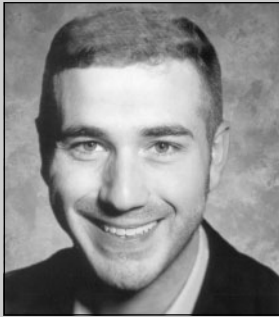


# Build Custom Java iViews for SAP Data Using Eclipse: A Guide for Developers and Implementation Teams

Carl Vieregger



*Carl Vieregger is a senior consultant with IBM Business Consulting Services. Since 1998 he has been a member of the Global SAP Centre of Expertise (GCOE) based in Walldorf, Germany. After starting as a certified ABAP developer and business workflow specialist, his current focus is on portals and portlet application development.*

*(complete bio appears on page 136)*

iViews provide integrated views of business information in the mySAP Enterprise Portal (EP), and SAP offers a large selection of pre-built iViews, organized into role-based “business packages.” When you import these business packages from the iViewStudio, the technical implementation of the iViews is already done for you. Indeed, portal administrators can set up a standard business package without any programming language experience.

While SAP makes an extensive range of content available in its pre-built business packages, these solutions cannot possibly address all of your organization’s unique requirements. Sooner or later, portal users will begin to look for specific business content that can only be provided by custom-built iViews. In this article, I guide developers and implementation teams through the tasks involved in developing a customer-specific iView that displays data from an SAP system:

1. Custom iViews in Java are built within a software construct called a “project.” In the first section, I will provide an overview of this development structure and introduce the different types of files in a Java-based iView.
2. While you can use any integrated development environment (IDE) to build a Java iView, this article employs Eclipse, and I will describe its installation and configuration in the second section. I’ll also show you how to set up Eclipse for your own iView development projects.
3. In the third section, I will look at a custom-built example iView that retrieves and renders SAP data. Then I’ll walk you through the

process of building and deploying the example, which I call the “HelloSAP” iView. My assumption throughout is that you have a basic understanding of Java, but experienced developers will also benefit from the hands-on exploration of SAP’s API. The HelloSAP example is based on version 5.0 of the Enterprise Portal, using service pack 4 (SP4).

4. Once the HelloSAP iView is running in the SAP Portal Development Kit (PDK), I will demonstrate how to enhance the example with language-independent features, a process known as “internationalization.”

This article contains all of the resources you need to build the HelloSAP example — your first Java iView. While the example is simple enough to help you get started, it still provides valuable functionality for an EP production system. You can download the HelloSAP iView detailed in this article from the “Download Files” section at [www.SAPpro.com](http://www.SAPpro.com) and import it directly into your own PDK.

### ✓ *Note!*

*Before working through the example iView in this article, you must first install the PDK and configure it for single sign-on (SSO) access to an SAP system. I describe this installation and configuration process in the November/December 2002 issue of this publication. If you have already deployed the PDK, you can test your SSO settings for SAP with the JCo Connection Pool iView, which is located on the PDK’s Examples tab.*

## Development Projects for Custom Java iViews

The HelloSAP example iView is shown in **Figure 1**. Like all iViews, it is a web application component

that runs within the mySAP Enterprise Portal (EP). Users can access and interact with the iView through the EP’s Internet browser interface. Java-based iViews are built on the multi-tier distributed architecture of the J2EE platform (see the sidebar on the next page), and they are often referred to as “portal components,” particularly while still under construction.

At the development level, portal components are assembled within a programming construct called a “project.” A development project in Java is almost always file-based, consisting of an assortment of root folders and subfolders. All of the source and build files, as well as any additional resources and connectors, are included within the project’s file structure.

**Figure 2** provides a look at the root folders of the development project for my HelloSAP portal component. As seen via Eclipse, the full name of the project is *com.ibm.gcoe.HelloSAP*.

**Figure 1**      *The HelloSAP Example iView for Displaying SAP Data*

Code	Company Name
0001	SAP A.G.
1000	Giotto LTD
2000	IDES UK
2100	IDES Portugal
2200	IDES France
2300	IDES España
2400	IDES Italia
2500	IDES Netherlands
3000	IDES US INC
4000	IDES Canada
5000	IDES Japan
6000	IDES México, S.A. de C.V.
F100	Bankhaus Frankfurt
F300	Liberty Bank
R100	IDES Retail GmbH
R300	IDES Retail INC US
S300	IDES Services
ZP10	SAP PORTALS Company Code

Page 1/1

## Java 2 Platform, Enterprise Edition (J2EE)

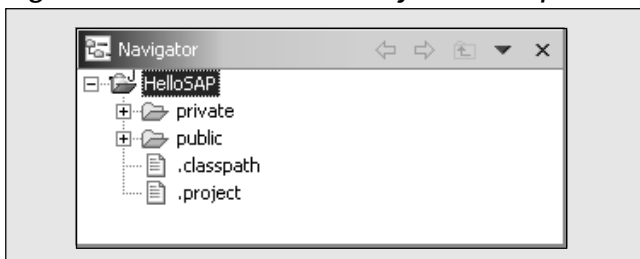
SAP Java iViews are web application components on the J2EE platform. The distributed application model of the J2EE platform consists of three separate tiers: client, web, and backend. Portal users access iViews on the client tier, which is usually an Internet browser. SAP portal components are built on the web tier with JavaServer Pages (JSP) technology and Java servlets, both of which are responsible for the dynamic rendering of the iView. At runtime, JSPs and servlets are transformed into the necessary markup language of the client. The connector architecture of the J2EE platform supports access from the web tier to the backend tier, enabled via plug-in adapters for data sources.

