Kick up the speed and quality of your Java development with Java plug-in programming in SAP NetWeaver Developer Studio

by Sebastien Cherry

Sebastien Cherry
SAP Technical Consultant,
Netlab Inc.

Sebastien Cherry received a bachelor's degree in management computing in 2002, and is currently pursuing a Ph.D. in computer engineering at École Polytechnique de Montréal. In his early career, Sebastien worked at a startup company where he acquired a taste for technological entrepreneurship. He joined SAP in 2002 as a software developer, mainly in the mySAP CRM group. In 2006 he left SAP to co-found his own IT company. In addition to providing expert SAP technical consultations, he has developed computer programs for which patent applications have been submitted. You can reach him at sebastien.cherry@netlab.ca.

If you're a Java developer or manager of a Java development team, you probably already know that SAP NetWeaver Developer Studio is SAP's new integrated development environment (IDE) for building and deploying Java applications.[1] Perhaps you also know that SAP NetWeaver Developer Studio was built on IBM's extensible, open source Eclipse platform — a platform that enables companies (such as SAP) to quickly set up new development environments (such as SAP NetWeaver Developer Studio) as a set of plug-ins to the platform.[2]

What you may not know, however, is that Eclipse offers an open application programming interface (API) you can use to develop your own plug-ins, or to leverage the thousands available for free on the Web,[3] to expand and enhance those provided with SAP NetWeaver Developer Studio.

Here are just a few examples of already-developed plug-ins for the Eclipse platform that are available on the Web:

- A wizard-based plug-in that automatically sets up a new development environment within SAP NetWeaver Developer Studio (i.e., a complete setup for a specific project and software version that usually needs to be done manually in preparation for the project), saving time when creating new projects or maintaining previous versions of the software. The

[1] Karl Kessler's article "Get Started Developing, Debugging, and Deploying Custom J2EE Applications Quickly and Easily with SAP NetWeaver Developer Studio" (*SAP Professional Journal*, May/June 2004) shows just how easy it can be to create custom Java Web applications using SAP NetWeaver Developer Studio.

[2] The Eclipse platform provides the basic functionality you expect all IDEs to have, such as project management, debugging, and a user interface organized into windows, views, and menus. To create a new development tool, like SAP NetWeaver Developer Studio, companies such as SAP need to develop plug-ins that implement the "specifics" of the target language and platform (in the case of SAP, for example, a Web Dynpro compiler, deployment details for SAP Web Application Server, and so on).

[3] You can find thousands of available plug-ins at www.eclipse-plugins.info/.

plug-in asks a few questions, adds all the required libraries and sources, sets environmental settings, and then builds and deploys the application optimally for the task at hand.

- A plug-in that automatically synchronizes project sources with the deployed application when the project is saved, which in turn saves the development team hours when debugging the project (possibly days when debugging large projects). Without the plug-in, the developer has to build and deploy the project manually each time changes are made.

- A basic performance-testing plug-in that identifies slow spots in the developer's application. The plug-in is coded to record and display how long each page takes to load as the developer navigates through the application to test it.

As you can see, not only do plug-ins offer you a way to greatly improve your efficiency, alleviate pain points, or just make the tool more user friendly, but they also can be shared with your team. This makes the return on investment (ROI) in developing plug-ins worthwhile.

This article shows you how to build, develop, and deploy your own plug-ins to enhance SAP NetWeaver Developer Studio functionality. (The process of downloading and installing other people's plug-ins from the Web is self explanatory.) To gain first-hand experience with programming in the Eclipse environment, we'll build a simple plug-in that emulates a console that displays messages written by Java applications to the SAP Web Application Server (SAP Web AS) J2EE Engine standard output. This example plug-in could be a real timesaver when troubleshooting new Java applications. Without it, you would have to inspect log files manually in the J2EE Engine Log Viewer to view standard outputs.

Before we can start coding the example plug-in, there are some important concepts that you need to understand. We'll begin with an overview of the Eclipse platform architecture and the plug-in programming paradigm. We'll then build the various components of a plug-in project in SAP NetWeaver Developer Studio and code the three classes needed for the example console plug-in. After coding each class, we will test and debug the plug-in to ensure it works the way we want it to. Finally, we'll walk through how to deploy the plug-in.

To understand this article, you'll need some basic knowledge of Java programming, as well as some basic experience working with SAP NetWeaver Developer Studio. Also, to build the components

Note!

As of the 2004 version of SAP NetWeaver Developer Studio, the local test SAP Web AS system runs as its own service. Since it no longer runs as a standard process (as was the case with the former SAP J2EE Engine), standard outputs do not appear in an output console view, but are instead written to a log file. The plug-in we develop in this article reads this output and writes it to a substitute console window.

Note!

Example code for the article, along with a compiled plug-in, is available at www.SAPpro.com in the "Downloads" section.

Attention managers!

This article will help you understand how these plug-ins can shave hours or days off your project plans and how they can help your developers improve the consistency and quality of their code.

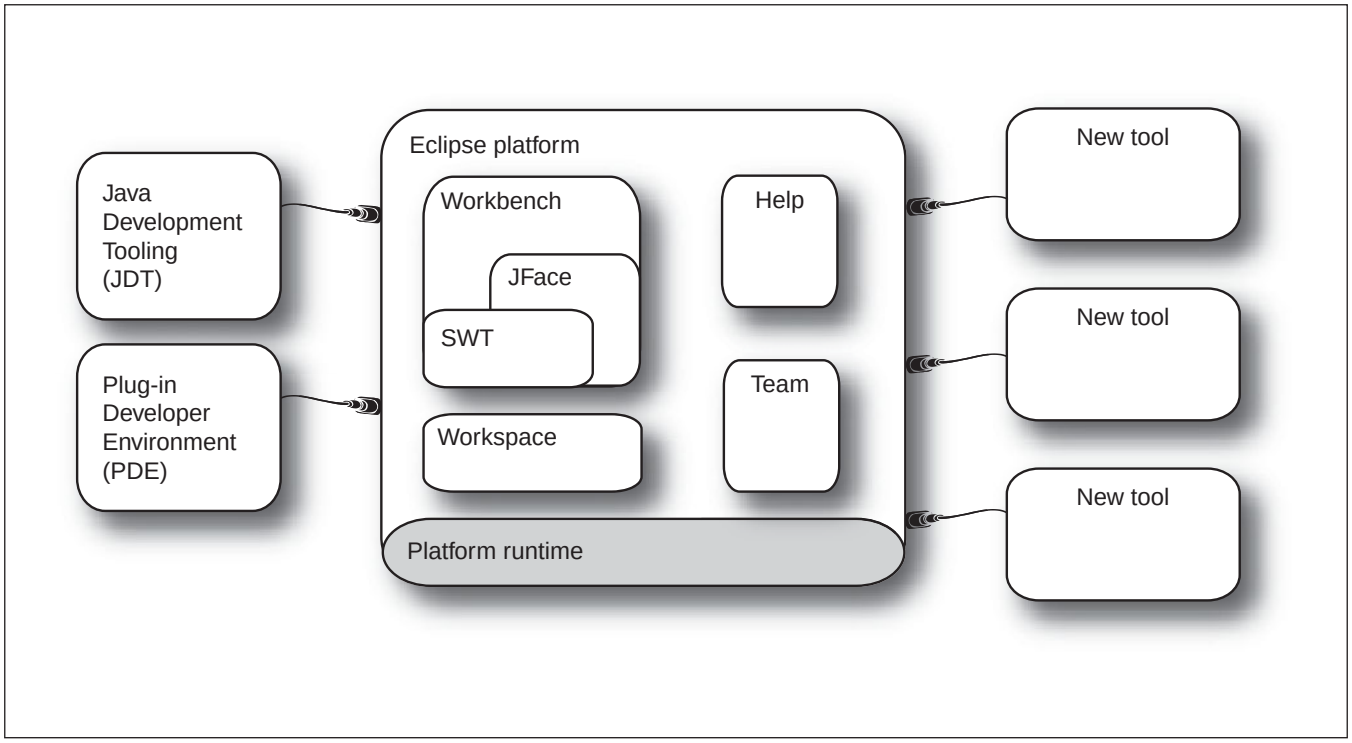

Figure 1 Eclipse platform architecture

of the plug-in project, you'll need the 2004 or 2004s version of SAP NetWeaver Developer Studio (the architecture changed[4] in the 2004 version). You can download a free Sneak Preview from the SAP Developer Network (SDN) at http://sdn.sap.com/.

Eclipse programming overview

The Eclipse platform was designed as a basic infrastructure for rapidly building robust IDEs (see **Figure 1**[5]). Consequently, the platform itself does not offer a large set of end user functionality; rather it provides an extensible architecture based on a plug-in model as well as an extensive user interface API, the Standard Widget Toolkit (SWT), which runs on top of multiple operating systems (OSs) and provides all the required abstraction to make developed plug-ins fully portable to all supported OSs.

Using the common workbench model, the Eclipse platform defines an open architecture that enables developers to build their own tools that plug into the workbench using predefined hooks called *extension points*. The Eclipse platform architecture comprises several subsystems, which are plug-ins themselves. The lowest layer, the platform runtime, defines possible extension points and implements the plug-in model dynamically by discovering available plug-ins during startup and maintaining information about them in the plug-in registry in the Eclipse system files (more specifically, in the workplace metadata files). The workspace component defines the APIs that are available for working with file system resources and managed by the system, such as projects, files, and folders. Built on top of the aforementioned components, the workbench component implements the "cockpit" that allows the developer to navigate through resources and use available plug-in tools. The workbench also defines the extension points that allow extending it with different

[4] As of 2004, the local test SAP Web AS installed by the tool runs as an external Microsoft Windows service and therefore runs in a separate process from SAP NetWeaver Developer Studio.

[5] Figure 1 is adapted from the Eclipse Platform Plug-in Developer Guide at http://help.eclipse.org/.

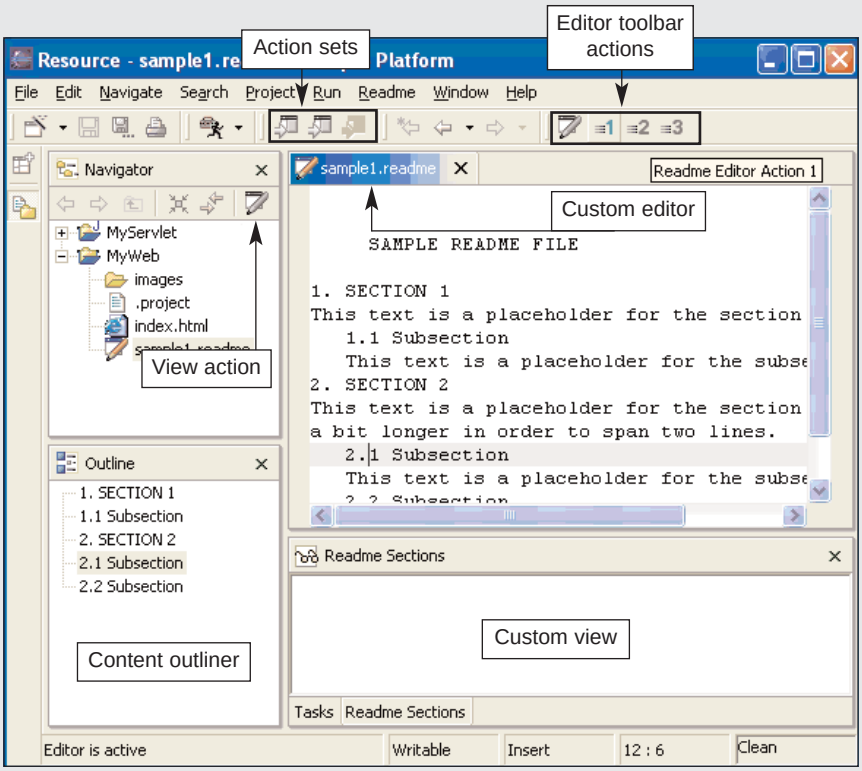

Figure 2 An example of a custom view with available UI components

user interface (UI) components, such as views, actions, and perspectives,[6] and provides complete Java UI toolkits, such as SWT and JFace.

