not found or not set to "hello".

Workbench awareness

Because most editors were originally designed as stand-alone Java applications, they are oblivious to the existence of the Workbench. They may not be able to handle some of the Platform's environment attributes, limiting the level of intimacy of the editor's integration with the Platform. In order to provide users with a smoother and more coherent development experience, developers and plug-in vendors should seriously consider enhancing their existing Swing tools to be Workbench aware.

For instance, Ed was coded to work directly with file system-based Java files. Therefore, the concepts of the Platform's Java Projects and Project References are foreign to it. In this section, we'll add a **JButton** to Ed to launch an SWT dialog box showing the referenced projects of the edited ThirdParty.java. From the user's point of view, he clicks on a Swing widget, triggering an

SWT window that displays Workbench-specific information, giving the illusion that the Swing editor, SWT, and the Workbench are tightly interacting with each other.

Enhancing the editor

Assuming you have access to the Ed source code, you can add additional Swing widgets for additional Workbench awareness functionality. Add a **JButton** to the main content pane of the editor, which would then launch an SWT dialog. Set the text of the **JButton** to "Referenced Project".

The Referenced Project button's event handling mechanism would work similar to that of the Save button's (see Round-tripping: Bringing source changes back into the workbench). The plug-in utility class would listen for events from this button. Upon receiving an event object fired by the Referenced Project Button, the utility class would perform the necessary operations to retrieve the project reference info and display it in SWT.

Retrieving the project reference information

Before the SWT dialog can be displayed, the plug-in needs to find out which projects in the Workbench are referenced by the project that contains the edited ThirdParty.java. This is the job of the plug-in class, and it may use a method as shown in Listing 8, where the passed-in string argument is the name of the project:

Listing 8. Retrieving the project reference info

```
private String[] getReferencedProjectArray(String arg) {
   String[] projectNameArray = null;

   try {
      IProject[] referencedProjects =
         ResourcesPlugin.getWorkspace().getRoot().getProject(
            arg).getReferencedProjects();

      int referencedProjectsLength = referencedProjects.length;

      if (referencedProjectsLength == 0) {
         projectNameArray = new String[1];
         projectNameArray[0] = "none";
      }
      else {
         projectNameArray = new String[referencedProjectsLength];
         for (int i=0; i < referencedProjectsLength; i++) {
            projectNameArray[i] = referencedProjects[i].getName();
         }
      }
      return projectNameArray;
   } catch (Exception e) {
      e.printStackTrace();
      return null;
   }
}
```

The SWT dialog

Exactly how the Project Referenced SWT dialog would look is up to the plug-in GUI designers. In this example, a simple SWT Shell with a `List` object (to display the referenced projects) will suffice:

Listing 9. SWT Shell with a List object

```
public class SWTProjectReferenceFrame implements Runnable {
   private Shell shell;
   private Display display;
   Thread myThread;

   public void run() {
      open();
   }

   public void open() {
      display = new Display();
      shell = new Shell(display);
      shell.setLayout(new org.eclipse.swt.layout.GridLayout());
      shell.setText("Projects Referenced - SWT Frame");
      shell.setSize(400, 400);
      createListGroup();
      shell.open();

      while (!shell.isDisposed()) {
         if (!display.readAndDispatch()) {
            EditorintegrationPlugin.getDefault().getEd().repaint();
            display.sleep();
         }
      }
      myThread = null; // disposing the thread when the SWT window is disposed.
   }

   // Other methods appear here ...
}
```

The method `createListGroup()` prepares the `List` object, and sets its content to contain projectNameArray (see Retrieving the project reference information).