### ✓ Tip

*While not the case for the HelloSAP example outlined here, you should note that it is possible for more than one portal component (i.e., more than one iView) to be maintained in a single development project.*

The J2EE platform not only supports the development of portal components in project file structures, it also enables their packaging and deployment. Packaging consists of building the separate files of a development project into an application archive, which SAP calls a portal archive (PAR) file. Deployment, then, is the process of copying and extracting the packaged PAR file to the EP's operational environment, the Portal Content Directory (PCD).

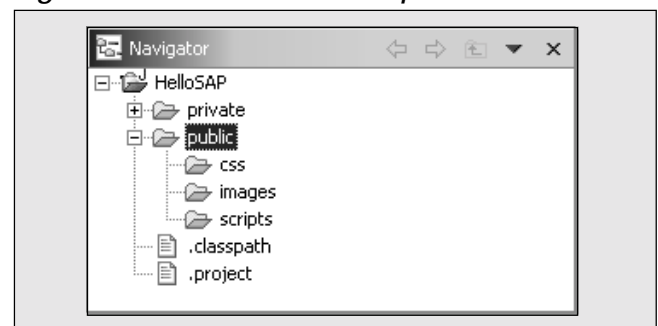
**Figure 2** The HelloSAP Project in Eclipse



Before I discuss the build and deployment process in further detail, I want to take a step back and look at the file structure for the HelloSAP portal component and introduce the individual files in the development project. As you can see in Figure 2, my HelloSAP project consists of a *private* and a *public* folder, plus two additional files in the project root directory, *.classpath* and *.project*. These two files are described in more detail in the sidebar “Project Build Files” on the next page.

Project files in the *public* folder are available to the client at runtime, while those in the *private* folder are not. As shown in **Figure 3**, the types of files that are saved in the *public* folder include Cascading Style Sheets (*css*), graphic files such as images and icons (*images*), and files containing JavaScript code (*scripts*). My HelloSAP project does not have any files in the *public* folder.

**Figure 3** Contents of the “public” Folder



## Project Build Files

The `.classpath` and `.project` files contain build- and deployment-related information for the portal component project. These particular build files are specific to Eclipse, and they are generated automatically when the development project is first created. Both files are XML-based.

The `.project` file contains the name of the development project (`com.ibm.gcoe.HelloSAP`) and some additional details about the project's build and run structure. The second file, `.classpath`, includes path references for the Java libraries that are required to compile the Java source. When the project is packaged and the PAR is created, the Java compiler (Javac) uses the `.classpath` file to locate the necessary libraries. If Javac can't find a required library, the build process will fail.

By way of comparison, JBuilder saves the location of required libraries via its "Project Properties" option; if you use the open source build-and-deploy tool Apache Ant, the Java libraries are maintained in the `libraries.properties` file. (Ant enables you to deploy PAR files without the use of a commercial IDE.)

Within the `private` folder are four subfolders (see **Figure 4**):

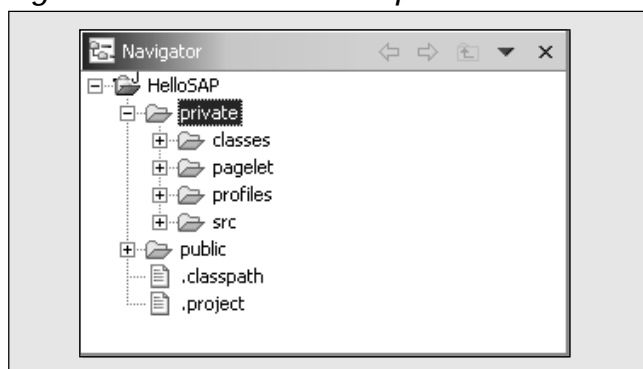
- `classes`
- `pagelet`
- `profiles`
- `src`

Let's take a closer look at each of these subfolders in turn.