These APIs take advantage of the full capabilities of each supported OS, which is not the case with the standard Java Abstract Window Toolkit (AWT).[7] The APIs enable applications built with these toolkits to reflect the look-and-feel of the underlying OS GUI while maintaining a consistent API on all platforms. Help and Team components enable extension points for both. They provide help and documentation in the form of browsable books as well as a team programming model for managing and versioning resources, complemented by the debug support component, which provides a language-independent debug model and UI classes for building debuggers.

Getting back to the extension points provided by the workbench component, **Figure 2**[8] provides an overview of the UI components that can be added to the Eclipse cockpit. As you can see, the Eclipse platform provides UI components[9] that you can use when you create a plug-in. For example, you might develop a plug-in that extends menus in the Navigator view, adds actions to the workbench menus and toolbars, and defines a custom view or a content outliner.

[6] A perspective is essentially a set of available views and actions logically grouped together to fulfill a particular task. For instance, the "Java" perspective allows Java application programming, the "Debug" perspective allows debugging, and the "Plug-in Development" perspective allows plug-in development.

[7] Java AWT is included in the Java Foundation Classes (JFC) and represents the standard API for designing GUIs in Java applications. More information can be found at www.java.sun.com/products/jdk/awt/.

[8] Figure 2 is adapted from the Eclipse Platform Plug-in Developer Guide at http://help.eclipse.org/.

[9] For more information about customizing an Eclipse platform, go to http://help.eclipse.org/.

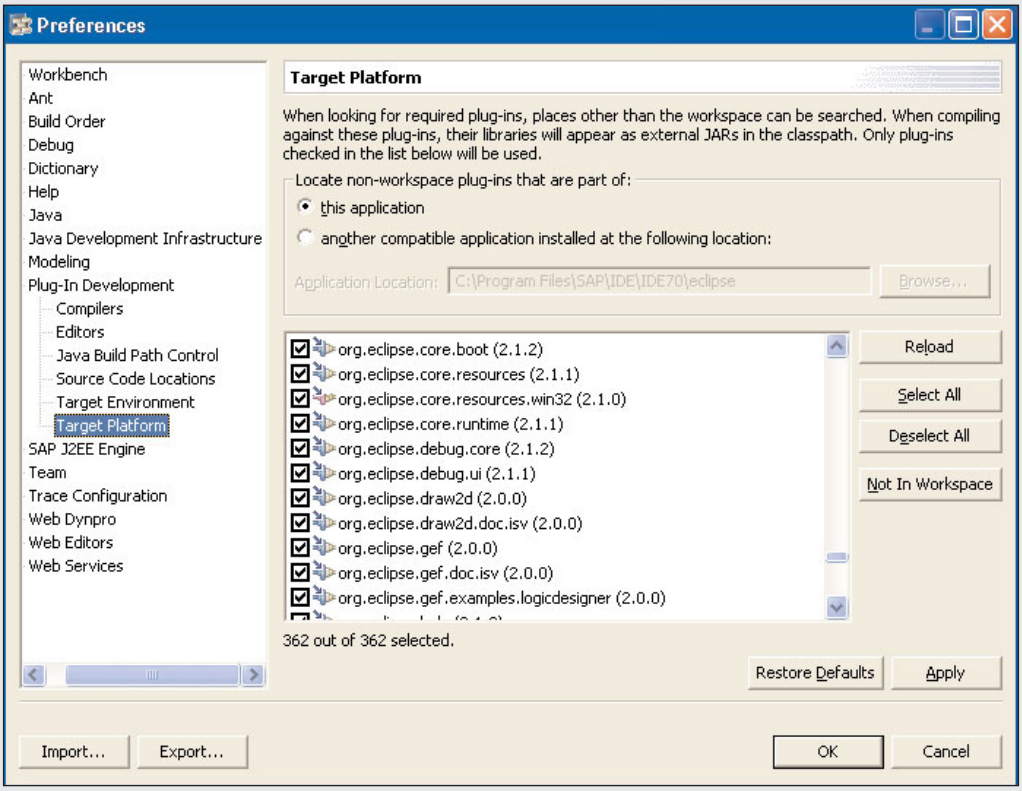

Figure 3 Loading available plug-ins into the workbench

Further customization might position the editor on the left and the navigator on the right. The plug-in developed in the course of this article will function as a custom view that emulates a console that displays messages written in the standard output from Java Web applications running on the SAP Web AS J2EE Engine.

With a basic understanding of the Eclipse platform, let's begin by creating the plug-in project, which lays the foundation for our example plug-in.

Creating a plug-in project

Before we create the example plug-in project, we need to perform a simple preliminary step. In SAP NetWeaver Developer Studio, navigate to Window → Preferences → Plug-In Development → Target Platform (see **Figure 3**). Click on Select All to mark all the plug-ins displayed in the list, click on Apply, and then click on OK. This step is required because, as mentioned earlier, available plug-ins are loaded by the workbench during startup. Consequently, to test or debug a plug-in developed in a particular instance of the workbench, you need to start another instance of the workbench so that the plug-in developed in the first instance can be loaded and launched. This preliminary step enables the required plug-ins at runtime. Selecting them all avoids having to resolve any dependencies between them if one plug-in requires another at runtime for using shared services, resources, etc. After we create the example plug-in project, we will walk through a few post-creation tasks that make required icons available and set a main icon for the plug-in.

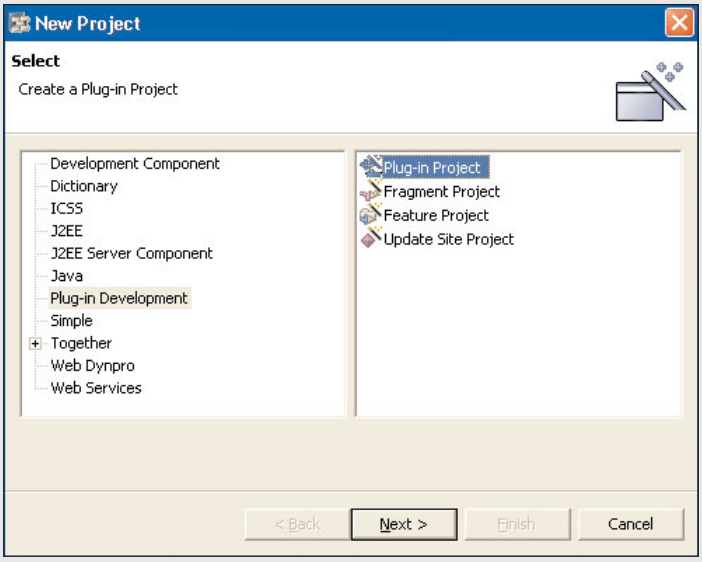

Figure 4 Creating a plug-in project using the project generation wizard

Tip!

As mentioned earlier, to create a plug-in project like the example in this article, you'll need the 2004 or 2004s version of SAP NetWeaver Developer Studio. You can download a free Sneak Preview version from http://sdn.sap.com/.

Tip!

It is recommended that you use the same parameters as those in the example instead of specifying your own values since further explanations will be based on this example. For example, make sure the Use default checkbox is selected as shown in Figure 5, so the contents of the project will be saved under the default Eclipse workspace directory.

With the plug-ins loaded into SAP NetWeaver Developer Studio, we are now ready to create the plug-in project. We will use the project generation wizard provided by the Plug-in Developer Environment (PDE). Let's begin:

1. In SAP NetWeaver Developer Studio, follow the menu path File → New → Project → Plug-in Development → Plug-in Project (see **Figure 4**) and click on Next.
2. As shown in **Figure 5**, specify a name for the project (in the example, SAP Pro J2EE Console) and then click on Next.
3. Next, specify a plug-in ID, which is a tag that uniquely identifies the plug-in in the Eclipse workbench. This ID should be the same name as the main class of the plug-in itself to make it easier to reference in the code. In the example, shown in **Figure 6**, the ID is com.sappro.example.plugin.jconsole.J2EEConsolePlugin. Here you also specify the type of project you are creating (i.e., a Java project for the example in this article vs. a

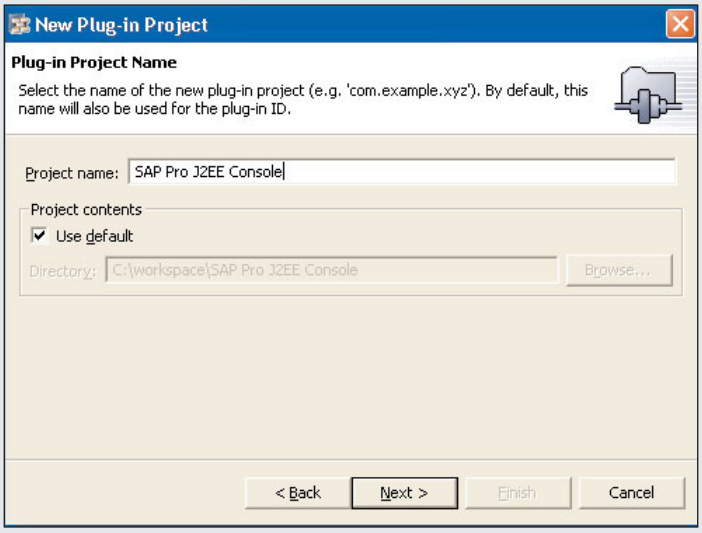

Figure 5 Specifying the name of the example plug-in project

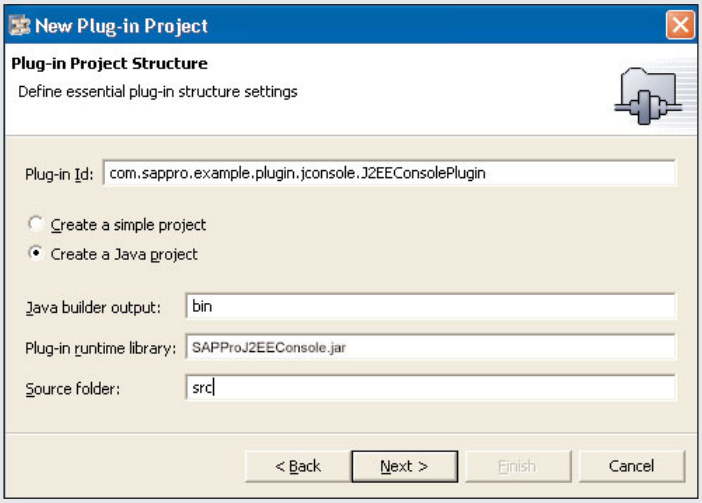

Figure 6 Specifying the structure of the example plug-in project

simple project, which might be a blank project). The Java builder output is bin, the runtime library is SAPProJ2EEConsole.jar, and the source folder is src, which is the file system folder where the Java source classes will be stored. Click on Next.

Note!

The plug-in ID must be unique among all available plug-ins in the workbench.

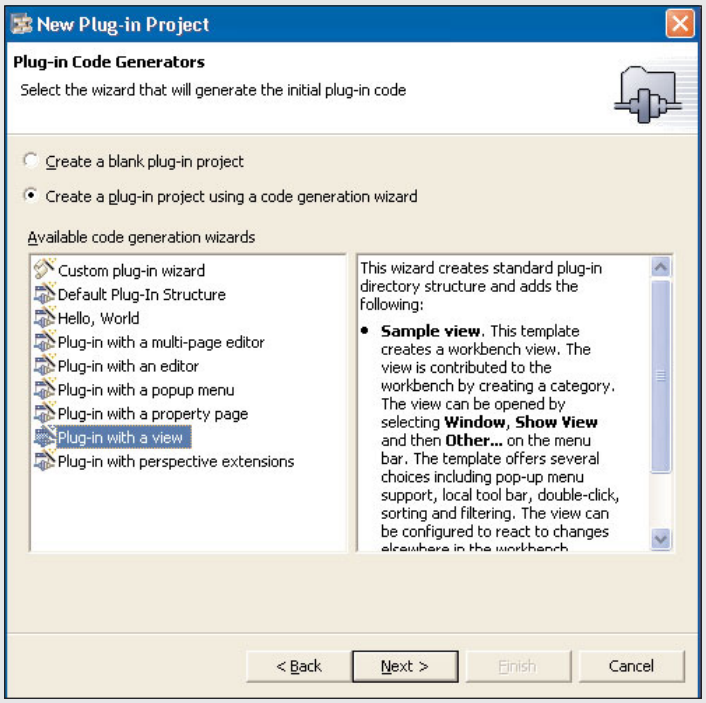

Figure 7 Enabling the initial code generation for the example plug-in project

4. Next you need to tell the program to generate the initial plug-in code.[10] Select the "Plug-in with a view" code generation wizard (see **Figure 7**), which means that when you click on Finish to generate the project, the wizard will automatically create a view base class and the settings for the example plug-in project.