Listing 10. Preparing the List object

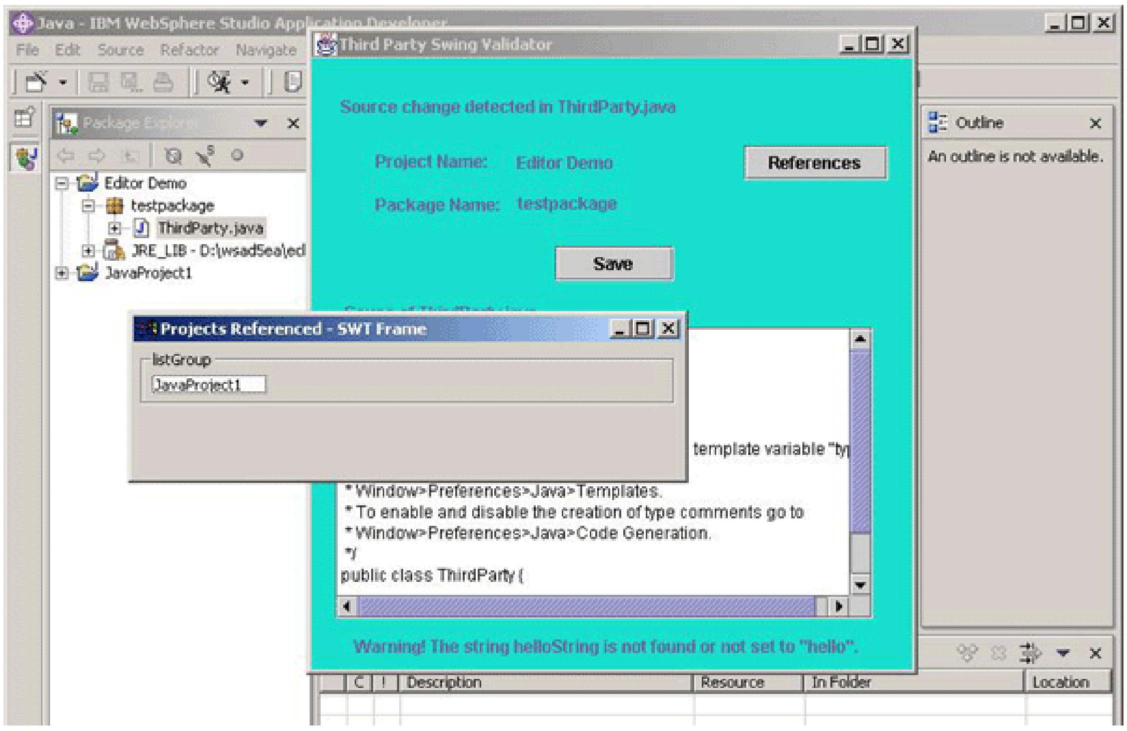
```
private void createListGroup() {
   Group listGroup = new Group(shell, SWT.NULL);
   listGroup.setLayout(new GridLayout());
   listGroup.setLayoutData(new GridData(GridData.GRAB_HORIZONTAL |
                                        GridData.HORIZONTAL_ALIGN_FILL |
                                        GridData.VERTICAL_ALIGN_FILL));
   listGroup.setText("listGroup");

   List list = new List(listGroup, SWT.V_SCROLL);
   list.setItems(projectNameArray);
}
```

Depending on how the SWT dialog is launched, you may need to execute the SWT window in a separate thread (as indicated by the `myThread` object in Listing 10) to avoid repainting problems in the Swing editor.

Launching an SWT frame from a **Swing** button is shown in Figure 5.

Figure 5. Launching an SWT frame from a Swing button



Source change detected in ThirdParty.java

Project Name: Editor Demo

Package Name: testpackage

An outline is not available.

```
template variable "ty
* Window>Preferences>Java>Templates.
* To enable and disable the creation of type comments go to
* Window>Preferences>Java>Code Generation.
*/
public class ThirdParty {
```

Warning! The string helloString is not found or not set to "hello".

Conclusion

The techniques described here offer an interim solution that can help you quickly integrate Swing-based tools into the Eclipse Platform. However, whenever possible possible, you should use tightly integrated SWT/JFace components over existing Swing widgets. For instance, instead of using individual preference dialog frames to handle user preferences, an editor should use the Eclipse Platform's Preference Page framework as the central entry point for configuring a plug-in.

Even though the concepts in this article are relatively simple and easy to implement, do not leave the Swing widgets as permanent fixtures in a plug-in. To harness and exploit all the services in the Eclipse project, you should gradually decrease the amount of legacy Swing code in your plug-ins in favor of various frameworks provided by the Eclipse project.

Resources

- For more background, read the eclipse.org article "How You've Changed! Responding to resource changes in the Eclipse workspace" by John Arthorne.
- Get an introduction to the Eclipse Platform and how it operates in the article "Working the Eclipse Platform" by Greg Adams and Marc Erickson (*developerWorks*, November 2001).
- For advice on writing Eclipse plug-ins for the international market, read "Internationalizing your Eclipse plug-in". Your roadmap starts with a quick review of the goals and challenges of internationalization, followed by detailed instructions. An added sidetrip is a look at how these steps were applied to the internationalization of the Eclipse Platform itself (*developerWorks*, June 2002).
- Then, to verify your translated plug-in files, read "Testing your internationalized Eclipse plug-in", which includes strategies for dealing with common errors -- and a download for a handy plug-in for comparing property files. This plug-in will help your translation testers find errors more quickly (*developerWorks*, July 2002).
- Take the online *developerWorks* tutorial "Developing and deploying plug-ins for WebSphere Studio" (short registration required).
- Visit the eclipse.org Web site for more information on Eclipse.

About the author

Terry Chan is a Software engineer at IBM in Toronto, Ontario. He worked as an analyst for IBM Global Services, specializing in OS/2 core dump analysis. He later joined the IBM Toronto Software Laboratory where he was a customer service analyst for VisualAge for Java. In 2001 he joined the WebSphere Studio Application Developer (WSAD) Jumpstart team, whose primary goal is to help ISVs to create commercial offerings based on WSAD. You can contact Terry at terrych@ca.ibm.com.

e-mail it!

What do you think of this article?

○ Killer! (5) ○ Good stuff (4) ○ So-so; not bad (3) ○ Needs work (2) ○ Lame! (1)

Comments?

Submit feedback