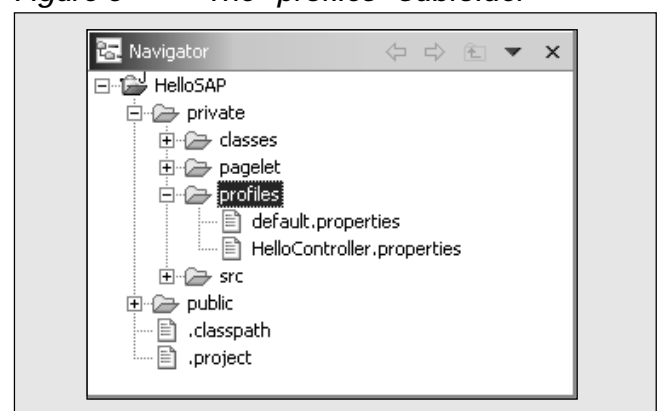
## The "profiles" Subfolder

Component profiles provide additional runtime properties for portal components and are saved in the `profiles` subfolder (see **Figure 5**). SAP portal components can use a number of standard properties that control how the iView is displayed. You can also define new properties to support custom runtime tasks and user personalization. A component profile is a plain-text file saved with the `.properties` extension, and each property in the profile is maintained as a name/value pair.

**Figure 4** Contents of the "private" Folder



**Figure 5** The "profiles" Subfolder



One profile, named *default.properties*, is required in every portal archive. When a development project contains multiple portal components, the properties in this default profile apply to all of them.

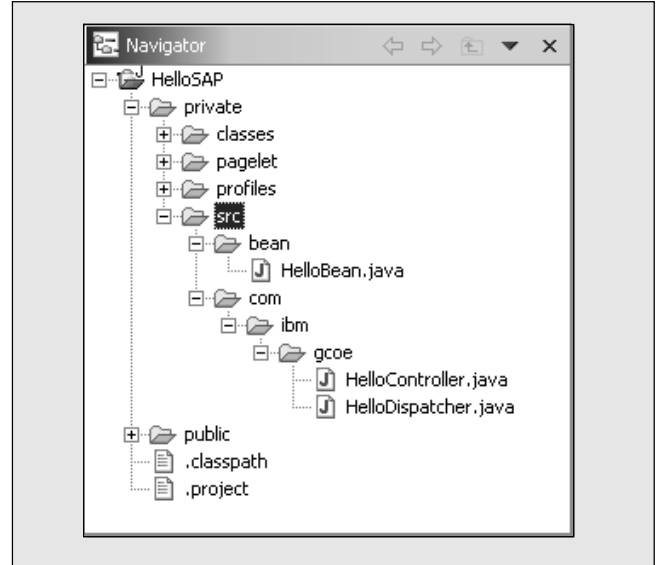
In addition to the *default.properties* file, a unique profile must be maintained for each portal component in the project. I refer to these files as “implementation profiles,” and the properties in them are portal-component-specific. An implementation profile generally shares the same name as the primary class in the portal component.<sup>1</sup> Since my HelloSAP project consists of a single portal component, it has just one implementation profile: *HelloController.properties*.

At runtime, profiles can be thought of as the starting point of the portal component. Indeed, the generated URI<sup>2</sup> for a Java iView points directly to the implementation profile of the deployed portal component. If you are familiar with ITS-based Internet applications, profiles in an SAP portal component can be loosely compared to the service (SRVC) file used in Internet Services, IACs, and MiniApps.<sup>3</sup>

### The “src” Subfolder

Two types of Java source files are saved in the *src* subfolder (see **Figure 6**): component application files (*com*) and JavaBeans components (*bean*).<sup>4</sup> Component application files contain the application logic of the portal component. These files work to make the connection to the backend system, to retrieve the required data, and to manage the dialog flow of the iView. JavaBeans components, on the other hand,

Figure 6 The “src” Subfolder



provide a container infrastructure for persisting data during the browser session of the iView. Typically, component application files will save data to a bean so that a JSP page can access and render it at a later time.

In the *src* subfolder, Java source files are organized in a directory structure that is dictated by the package directive for their class. For example, the package for the implementation class in my HelloSAP project is defined as:

```
package com.ibm.gcoe;
```

Accordingly, the source file for the implementation class is saved in the folder structure *private\src\com\ibm\gcoe*.

Component application files and JavaBeans components both end with the *.java* extension. At build time, you choose whether or not to include them in the portal archive.<sup>5</sup> My HelloSAP example has two component application files: *HelloController.java* (the implementation class) and *HelloDispatcher.java*. The name of the JavaBeans component in the example is *HelloBean.java*.

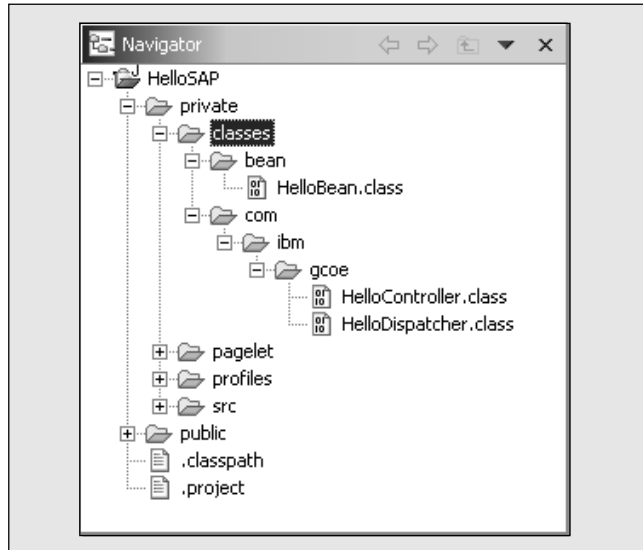
<sup>1</sup> The primary class in a portal component is the first Java class that is instantiated at runtime. It is sometimes referred to as the “implementation class.” In the example, this class is *HelloController*.