5. Next, you need to specify the general information for the plug-in. As shown in **Figure 8**, enter a name for the plug-in project (SAP Pro J2EE Console), and then specify a version (which defaults to 1.0.0 when you first create a plug-in), a provider name (e.g., sappro.com), and finally, a name for the plug-in base class (in this example, com.sappro.example.plugin.jconsole.J2EEConsole Plugin). Keep the default selections to generate the code for the plug-in base class and for adding a default instance access. However, deselect the options for supporting resource bundles and accessing the workspace; these options are not required for our example (to keep it as simple as possible, UI text elements such as button and field labels will be hard-coded). Click on Next.

6. The next part of the wizard provides options for creating the MainView class and categorizing the plug-in view. Enter the Java package name (com.sappro.example.plugin.jconsole), the view class name (MainView), and view name (SAP Pro J2EE Console). The view name will be displayed in the Show View dialog, where all view plug-ins are displayed. To make sure that the plug-in is properly categorized, you need to provide the ID of an existing category (e.g., org.eclipse.debug.ui is the category ID containing all views for debugging) or create a new one. Let's create a new one (see **Figure 9**) — example.sappro.com, with the

[10] The code generation wizard generates a default structure depending on the type of project you are creating; skeletons of main classes can be generated as well. However, if the default generation does not fit your particular needs, you can choose to create a blank project in step 3, which allows you to structure the project as required.

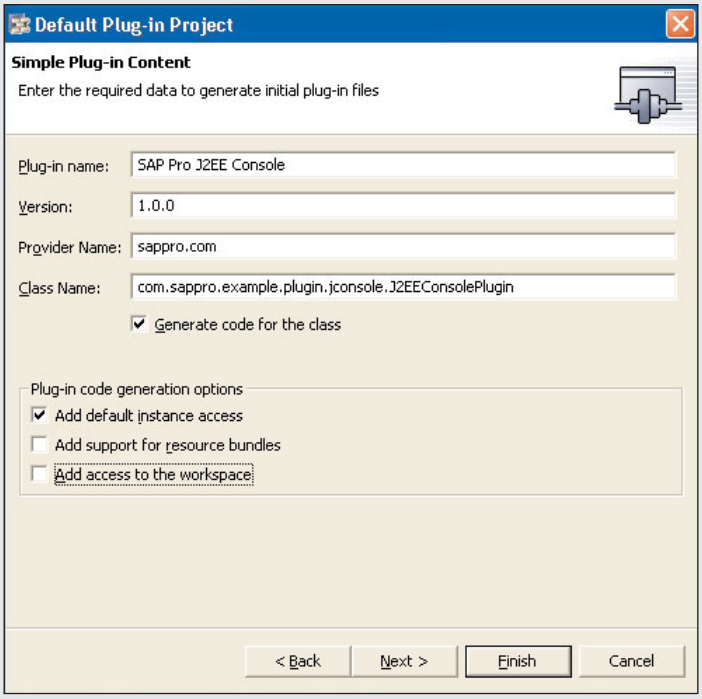

Figure 8 Specifying the default structure for the example plug-in project

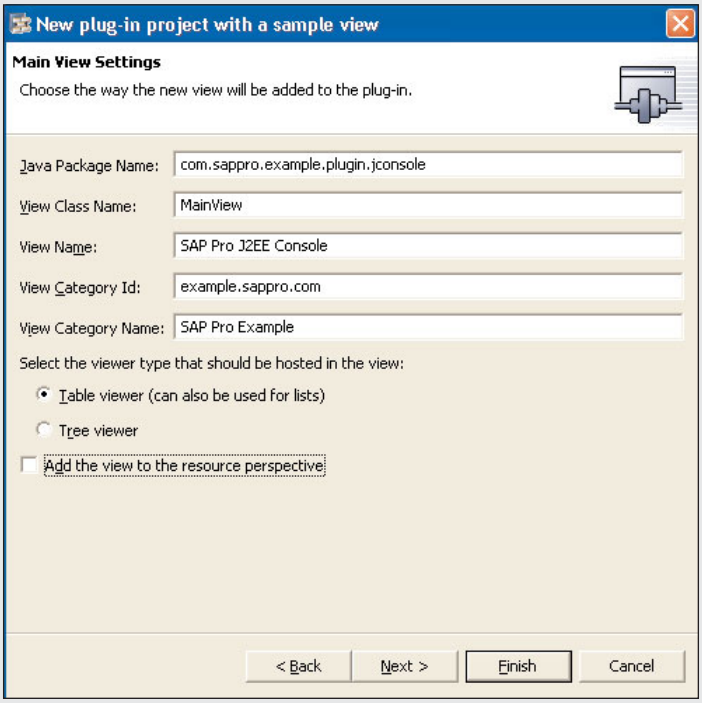

Figure 9 Creating the MainView class and categorizing the plug-in view

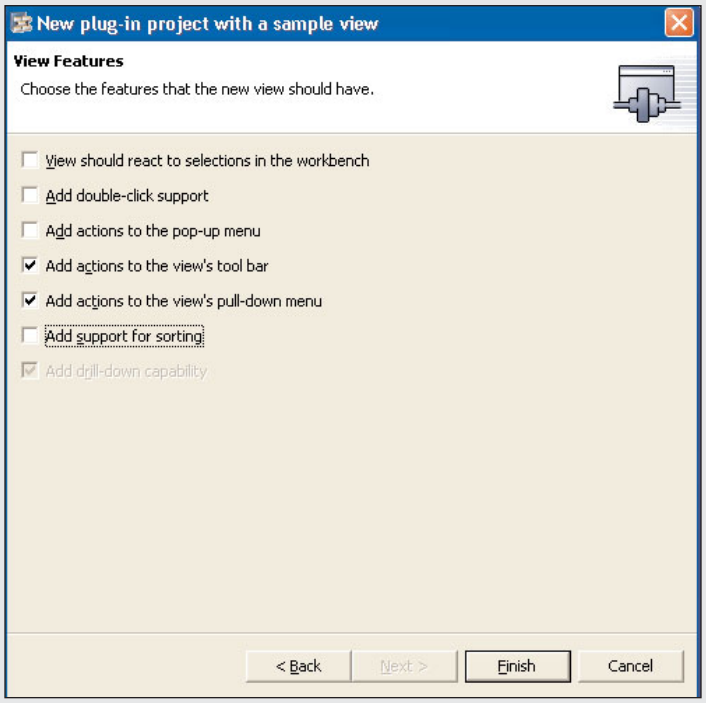

Figure 10 Specifying the options for the main view

category name SAP Pro Example. Finally, deselect the option for adding the view to the resource perspective,[11] since this is not relevant for the kind of plug-in we are building. Click on Next.

7. The last page of the wizard (**Figure 10**) provides options for generating functionalities that the new view should include. For the example plug-in project, deselect all options except the ones for adding actions to the toolbar and to the pull-down menu of the view. This generates example code in the MainView class, some of which you will remove for the example, but it minimizes the code that you would have to enter manually. Click on Finish.
8. A Confirm Perspective Switch dialog[12] appears during the project generation; click on Yes. This will open the Plug-in Development perspective, which includes relevant views and actions that you can use for the development of new plug-ins.

Once the generation wizard is done, the project appears in the Plug-in Development window, as shown in **Figure 11**.

On the left, you can see the project structure. It includes the Java sources under src, including the J2EEConsolePlugin class that represents the plug-in base class (which we don't currently need to modify) as well as the MainView class that represents the class

[11] The resource perspective in the Eclipse workbench is the perspective that includes all plug-ins that allow managing project resources. This perspective normally includes the Navigator view for displaying file system resources for each project, for example. The view to which I'm referring here is the main view of the plug-in we are currently building.

[12] The wizard detects the nature of the project created and then checks if a more suitable perspective is available in the Eclipse workbench. If so, a Confirmation Perspective dialog appears, asking if the user wants to switch to another perspective. In this case, the wizard suggests switching to the Plug-in Development perspective.

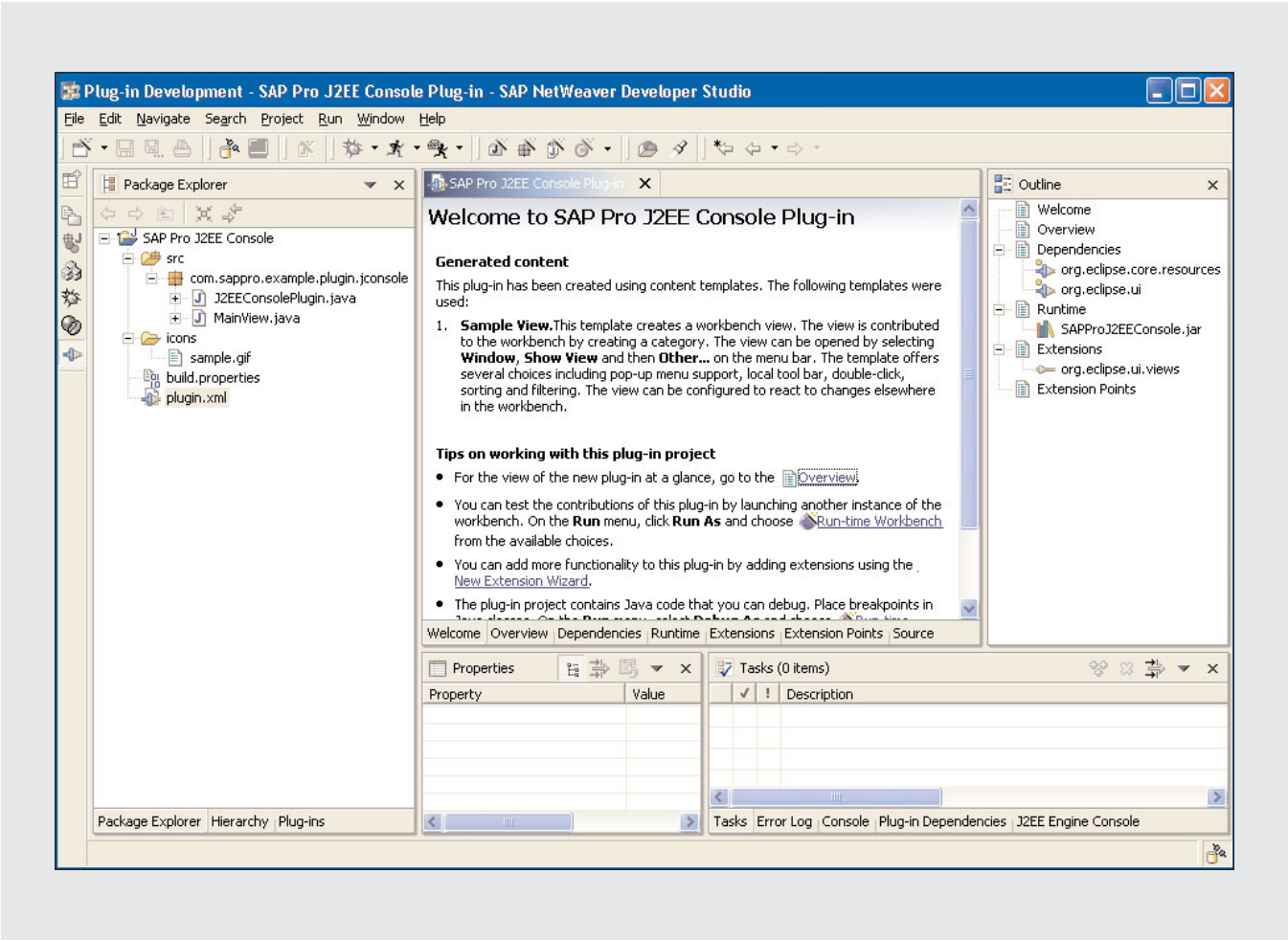

Figure 11 The example console plug-in project

that will render the view of the plug-in, which will emulate the console. In addition, the project structure includes the required icons under the icons folder, the build.properties that will be used for the build process of the plug-in, and finally the plugin.xml file that is the descriptor of the plug-in.

Post-creation tasks

Before coding the new plug-in, we must perform a few post-creation tasks to prepare the plug-in project framework for the functionality we want the plug-in to have. First, to design the plug-in UI, we need icons. You can find the required icons under eclipse/plugins/com.tssap.util/icons/sap.[13] Copy the following icons into the icons directory of the plug-in project:

- s_b_dele.gif
- s_b_info.gif
- s_connec.gif
- s_discon.gif
- s_s_locl.gif
- s_s_loop.gif
- s_x__sap.gif

[13] The Eclipse default directory is usually C:/Program Files/SAP/JDT/ in a typical Microsoft Windows environment.

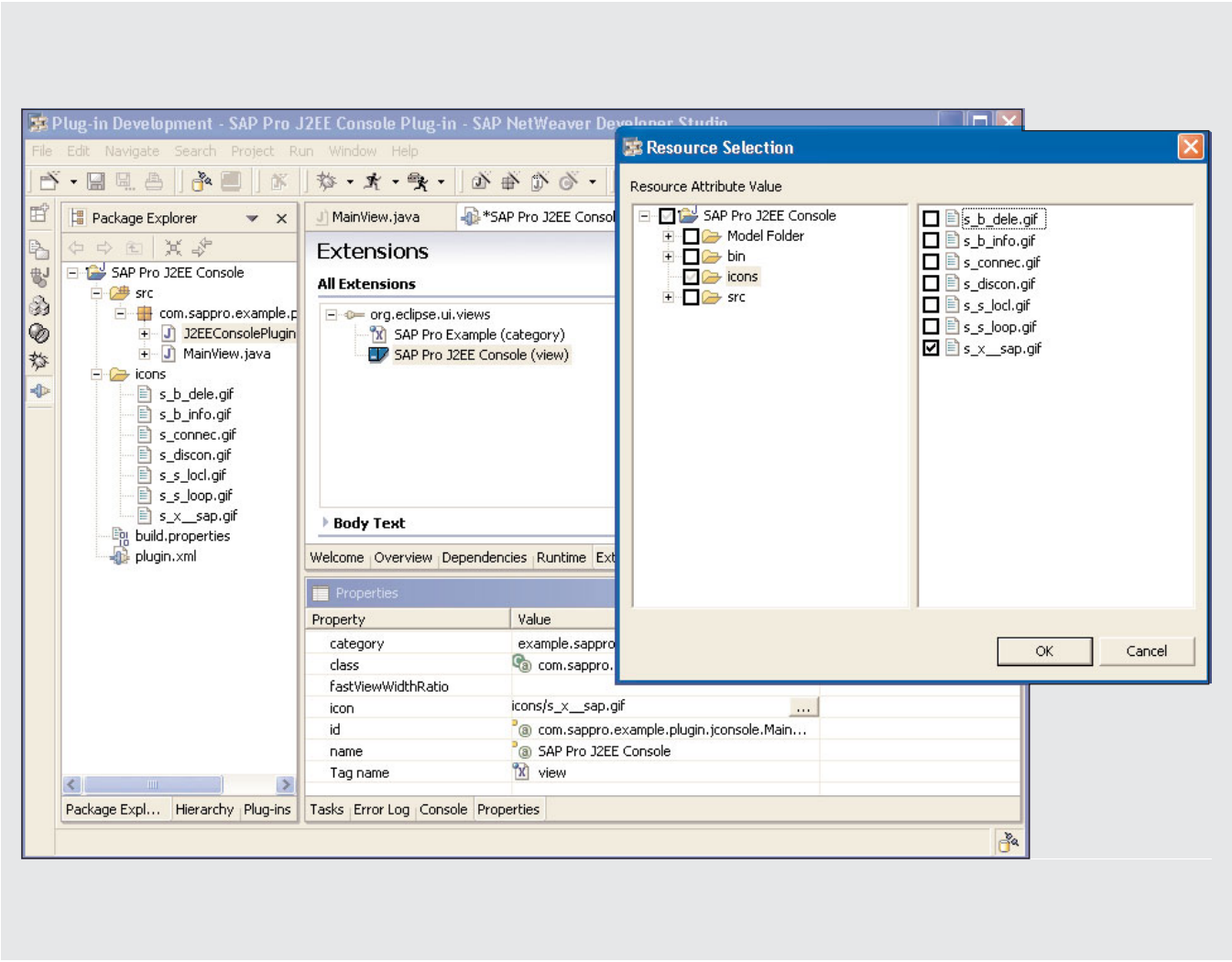

Figure 12 Setting the icon for the main view of the example plug-in

Since the icon sample.gif, generated by the wizard, will not be required for the example plug-in, you can just delete it.

Next, we need to set the main icon of the example console plug-in. This icon will be displayed next to the plug-in name in the title bar of the view, as well as listed in the Show View dialog, where all view plug-ins are categorized. To set the icon, open the plugin.xml file in the Package Explorer view. In the Outline view on the right, expand Extensions and then select the SAP Pro J2EE Console (view) node under the org.eclipse.ui.views extension. To customize the icon in the Properties view, select the s_x__sap.gif icon file, as shown in **Figure 12**. Click on OK to return to the development window.

Finally, under Dependencies, add a dependency to the plug-in com.tssap.util. This dependency will enable us to use utilities classes for loading our icons from the SAP UI Toolkit.

> ***Note!***
>
> Don’t forget to save the file!

Coding the example plug-in

The coding of the plug-in is pretty simple and limited to three classes (excluding the automatically generated J2EEConsolePlugin class, which we don't need to modify). We will modify the MainView class, which has been generated automatically, to define our custom view UI. The PreferencePage class will display a preference page, allowing us to collect basic settings needed to run the console, such as the server directory. Finally, the Console class will represent the console logic in the form of a thread looking for new text to display in the view.

Thus, we will build the plug-in in three steps, starting with the main view, then the preference page, and finally the console logic. After each step, we will test and debug the plug-in to monitor our progress.

Step 1: Code the MainView class

Figure 13 shows the code for the MainView class. This code, which will be explained by sections,[14] uses a combination of the code generated automatically

[14] The sections of code appear in alternating colors of gray and white for easier reading.

```
package com.sappro.example.plugin.jconsole;

import java.text.*;
import java.util.*;

import org.eclipse.jface.action.*;
import org.eclipse.jface.dialogs.*;
import org.eclipse.jface.resource.*;
import org.eclipse.jface.viewers.*;                                    ❶
import org.eclipse.swt.*;
import org.eclipse.swt.graphics.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.ui.*;
import org.eclipse.ui.part.*;

import com.tssap.util.ui.image.*;

public class MainView extends ViewPart {

   private static Composite parent;
   private static TableViewer tableViewer;                             ❷
   private static Table table;

   private static Action connectAction,
      deleteAction,
      scrollLockAction,                                                ❸
      infoAction;
```

Figure 13 Coding the MainView class

Continues on next page

Figure 13 *continued*

```
private static boolean scrollLocked = false, connected = false;                ❹

public final static ImageDescriptor
   IMAGE_CONNECT = getImageDescriptor("s_connec.gif"),
   IMAGE_DISCONNECT = getImageDescriptor("s_discon.gif"),
   IMAGE_DELETE = getImageDescriptor("s_b_dele.gif"),                          ❺
   IMAGE_SCROLL_LOCK = getImageDescriptor("s_s_locl.gif"),
   IMAGE_SCROLL_UNLOCK = getImageDescriptor("s_s_loop.gif"),
   IMAGE_INFO = getImageDescriptor("s_b_info.gif");

private static ImageDescriptor getImageDescriptor(String imgName) {
   final String imageName = imgName;
   return new ImageDescriptor() {
      public ImageData getImageData() {
         return SapIdeUtilImages
            .createImageFromIconsDirectory(                                    ❻
               J2EEConsolePlugin.class.getName(),
               imageName)
            .getImageData();
      }
   };
}

public MainView() {}

public void dispose() {                                                        ❼
   super.dispose();
}

public void createPartControl(Composite parent) {
   MainView.parent = parent;

   table = new Table(parent, SWT.NONE);
   TableColumn dateCol = new TableColumn(table, SWT.LEFT);
   dateCol.setText("Date");
   dateCol.setWidth(120);

   TableColumn messageCol = new TableColumn(table, SWT.LEFT);                  ❽
   messageCol.setText("Message");
   messageCol.setWidth(700);
   table.setHeaderVisible(true);

   table.setBackground(new Color(null, 0, 0, 0));
```

Continues on next page

Figure 13 *continued*

```
      table.setForeground(new Color(null, 255, 255, 255));

      tableViewer = new TableViewer(table, SWT.H_SCROLL | SWT.V_SCROLL);

      makeActions();
      contributeToActionBars();

      connect();
   }

   private void makeActions() {
      connectAction = new Action() {
         public void run() {
            if (connected)
               disconnect();
            else
               connect();
         }
      };
      connectAction.setToolTipText("Connect/Disconnect");
      connectAction.setImageDescriptor(IMAGE_CONNECT);

      deleteAction = new Action() {
         public void run() {
            table.removeAll();
            tableViewer.refresh();
         }
      };
      deleteAction.setToolTipText("Delete");
      deleteAction.setImageDescriptor(IMAGE_DELETE);

      scrollLockAction = new Action() {
         public void run() {
            scrollLocked = !scrollLocked;
            if (scrollLocked)
               scrollLockAction.setImageDescriptor(IMAGE_SCROLL_UNLOCK);
            else
               scrollLockAction.setImageDescriptor(IMAGE_SCROLL_LOCK);
         }
      };
      scrollLockAction.setToolTipText("Scroll Lock/Unlock");
      scrollLockAction.setImageDescriptor(IMAGE_SCROLL_LOCK);

      infoAction = new Action() {
```

❾

Continues on next page

Figure 13 *continued*

```
            public void run() {
               MessageDialog.openInformation(
                  parent.getShell(),
                  "SAP Pro J2EE Console",
                  "If you have any feedback, comments or questions about " +
                  "this plug-in, please contact Sebastien Cherry at " +
                  "sebastien.cherry@netlab.ca.");
            }
         };
         infoAction.setText("Info");
         infoAction.setImageDescriptor(IMAGE_INFO);
      }

      private void contributeToActionBars() {
         IActionBars bars = getViewSite().getActionBars();
         fillLocalPullDown(bars.getMenuManager());
         fillLocalToolBar(bars.getToolBarManager());
      }

      private void fillLocalToolBar(IToolBarManager manager) {
         manager.add(connectAction);
         manager.add(new Separator());
         manager.add(scrollLockAction);
         manager.add(deleteAction);
      }

      private void fillLocalPullDown(IMenuManager manager) {
         manager.add(infoAction);
      }

      public void connect() {
         connected = true;
         connectAction.setImageDescriptor(IMAGE_DISCONNECT);
      }

      public void disconnect() {
         connected = false;
         connectAction.setImageDescriptor(IMAGE_CONNECT);
      }

      public void setFocus() {
         tableViewer.getControl().setFocus();
      }
   }
```

⑩

⑪

⑫

by the wizard and the necessary custom code. (Any code not useful for the example console plug-in has been deleted.) Let's walk through the code section by section:

- Section ❶ declares the MainView class in its package and imports required packages into our coding. The MainView class extends the `ViewPart` class that is the base abstract class of all views in the workbench. This code was generated automatically by the wizard, but the imports needed to be readjusted, as shown in Figure 13.

> ***Tip!***
>
> To easily declare required imports, use the code generation feature in Eclipse for organizing imports. Right-click on your code page, select Source on the context menu, and then select Organize Imports, or simply press Ctrl-Shift-O each time an import must be added.