<sup>2</sup> The uniform resource identifier (URI) is also known as the web component request path.

<sup>3</sup> For more on IACs and MiniApps, see the articles “No Service Reps Required! Learn How to Supply 24x7 Sales Order Information Online with the Sales Order Status IAC” and “Ready to Build Your First MiniApp? It’s Quick and Easy with the ABAP Workbench!” in the May/June 2001 issue of this publication.

<sup>4</sup> The files in the resource bundle that support internationalization are also maintained in the *src* subfolder. I will demonstrate how to use internationalization features in the final section of this article.

<sup>5</sup> For its part, SAP does not include the source code in the PAR files of its standard iViews.

**Figure 7** The “classes” Subfolder

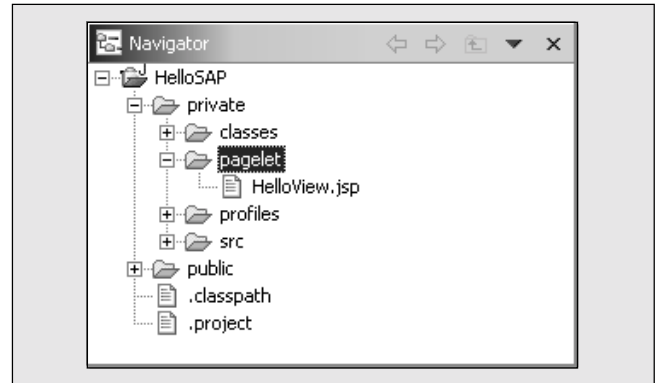
### The “classes” Subfolder

Closely related to the Java source files are the compiled bytecode files of the development project. Within Eclipse, Javac (the Java compiler) compiles the source code each time the project is built, and the resulting CLASS files are saved in the *classes* subfolder (see **Figure 7**) with the *.class* extension. The directory structure in *classes*, just as for *src*, is based on the package directives of the Java source files. The CLASS files in the example are *HelloController.class*, *HelloDispatcher.class*, and *HelloBean.class*.

### The “pagelet” Subfolder

The HelloSAP example detailed in this article is rendered in the EP via JSP pages. In a portal component development project, the JSP pages are saved (with the file extension *.jsp*, naturally) in the *pagelet* subfolder (see **Figure 8**). The JSP page in the *pagelet* subfolder of the example is named *HelloView.jsp*.

JSP technology allows you to integrate Java with HTML, where the Java code is included on an HTML page via a special tag called a “scriptlet.” Combining

**Figure 8** The “pagelet” Subfolder

scriptlets with standard HTML, though, can quickly lead to unmanageable code, because it mixes server-side application logic with client-side rendering.

To simplify the development of JSP pages for the EP, SAP developed a tag library called HTML Business (HTMLB) for Java. HTMLB is a set of web controls that support the use of HTML-like forms and layers, as well as GUI elements such as text fields and pushbuttons. HTMLB also provides support for event handling (for example, what the application should do when the portal user fills a text field and selects a pushbutton).<sup>6</sup>

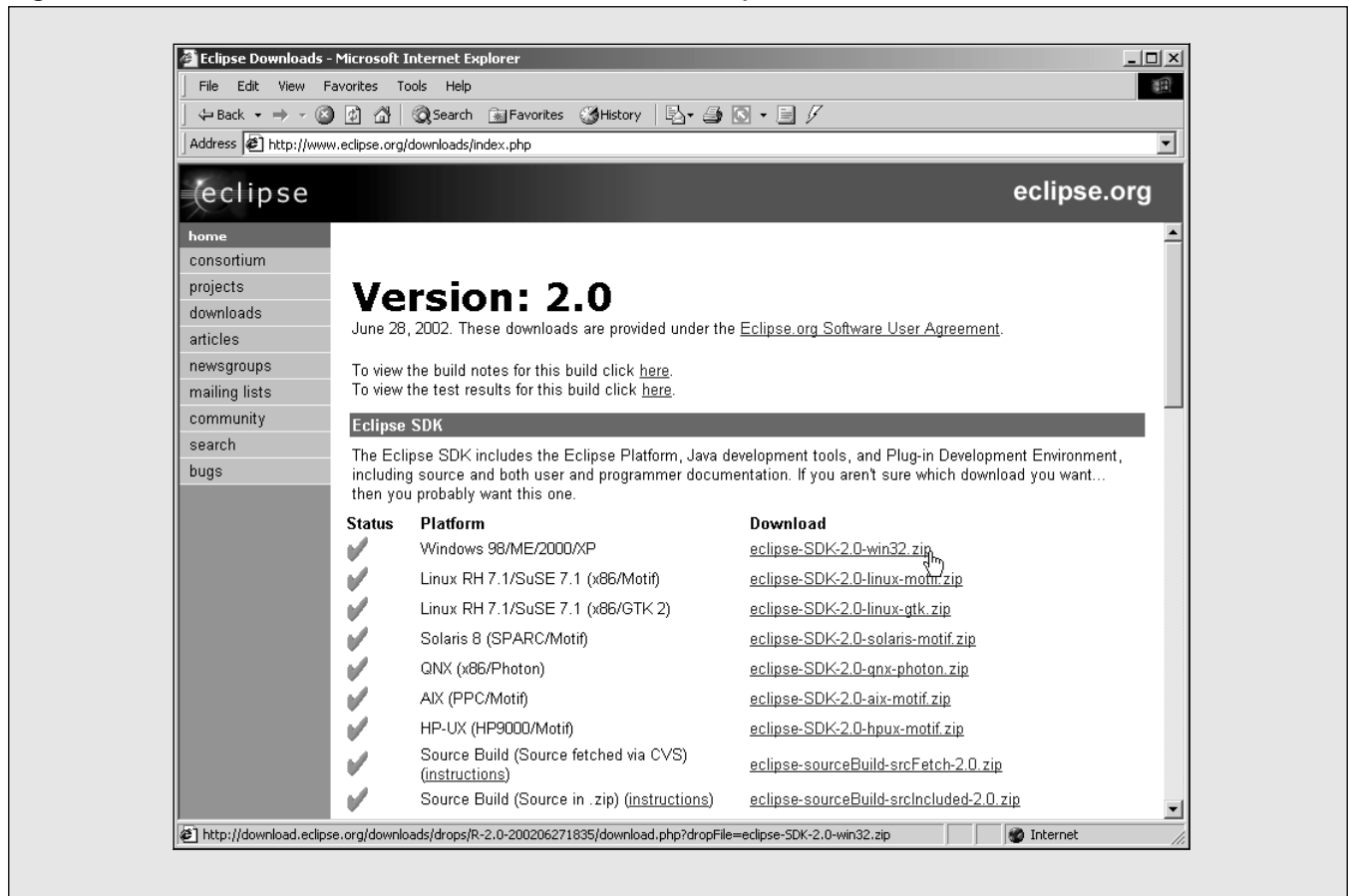
## Installation and Configuration of Eclipse

In the previous section, I introduced SAP portal components as enterprise applications based on the J2EE platform. I also walked through the development project structure behind the HelloSAP iView. To support the construction of portal component projects, SAP provides a plug-in for Eclipse, an open source IDE, and extensible development platform. While you can use any IDE for iView development, SAP is planning to provide continued support only for Eclipse. Before working further with the HelloSAP example,

<sup>6</sup> The HTMLB library can also be used with BSP pages in SAP for web application development. For more detail on this, see the article “Build More Powerful Web Applications in Less Time with BSP Extensions and the MVC Model” on page 3 of this issue.

Figure 9

## Download the Eclipse SDK



then, I will describe the details of how to install Eclipse and configure it with the SAP plug-in.

### The Eclipse SDK

According to a technical white paper on its project web site, Eclipse is “an IDE for anything, and for nothing in particular.”<sup>7</sup> Essentially, Eclipse is an open tool that can be extended with additional features and new functionality, which are integrated into its runtime platform via a plug-in architecture.

A standard plug-in that is delivered with the Eclipse SDK, the Java development tools (JDT),

provides all the features of a robust Java IDE for the Eclipse Platform. Other plug-ins, such as editors for JSP pages and XML files, can also be added to Eclipse’s feature set. SAP provides an Eclipse plug-in to extend the platform’s support for portal components in Java. In conjunction with the PDK, Eclipse offers a comprehensive development environment for building, deploying, and testing Java iViews.