- Section ❷ declares static attributes for holding required references on different UI widgets. The attribute parent represents the UI composite parent encompassing the view itself. The `tableViewer` and `table` attributes represent the table widgets on which texts will be displayed in the console. These attributes, like several methods in this class, are declared as static since there is always only one instance of the MainView class created by the workbench, even if the view is used in several perspectives.
- Section ❸ keeps a reference on possible actions that could be performed on our view.
- Section ❹ keeps the different states of our actions.
- Section ❺ declares and initiates images required to represent our actions in the UI.
- Section ❻ implements a method that enables us to instantiate a particular image and that is used in the preceding section (i.e., section ❺). The implementation of this method uses available functions in the SAP UI Toolkit and, more specifically, the static method `createImageFromIconsDirectory` of class `SapIdeUtilImages`. This method needs the plug-in ID from which the image should be loaded as well as the image name. In the example, we declared the plug-in ID with the name com.sappro.example.plugin.jconsole.J2EEConsole Plugin; this name can be dynamically retrieved from the class name itself.
- Section ❼ declares the default constructor and the method `dispose` that is called when the plug-in is discarded. This section is generated automatically by the wizard.
- Section ❽ declares a callback method inherited from the superclass `ViewPart`. The method is called by the workbench in order to design and initiate the view. First, the table in which the messages in the console will be displayed is created. Then two columns, one for the date and hour of the message and another for the message itself, are created by passing the reference on the previously instantiated table. Then the background color of the table is customized in black and the foreground color in white to give our console view an air of authenticity. Thereafter, the methods `makeActions` and `contributeToActionBars` are called in order to instantiate possible actions that could be performed on our UI, and to make them appear in the toolbar and in the pull-down menu. Finally, the connection to the J2EE Engine server is performed, or emulated, if proper settings have been maintained in the preference page (which we'll code in Step 2).
- Section ❾ shows the implementation of method `makeActions`, used for creating all actions that will be made available in the main view. Each action instantiation is similar and consists of implementing method `run` of the `Action` abstract class that is called when the action must be performed (when clicking on the corresponding button or selecting the corresponding menu item). Implementations of actions are self explanatory; we won't go into them. Furthermore, action declaration consists of setting a text or a tooltip text as well as a previously instantiated icon to the action

— i.e., `connectAction.setToolTipText("Connect/Disconnect");` sets the tooltip text, whereas `connectAction.setImageDescriptor(IMAGE_CONNECT);` sets the icon for the button.

- Section ❿ includes methods called after the creation of all required actions and consists of linking appropriate actions to the toolbar menu (e.g., `scrollLockAction`) or to the drop-down menu (e.g., `infoAction`).
- Section ⓫ presents `connect` and `disconnect` methods that will be called at different places in the logic of the plug-in. For now, they only consist of keeping the state of the console and displaying the appropriate icon depending on the state, but they will be extended later to launch or stop the console thread.
- Finally, section ⓬ has been generated automatically by the wizard and is used to receive the keyboard focus and be notified of all keyboard events for any feature that may require them. For the example, the table viewer gets the focus, but no features require it since the console displays server messages in read-only mode.

Test and debug the main view

To test our work so far, as explained previously, we need to launch another instance of the Eclipse workbench so that our new plug-in can be loaded and made available in the workbench. Follow the menu path Run → Run As → Run-Time Workbench, which launches another instance of the workbench; once the new instance is loaded, run the new plug-in by following the menu path Window → Show View → Other, and then selecting the plug-in SAP Pro J2EE Console under the SAP Pro Example category in the Show View dialog (see **Figure 14**).

If everything works, the plug-in should launch in a custom view area of the workbench, as shown in **Figure 15**.

To test whether the actions that we implemented perform correctly, click on the buttons in the toolbar and select the actions in the drop-down menu to execute them. For the Connect/Disconnect and Scroll Lock/Unlock actions, you should see the icons toggle back and forth, depending on the state of the action. For the Delete action, you should notice no change in the UI since no message has yet been displayed in the table, and for the Info action, the message dialog should pop up displaying the hard-coded text.

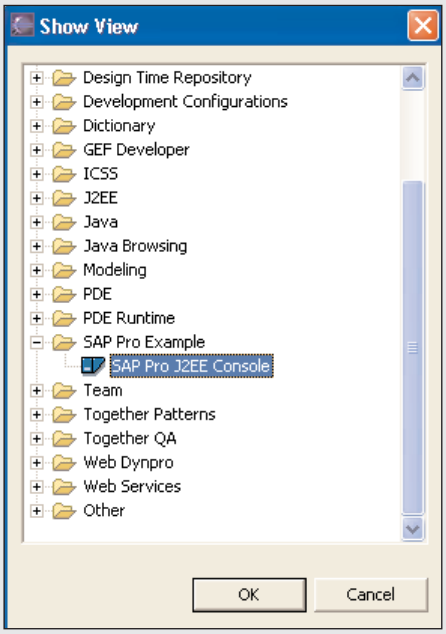

Figure 14 Launching the SAP Pro J2EE Console plug-in

If something goes wrong with the plug-in, or if you want to see what happens in the code during the execution, you can debug the plug-in. Simply launch the plug-in in debug mode in a workbench instance that is separate from the workbench instance where the plug-in is being developed, and insert breakpoints in the code where you want to take back control during execution. To insert a breakpoint, double-click in the left margin next to the line of code where you want the breakpoint. For example, insert a breakpoint next to the first line of code in the method `createPartControl`, which is called when the plug-in is instantiated (see **Figure 16**).

Next, launch the debugger by following the menu path Run → Debug → As Run-Time Workbench. The debugger perspective should appear in the instance

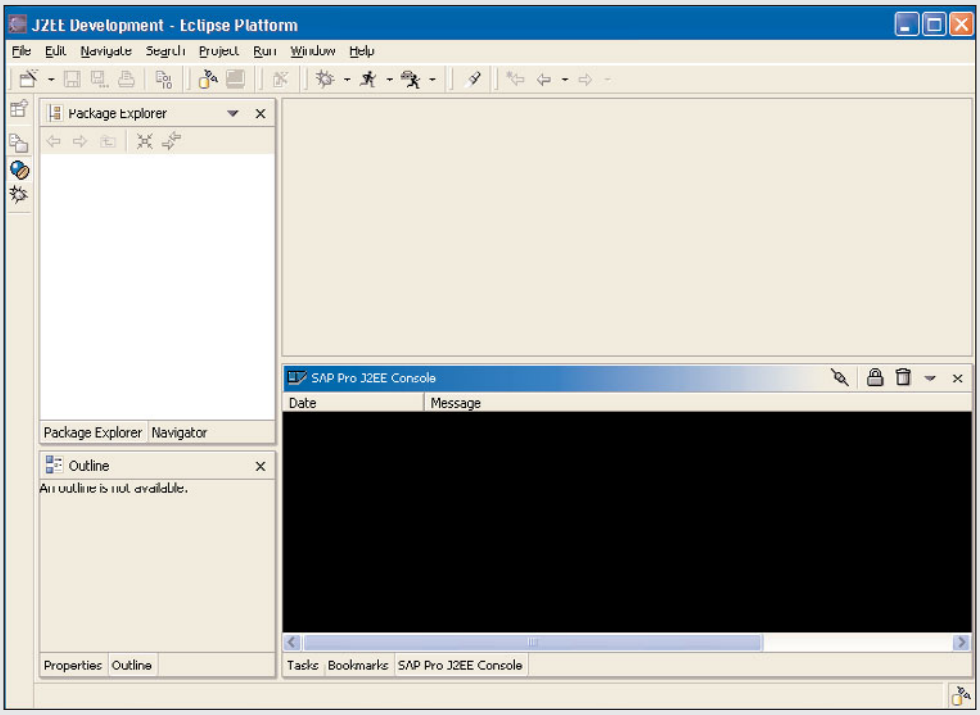

Figure 15 SAP Pro J2EE Console plug-in launched as a custom view in the workbench

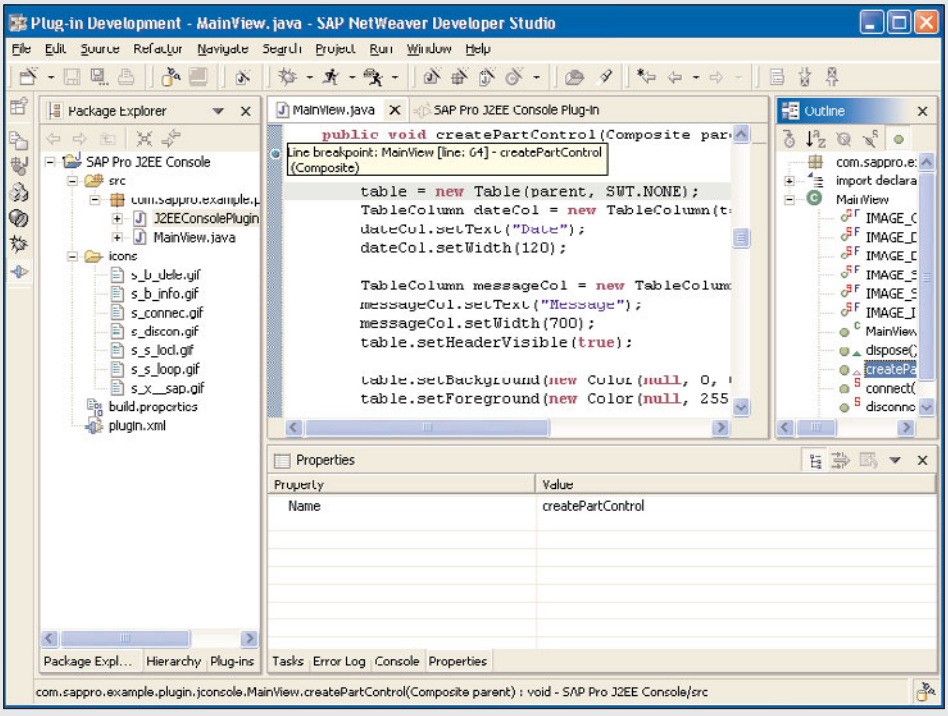

Figure 16 Viewing the code during execution and setting a breakpoint

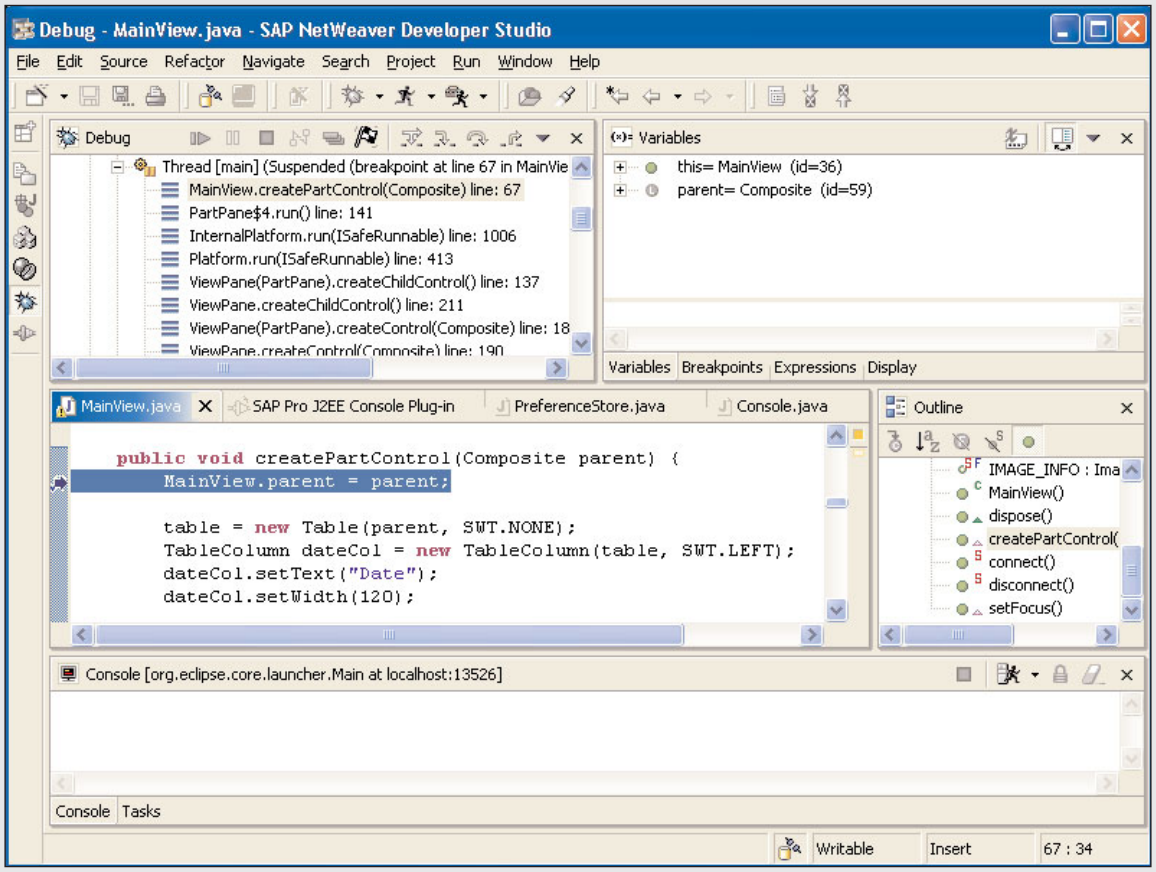

Figure 17 Debugging the plug-in

where the plug-in is being developed while a new instance is launched. The control returns to the first instance as soon as the breakpoint is reached, as shown in **Figure 17**. To step in the code, you can use the available debugging functions provided on the Run menu. Debugging threads are displayed in the upper left view while relevant information such as variable values is displayed in the upper right view. A console displaying the standard outputs of the plug-in itself appears in the lower view.

To ensure that your actions (for instance, the Delete action) are properly called, you can also test them by inserting a breakpoint in their respective implementation methods and clicking on the corresponding button in the instance running the plug-in.

> ***Tip!***
>
> You don't need to restart the debugger when adding new breakpoints in code regions that can be reached by triggering events on the UI.

Next we need to code the preference page that will allows users to collect basic settings needed to run the console, such as the server directory.[15]

[15] In this example, only one setting will be available on the preference page. However, other settings, such as the console message text font and color, could be made available.

Step 2: Code the PreferencePage class

For coding the preference page, start by creating a new class named PreferencePage using the class wizard (follow the menu path File → New → Class). As shown in **Figure 18**, specify the package in which you want to create the new class (in this example, com.sappro.example.plugin.jconsole), and then enter the name of the class and specify its superclass, which in this example must be org.eclipse.ui.internal.dialogs.WorkbenchPreferencePage. Finally, deselect the option for creating the main method — i.e., clear the method stubs checkbox labeled "public static void main(String[] args)" — and click on Finish.

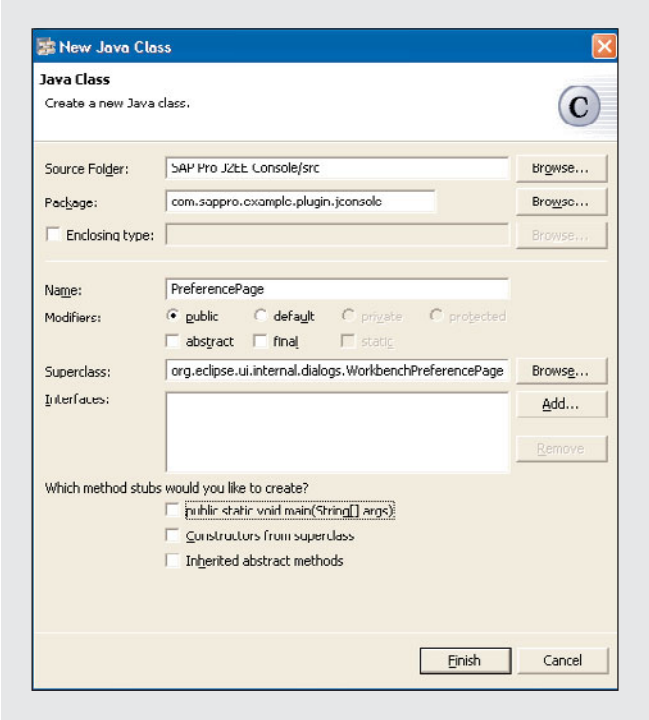

Figure 18 Creating the PreferencePage class

Figure 19 shows the code for the PreferencePage class. This code is a combination of the code automatically generated by the class wizard and code that needs to be entered manually so the page will function as it should. (As with the MainView code, code not needed for the example plug-in has been deleted.)

Let's look at each section to better understand what the code means:

- Section ❶ declares the PreferencePage class as generated by the wizard and as an extension of the `WorkbenchPreferencePage` class, a default implementation of a preference page.
- Section ❷ defines and initiates the preference store, where all preferences are kept and saved in the workbench system files and which will be used to retrieve custom preferences.
- Section ❸ defines attributes used to keep references to our UI elements.

```
package com.sappro.example.plugin.jconsole;

import java.io.*;

import org.eclipse.jface.dialogs.*;
import org.eclipse.jface.preference.*;                    ❶
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.graphics.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.ui.internal.dialogs.*;
```

Figure 19 Coding the PreferencePage class

Continues on next page

Figure 19 *continued*

```
public class PreferencePage extends WorkbenchPreferencePage {

   private final static IPreferenceStore preferenceStore =
      J2EEConsolePlugin.getDefault().getPreferenceStore();                    ❷

   private static Composite parent;
   private static Text serverPathTextField;                                   ❸

   private final static String SERVER_ROOT = "SERVER_ROOT";                   ❹

   public static String getServerRootPreference() {
      return preferenceStore.getString(SERVER_ROOT);                          ❺
   }

   private final static SelectionAdapter browseButtonListener =
      new SelectionAdapter() {
      public void widgetSelected(SelectionEvent e) {
         DirectoryDialog directoryDialog =
            new DirectoryDialog(parent.getShell());
         directoryDialog.setText("Browse to server root");

         String serverRootPath = serverPathTextField.getText();

         if (serverRootPath != null && serverRootPath.length() != 0)
            directoryDialog.setFilterPath(serverRootPath);

         String path = directoryDialog.open();
         if (path != null && path.length() != 0) {                            ❻
            if (new File(path, "apps").exists()
               && new File(path, "log").exists()) {

                  serverPathTextField.setText(path);
            }
            else {
               MessageDialog.openError(
                  parent.getShell(),
                  "J2EE Console",
                  "Invalid server path!");
            }
         }
      }
   };

   protected Control createContents(Composite parent) {
```

Continues on next page

Figure 19 *continued*

```
        PreferencePage.parent = parent;

        Composite serverPathComposite = new Composite(parent, SWT.NONE);
        Label serverPathLabel = new Label(serverPathComposite, SWT.BOLD);
        serverPathLabel.setText("Server Root Path:");
        serverPathLabel.setBounds(10, 10, 100, 20);

        serverPathTextField = new Text(serverPathComposite, SWT.BORDER);
        serverPathTextField.setBackground(new Color(null, 255, 255, 255));
        serverPathTextField.setText(getServerRootPreference());                ❼
        serverPathTextField.setEditable(false);
        serverPathTextField.setBounds(10, 30, 250, 20);

        Button browseButton = new Button(serverPathComposite, SWT.PUSH);
        browseButton.setText("Browse...");
        browseButton.setBounds(280, 30, 75, 20);
        browseButton.addSelectionListener(browseButtonListener);

        return new Composite(parent, SWT.NULL);
    }

    protected void performDefaults() {
        serverPathTextField.setText("");
    }

    public boolean performOk() {
        save();
        return true;
    }

                                                                               ❽
    protected void performApply() {
        save();
    }

    public void save() {
        String path = serverPathTextField.getText();
        if (path != null)
            preferenceStore.setValue(SERVER_ROOT, path);
        else
            preferenceStore.setValue(SERVER_ROOT, "");
    }
}
```

- Section ❹ defines a constant that will be used to map the server root preference in the preference store. It is used, for example, in a public method in section ❺ to enable us to retrieve the server root preference value.
- Section ❻ defines and instantiates a button selection adapter. This adapter will be invoked when the user clicks on the Browse button on the preference page to specify the server root from which messages should be displayed in the console. The implementation consists of the definition of the `widgetSelected` method of the `SelectionAdapter` class in which a directory dialog will be displayed to indicate the server root and that will validate, to some extent, the path indicated. If the path is invalid, a message dialog pop up will inform the user. If the path is valid, it will be inserted in the server root input field designed in the next section.
- Section ❼ defines method `createContents` inherited from the superclass that will be called automatically by the workbench during the creation of the preference page, in the same way that method `createPartControl` (refer back to section ❽ in Figure 13) is called when a view is instantiated. In this section, a label is created for an input field. This input field, as previously mentioned, will be used to indicate the root of the server from which messages should be displayed. However, to reduce chances of invalid inputs, the input field is customized to be non-editable and a browse button enabling a directory dialog will be made available to specify the path. Consequently, a browse button is created and placed next to the input field, and the previously defined button selection adapter is linked to it in order to be notified when the button is clicked.
- Finally, section ❽ includes method redefinition from the superclass, which must be implemented in every preference page for resetting default values when the Restore Defaults button is clicked and applying specified values when the Apply and OK buttons are clicked.

Now, since a preference page constitutes in itself an extension point, and in order to make the workbench load our preference page when it is instantiated, we must declare it in our plugin.xml file. To do this, open the file, and under the Source tab, insert the highlighted lines shown in **Figure 20**.

The `extension` tag indicates a new extension point definition, specified by the `point` attribute, into which underlying plug-ins are plugged. The tag `page` then represents a new preference page declaration. The preference page will be reached under the node indicated by the attribute name in the Preferences window under the Window menu. Finally, the corresponding class of the preference page is indicated by the `class` attribute and a unique ID must be attributed to the page (in the code, `com.sappro.example.plugin.jconsole.PerferencePage`).

> ***Note!***
>
> As with the other IDs defined in this example, we use the name of the corresponding class.

> ***Note!***
>
> Declaring the new preference page extension point in the plug-in provides you greater insight into its structure. However, this could be achieved more easily by using the wizard: Switch to the Extensions tab, and click on Add. In the window that appears, select Extension Templates, and then select the extension point you want. Subsequent steps depend on the type of the extension point selected.

Test and debug the preference page

The procedure for testing the preference page is the same as the procedure for testing the main view.


```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
   id="com.sappro.example.plugin.jconsole.J2EEConsolePlugin"
   name="SAP Pro J2EE Console Plug-in"
   version="1.0.0"
   provider-name="sappro.com"
   class="com.sappro.example.plugin.jconsole.J2EEConsolePlugin">

   <runtime>
      <library name="SAPProJ2EEConsole.jar"/>
   </runtime>
   <requires>
      <import plugin="org.eclipse.core.resources"/>
      <import plugin="org.eclipse.ui"/>
      <import plugin="com.tssap.util"/>
   </requires>

   <extension
         point="org.eclipse.ui.views">
      <category
            name="SAP Pro Example"
            id="example.sappro.com">
      </category>
      <view
            name="SAP Pro J2EE Console"
            icon="icons/s_x__sap.gif"
            category="example.sappro.com"
            class="com.sappro.example.plugin.jconsole.MainView"
            id="com.sappro.example.plugin.jconsole.MainView">
      </view>
   </extension>

   <extension
         point="org.eclipse.ui.preferencePages">
      <page
            name="SAP Pro J2EE Console"
            class="com.sappro.example.plugin.jconsole.PreferencePage"
            id="com.sappro.example.plugin.jconsole.PreferencePage">
      </page>
   </extension>

</plugin>
```

Figure 20 Declaring the preference page as an extension point

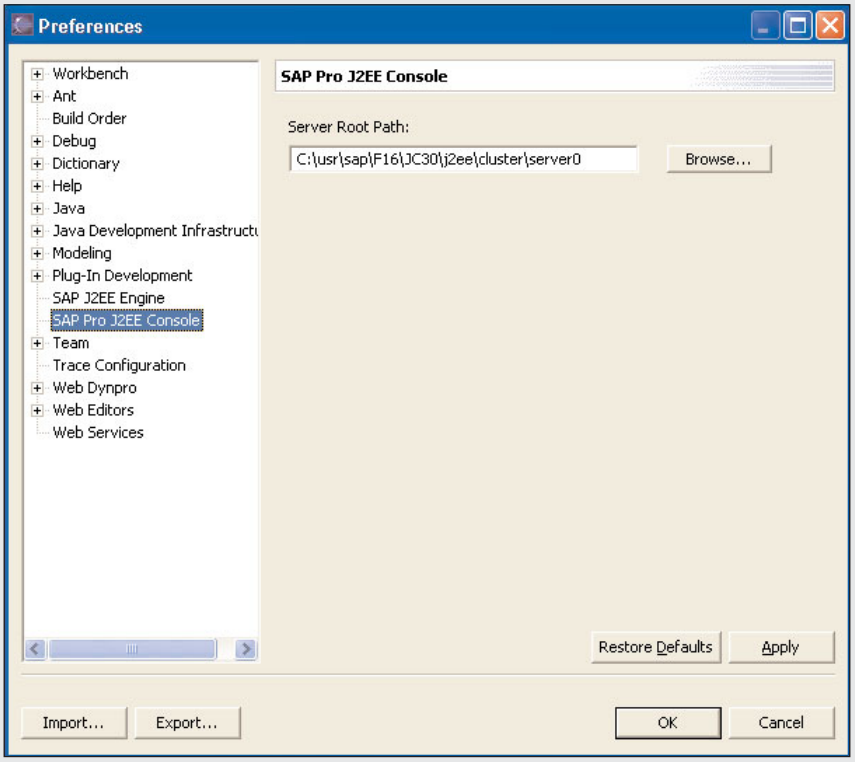

Figure 21 Testing the preference page

Run a new instance of the workbench, and once the main window appears, open the Preferences window from the Window menu. Select SAP Pro J2EE Console in the left pane, and then click on Browse to navigate to the server root (see **Figure 21**).