### Download and Install the Eclipse SDK

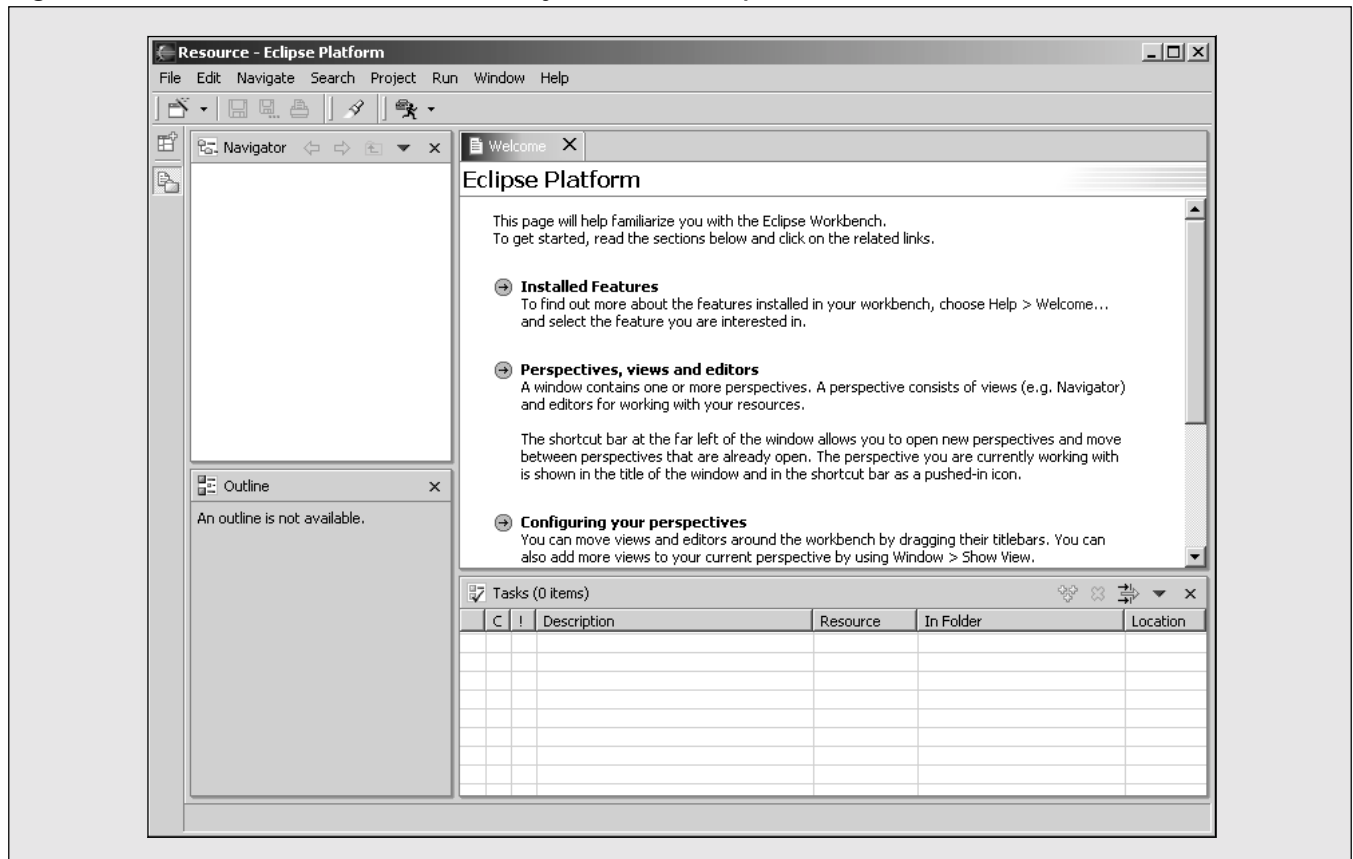
Eclipse 2.0 is available for download under the *Eclipse.org Software User Agreement*. While 2.0 is not the most recent release of Eclipse, it is the version for which SAP’s plug-ins are presently supported. Make sure to download the complete SDK version for Windows (see **Figure 9**).

<sup>7</sup> This white paper is a good introduction to Eclipse, and is available at [www.eclipse.org/whitepapers/eclipse-overview.pdf](http://www.eclipse.org/whitepapers/eclipse-overview.pdf).



Figure 10

Default Layout of the Eclipse Workbench

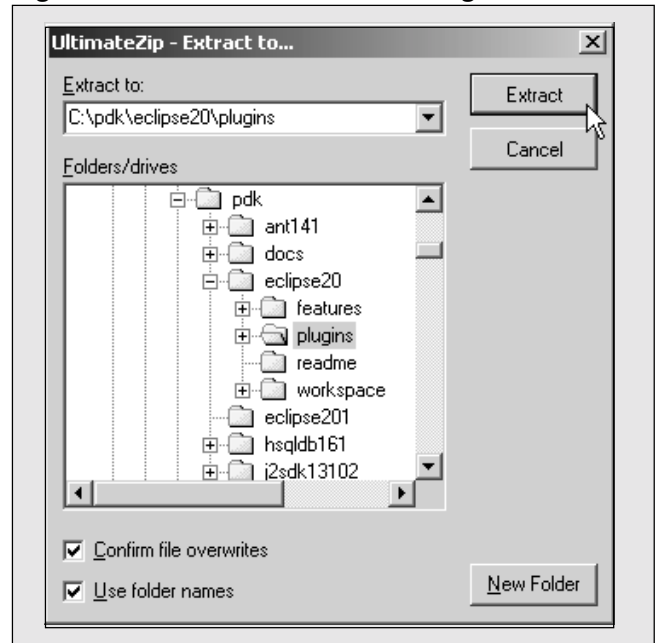


Once the download is complete, simply extract the file to install the application. (I extracted Eclipse to `C:\pdk\` and changed the name of its root folder to `eclipse20`.) To complete the installation automatically, you need to execute `eclipse.exe` in the root directory of the extraction. The default layout of the Eclipse Workbench is shown in **Figure 10**. Behind the layout, you can change the default runtime of Eclipse. For details on changing the default layout, see the sidebar “Change the Eclipse Java Runtime Environment (JRE)” on the next page.