To get familiar with callback method sequences and check whether proper methods are called when the corresponding buttons are clicked on the page, you can set some breakpoints in the code and launch the workbench in debug mode. You can also test whether the path validation condition works as expected by entering invalid paths.

> ***Note!***
>
> The server path should follow the pattern /usr/sap/<SID>/JC<J2EE instance>/j2ee/cluster/server<server node number>.

Step 3: Code the Console class

In order to understand how we will code our console logic, let's revisit a few concepts. As previously mentioned, with the advent of SAP NetWeaver, the new J2EE Engine does not run as a standard process. Instead it runs as a service, making the visualization of standard outputs impossible. However, when properly configured, the J2EE Engine can generate standard outputs, errors, and exception stack traces in a trace file (this step is explained in the upcoming "Test and debug the console" section). Consequently, the logic of our console emulator should open the file in which the standard outputs are written and show new messages in the view table. The console should run in a thread to avoid blocking in-progress operations in the workbench, and in addition, read the trace


```
public void appendTextToTable(String text) {
   TableItem tableItem = new TableItem(table, SWT.NONE);
   tableItem.setText(0, dateFormat.format(new Date()));
   tableItem.setText(1, text);
   if (!scrollLocked)
      table.showItem(tableItem);

   tableViewer.refresh();
}
```

Figure 22 Coding to display new messages retrieved from the server in the console

file and refresh the content of the table every second to emulate a real-time behavior.

Moreover, trace files on the J2EE Engine are written in a loop starting from defaultTrace.0.trc. When the file reaches a fixed size, messages are written to defaultTrace.1.trc, then defaultTrace.2.trc, and so on. When the maximum number of files is reached, traces are written back to the initial defaultTrace.0.trc file, cleared from the old messages. This implies that on each refresh loop in our thread, we should check whether a more recent file exists due to a trace file change by the server.

Now, keeping these concepts in mind, and apart from the coding of the Console class itself, two communications must be performed:

- First, from the MainView class to the Console class in order to start or stop the console, depending on the action performed on the UI. This will be done once the console is created.
- Second, from the console to the main view for two reasons: to display in the main view new messages retrieved from the server, and to acknowledge that the console thread is disconnected due to an inappropriate configuration setting or an unexpected exception during its execution.

These communications will be performed by implementing a method in the MainView class to `appendTextToTable`, and another method that actually exists in the MainView class (refer back to section ⓫ in Figure 13) that disconnects the console and maintains the appropriate state of the console while displaying the appropriate icon on the button.

So, to implement the first method needed for the second communication process, add the code, as shown in **Figure 22**, in the MainView class.

As its name suggests, the function `appendTextToTable`, which is called by the console, will append a text line to the table. If the scrollbar isn't locked, it will force the new inserted text to be displayed in the view.

> ***Note!***
>
> Normally, in a layered architecture, you must avoid bottom-up communication. However, in our example, since the main view represents our UI and the console could be considered the model, we should avoid tying the console to the main view by referring the former directly to the latter. In situations like these, mechanisms such as the observer pattern must be implemented to avoid this dependency. However, no other UI technology will be used in our case, and to keep our example as simple as possible, we will directly call methods on the main view rather than implementing this kind of pattern.

To this new code, add the following required constant to class MainView in order to define the date format used for displaying the date and time of the message:

```
private final static SimpleDateFormat
      dateFormat = new
      SimpleDateFormat("MMM.dd
      hh:mm:ss");
```

We are now ready to code the Console class. Use the class wizard to create a new class named Console in the package com.sappro.example.plugin.jconsole that extends the class java.lang.Thread, just as you did for the PreferencePage class. Once this is done, you can begin to edit the code generated by the wizard and to enter the code needed for the console (see **Figure 23**). (As with the MainView and PreferencePage classes, code not needed for the example console plug-in has been deleted.)

Let's look at each section to better understand what the code means:

- Section ❶ declares the Console class itself as generated by the wizard.
- Section ❷ declares an attribute that will keep a reference to the MainView class in order to inform it of any new text to display or of a potential disconnection, as explained previously.
- Section ❸ declares attributes that will keep references to the current opened trace file and a file buffer, keeping in memory the number of the last line read from the file.
- Section ❹ declares a `boolean` attribute that will let you know whether the last line was parsed correctly or not for a specific algorithm technicality (an explanation of which goes beyond the scope of this article).
- Section ❺ constitutes the constructor of the Console class that enables us to keep the reference to the calling `MainView` instance.
- Section ❻ represents the main thread loop, refreshing the `MainView` instance every second with new texts retrieved in the trace files. The loop continues endlessly as long as a trace file can be read. The core of the loop has been implemented as another `Runnable` (for technical reasons that are beyond the scope of this article). However, in this new thread, another loop takes place as long as a line can be read in the current trace file. Then the line is parsed to extract the message text from it because messages logged in the J2EE Engine trace files are specially formatted to be read by the internal log viewer. Finally, a call is transmitted to the `MainView` instance in order to display the message. At the end of the outer loop, the method `disconnect` to the `MainView` instance is called to ensure consistency between the state of the console and the UI.
- Section ❼ shows the method called in the main loop condition. The method is used to make sure that the most recent file is read because a trace file change by the J2EE Engine may occur at any time. It returns a positive boolean value to indicate that a valid trace file is currently opened, and a negative value otherwise. The implementation calls another method in order to retrieve the most recently modified file. If a file other than the current file is found, the file switch occurs.
- Section ❽ represents the implementation of the method evoked in the section before and used for retrieving the most recently modified file. The implementation of it retrieves the server root specified in the preference page because it may change at any time, and loops through all the files in the log subfolder of the specified directory to search for the most recent modified file.
- Section ❾ consists of methods (i.e., `closeFileInputStreams` and `openFileInputStreams`) also used in the method of section ❶, and enables the closing and opening of a trace file. It should be noted that if the `currentFile` attribute was previously null, this is the first time that we will get in the loop and we must skip all previously logged lines in the new opened trace file as we do with the loop.
- Finally, section ❿ is the method used to parse and extract the relevant message text from the formatted message retrieved in the trace file. It is also used for filtering and returning only messages


```
package com.sappro.example.plugin.jconsole;

import java.io.*;                                                    ❶

import org.eclipse.swt.widgets.*;

public class Console extends Thread {

   private MainView mainView;                                        ❷

   private File currentFile;

   private FileReader fileReader = null;                             ❸
   private LineNumberReader lineNumberReader = null;

   private boolean lastLineParsed = false;                           ❹

   public Console(MainView mainView) { this.mainView = mainView; }   ❺

   public void run() {

      try {
         while (switchToLastModifiedFile()) {

            Display.getDefault().syncExec(new Runnable() {
               public void run() {
                  String line = null;
                  try {
                     do {
                        line = lineNumberReader.readLine();
                        if (line != null) {
                           String text = parseTextLine(line);          ❻
                           if (text != null && text.length() > 0)
                              mainView.appendTextToTable(text);
                        }
                     }
                     while (line != null);
                  }
                  catch (Exception ex) {}
               }
            });
            Thread.sleep(1000);
         }
      }
      catch (Exception ex) {}

      mainView.disconnect();
   }
```

Continues on next page

Figure 23 Coding the Console class

Figure 23 *continued*

```
private boolean switchToLastModifiedFile() {
   try {
      File lastModifiedFile = getLastModifiedFile();
      if (lastModifiedFile != null) {

         if (currentFile == null
            || !lastModifiedFile.getAbsolutePath().equals(
               this.currentFile.getAbsolutePath())) {

            closeFileInputStreams();                                    ❼

            openFileInputStreams(lastModifiedFile);
         }

         return true;
      }
   }
   catch (IOException ioex) {
      closeFileInputStreams();
   }

   return false;
}
```

```
private File getLastModifiedFile() {

   String rootPath = PreferencePage.getServerRootPreference();

   if (rootPath == null || rootPath.length() == 0)
      return null;

   rootPath = rootPath + File.separator + "log";

   File rootFile = new File(rootPath);
   if (!rootFile.exists() || rootFile.isFile())
      return null;                                                      ❽

   long lastDate = 0;

   File lastModifiedPath = null;

   File[] childFiles = rootFile.listFiles();
   if (childFiles != null) {
      for (int i = 0; i < childFiles.length; i++) {
         if (childFiles[i].isFile() &&
            childFiles[i].getName().matches("defaultTrace.*.trc")) {

            if (lastDate <= childFiles[i].lastModified()) {
```

Continues on next page

Figure 23 *continued*

```
                lastDate = childFiles[i].lastModified();
                lastModifiedPath = childFiles[i];
             }
          }
       }
    }

    return lastModifiedPath;
}

private void closeFileInputStreams() {
    try {
       if (lineNumberReader != null)
          lineNumberReader.close();
    }
    catch (IOException ioex) {}

    lineNumberReader = null;

    try {
       if (lineNumberReader != null)
          lineNumberReader.close();
    }
    catch (IOException ioex) {}

    fileReader = null;                                                    ❾
}

private void openFileInputStreams(File fileToOpen)
    throws IOException {
    fileReader = new FileReader(fileToOpen);
    lineNumberReader = new LineNumberReader(fileReader);

    if(this.currentFile == null)
       while (lineNumberReader.readLine() != null);

    this.currentFile = fileToOpen;
}

private String parseTextLine(String line) {
    String text = line;

    int index = -1;                                                       ❿

    if (line.startsWith("#")) {
       if ((line.indexOf("#System.out#") != -1
          || line.indexOf("#System.err#") != -1)
          && (index = line.lastIndexOf("#Plain###")) != -1)
```

Continues on next page

Figure 23 *continued*

```
            lastLineParsed = true;
         else {
            lastLineParsed = false;
            return null;
         }
      }
      else if (!lastLineParsed)
         return null;

      if (index != -1)
         text = line.substring(index + 9);

      if ((index = text.lastIndexOf("#")) != -1)
         text = text.substring(0, index);

      return text.replaceAll("\t", "        ");
   }
}
```

from the standard output from the messages logged by the SAP Logging API. This is a very specific algorithm, the detailed description of which is beyond the scope of this article.

Now that the console is coded, you need to plug it into the UI. To achieve this, you need to insert a few lines of code in the MainView class.

First, as shown below, you need to declare a static attribute referring to the `Console` instance:

```
private static Console console;
```

Then, you need to insert the two highlighted lines of code below, for properly disposing of the console in the `dispose` method of the view:

```
public void dispose() {
   if (console != null && console.isAlive())
      console.interrupt();
   super.dispose();
}
```

Finally, in the `connect` and `disconnect` methods, you need to insert the highlighted lines of code shown in **Figure 24** to instantiate the console and start or interrupt it, and to release it, depending on the case.

Test and debug the console

Your console is now complete. However, before you test it, you must perform a preliminary step on the J2EE Engine to configure it to log standard outputs, errors, and exceptions in the trace file. To do this, you need to start your J2EE Engine Visual Administrator and log onto your J2EE Engine server.