### Customize Eclipse for SAP Portal Component Development

You can download SAP’s Eclipse plug-in (`SAPPDKEclipsePlugins.zip`) from the iViewStudio Java DevZone at [www.iviewstudio.com](http://www.iviewstudio.com). To install the plug-in, first close Eclipse. Now simply extract

Figure 11 Extract the SAP Plug-In

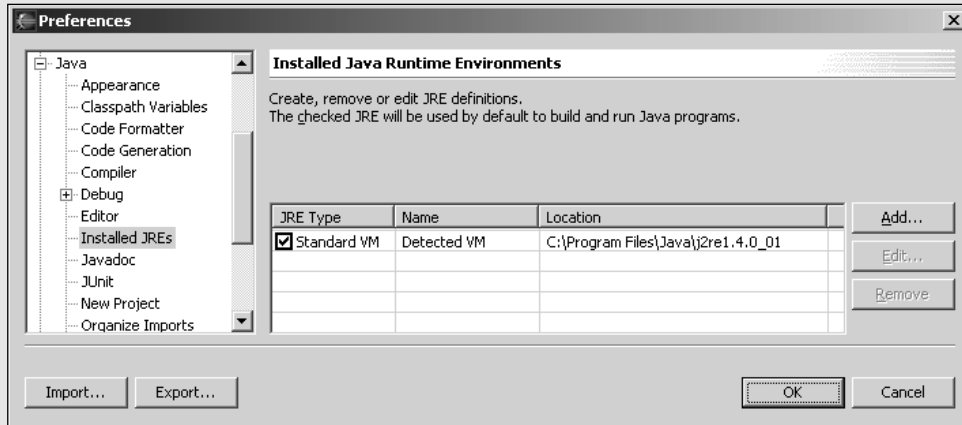




## Change the Eclipse Java Runtime Environment (JRE)

In order to mirror the actual EP environment as closely as possible, you can ensure that Eclipse's default JRE is the same version used for your own Portal Development Kit (PDK) deployment.

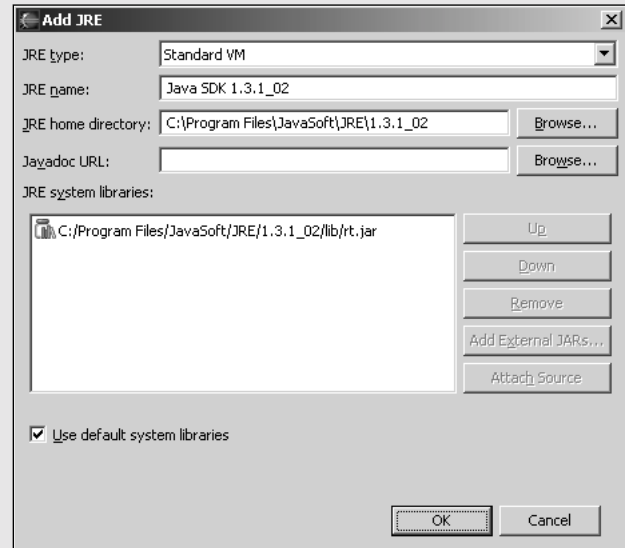
To change the default JRE, go to *Window* → *Preferences*. In the tree structure under *Java*, choose *Installed JREs* to view the detected runtime environment, shown in the screenshot below.



Include another JRE in the list with the *Add* pushbutton. In the pop-up dialog that appears (see the screenshot to the right), name the JRE and browse to its home directory.

Note that the home directory is not where you installed the Java SDK, but rather it is typically under *Program Files\JavaSoft\JRE*. Select the folder of the version you want to include, and the corresponding *rt.jar* library should appear in the box for *JRE system libraries*. Select *OK* to add the JRE, and then choose it from the list below:

JRE Type	Name	Location
<input type="checkbox"/> Standard VM	Detected VM	C:\Program Files\Java\j2re1.4.0_01
<input checked="" type="checkbox"/> Standard VM	Java SDK 1.3.1_02	C:\Program Files\JavaSoft\JRE\1.3.1_02
<input type="checkbox"/>		
<input type="checkbox"/>		