> ***Note!***
>
> Launch the J2EE Engine Visual Administrator by running the go script, located under the directory that follows the pattern /usr/sap/<SID>/JC<J2EE instance>/j2ee/admin.


```
public void connect() {
   connected = true;

   connectAction.setImageDescriptor(IMAGE_DISCONNECT);

   if (console == null || !console.isAlive()) {
      console = new Console(this);
      console.start();
   }
}

public void disconnect() {
   connected = false;

   connectAction.setImageDescriptor(IMAGE_CONNECT);

   if (console != null && console.isAlive()) {
      console.interrupt();
      console = null;
   }
}
```

Figure 24 Coding the connect and disconnect methods

> ***Note!***
>
> To log on to the J2EE Engine server via the Visual Administrator, you must specify an administrator user name and password.

> ***Note!***
>
> Since this configuration can have a direct impact on system performance, this should only be performed on local test systems — not in the production system!

Once you log on to the specific J2EE Engine server, open the server node in the left pane and then the Services branch, and click on the Log Configurator service. In the central pane, switch to the Locations tab and open the System node. There, select the err node and change the severity to All in the drop-down menu on the right pane, as shown in **Figure 25** on the next page. Repeat the same operation for the out node and save your settings.

Once this is done, and in order to test the console plug-in with the Console class completed, you need to output some messages in the standard output from an existing Java Web application. You could also choose to create a test application and deploy it to the J2EE Engine server, as explained in Karl Kessler's article (refer back to footnote 1 on page 99). For example, **Figure 26** on the next page is a simple JSP page, logging some messages and exceptions, that could be deployed in a Web application on the J2EE Engine.

Now run the plug-in as explained earlier and invoke your own Web applications or a new application containing the JSP page shown in Figure 26 to log some messages in the standard output. You should

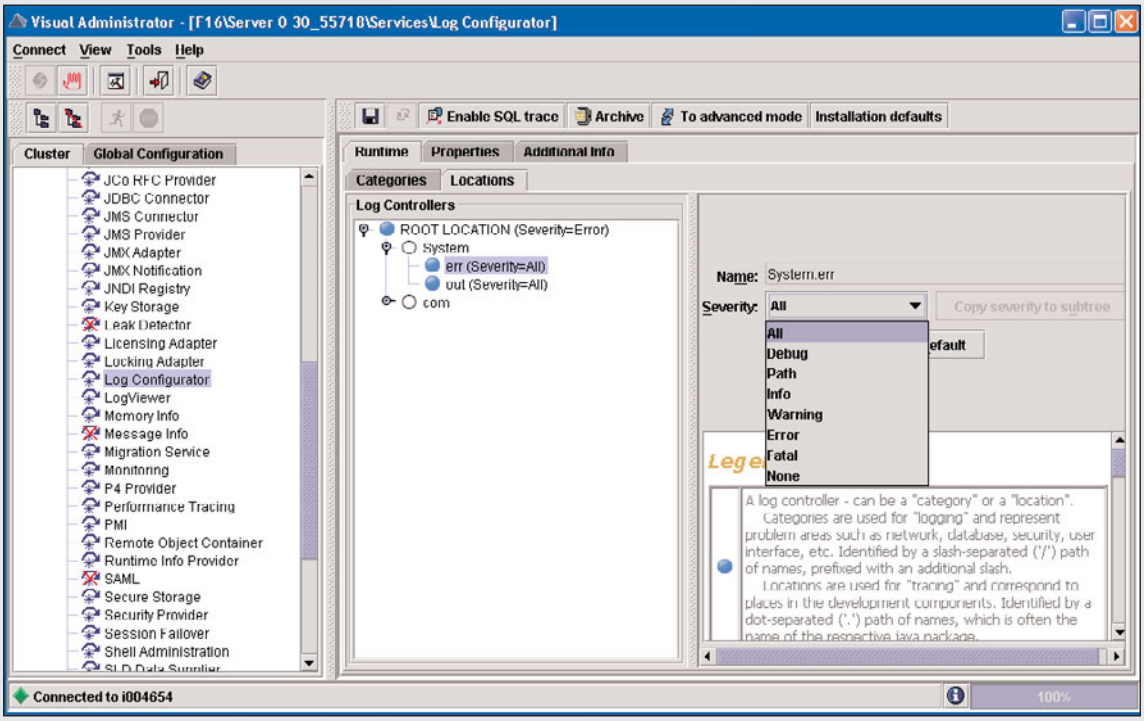

Figure 25 Configuring the J2EE Engine to log standard outputs, errors, and exceptions in the trace file

```
<%@ page language="java" %>
<%
System.out.println("This a standard output test!");

System.err.println("This an error output test!");

try {
   throw new Exception("This is a exception test");
}
catch(Exception myex) { myex.printStackTrace(); }
%>

<html>
   <head>
      <title>
         Title
      </title>
   </head>
   <body>
      <h1>
         Generated JSP Test
      </h1>
   </body>
</html>
```

Figure 26 Example of standard output messages

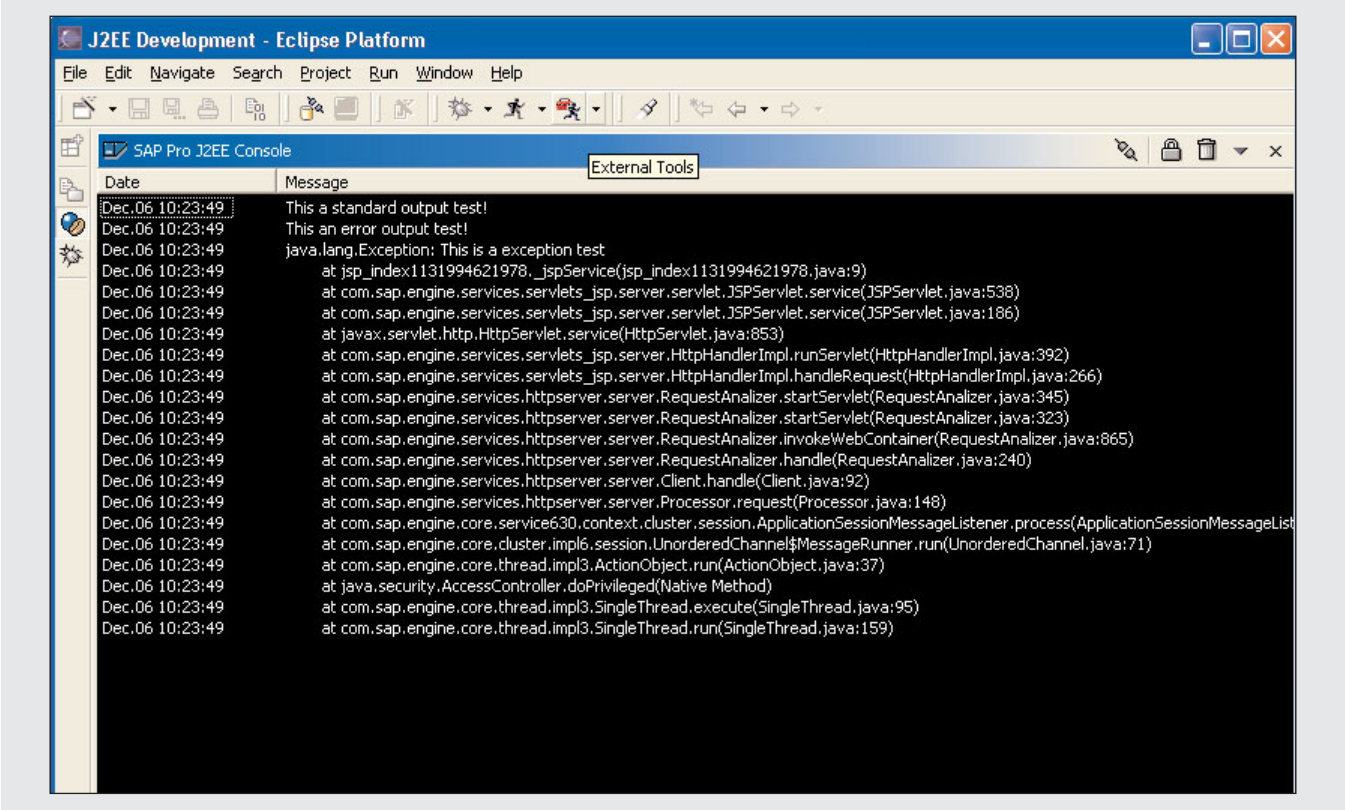

Figure 27 Displaying log messages via the new console

now see the results on your new console, similar to those shown in **Figure 27**.

Deploying the plug-in

You can share[16] a new plug-in by building it in a deployable file, such as a ZIP file. To build the deployable ZIP file, first open the build.properties file in your project and insert the line highlighted below:

```
source.SAPProJ2EEConsole.jar = src/
bin.includes = plugin.xml,\
               *.jar,\
               icons/,\
               SAPProJ2EEConsole.jar
```

This line indicates that the icons folder and its content must be included in the ZIP file.

Now, follow the menu path File → Export. In the dialog that opens (see **Figure 28** on the next page), select the plug-ins and fragments that you want to deploy, and click on Next. On the next page, check the plug-in ID and version, choose to export as a single deployable ZIP file, specify the file path and name, and click on Finish.

You should now have a ZIP file containing all the necessary files for running the plug-in on other SAP NetWeaver Developer Studio installations.[17] To test whether it works, install it on your own SAP NetWeaver Developer Studio environment! Simply extract the plug-in ZIP file under the SAP NetWeaver

[16] Update sites can be used to share plug-ins. For more information about this feature, go to http://help.eclipse.org/.

[17] In a Microsoft Windows installation, the SAP NetWeaver Developer Studio directory is usually found under C:\Program Files\SAP\IDE\IDE70 or C:\Program Files\SAP\JDT depending on the version installed.

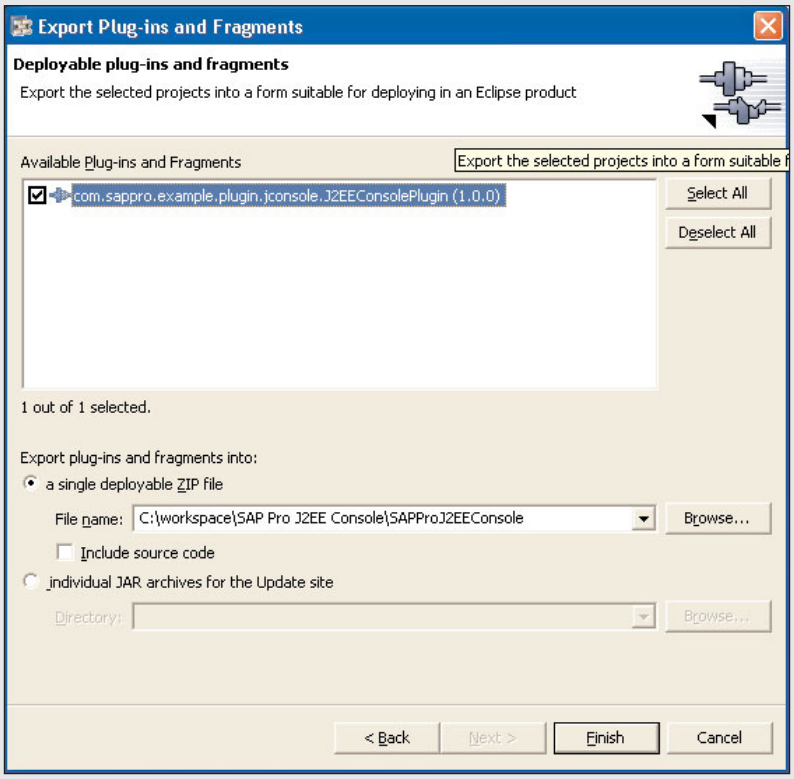

Figure 28 Deploying the console plug-in via a ZIP file

Developer Studio installation directory containing the Eclipse subdirectory and restart your SAP NetWeaver Developer Studio instance. Then, open the plug-in from the Show View dialog and you're on your way.

Conclusion

The purpose of this article has been to show you just how easy it is to develop, build, and deploy plug-ins to enhance your SAP NetWeaver Developer Studio environment. Thanks to the openness, flexibility, and extensibility of the Eclipse platform, as well as its convenient plug-in development environment, tailoring your own development environment is now child's play.

Furthermore, this article aimed to make project managers as well as developers aware of the very real ROI that can be gained from tapping into the full potential of the SAP NetWeaver Developer Studio environment. You need to spend only a small amount of time and effort to develop useful tools that can enable all developers to deliver better-quality software more efficiently.