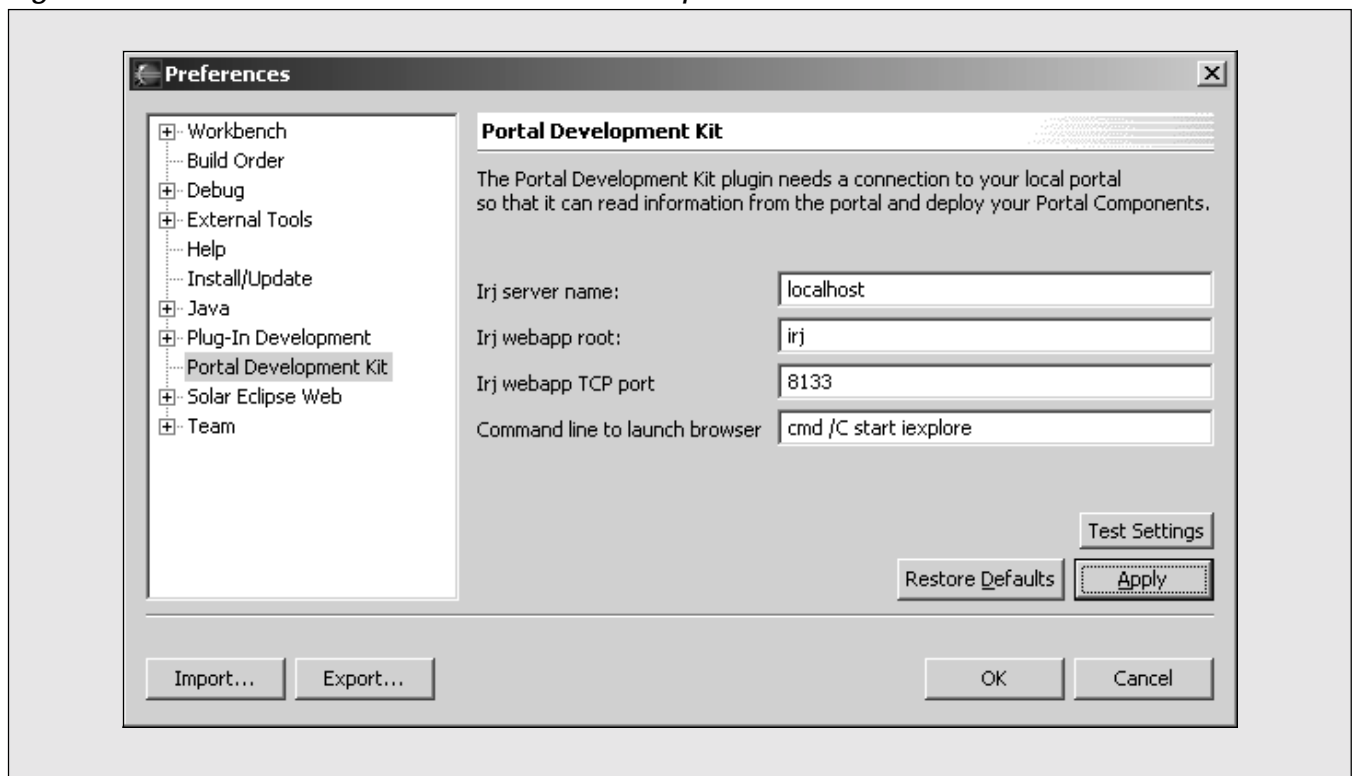


the ZIP file to the *plugins* subfolder under the Eclipse root directory. As you can see in **Figure 11**, the extract destination for my environment is *C:\pdk\eclipse20\plugins*.

Now open Eclipse again. Before you can make use of the new features for portal component development, you need to maintain the connection parameters to your PDK platform. Choose

Figure 12

## Maintain the Eclipse Preferences



Window → Preferences from the Eclipse menu.  
In the dialog that appears, choose *Portal Development Kit* from the tree list on the left (see **Figure 12**).

The default settings in the dialog will connect to the default location of the PDK in Tomcat<sup>8</sup>: *localhost* at port 8080. Since I have changed the port where my computer listens for Tomcat, I must enter this modified value (8133) in the *Irj webapp TCP port* field. I can also insert the name of my computer in place of *localhost*. The other default settings should be okay, and you can select the *Apply* pushbutton to verify them.

If the verification is not successful, you will get a message similar to that depicted in **Figure 13**. Note that Tomcat must be running for the verification to succeed.

<sup>8</sup> For more on installing and configuring Tomcat for use with the PDK, see my previous article in the November/December 2002 issue of this publication.

### ✓ Tip

You can also choose the *Test Settings* pushbutton, which verifies the connection settings before launching the PDK in a new browser window. In an offline development environment, a pop-up may prompt you to either work offline or connect to the Internet.

Figure 13 Failed Verification Message

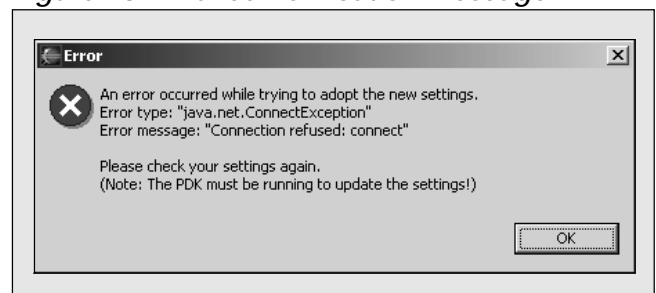
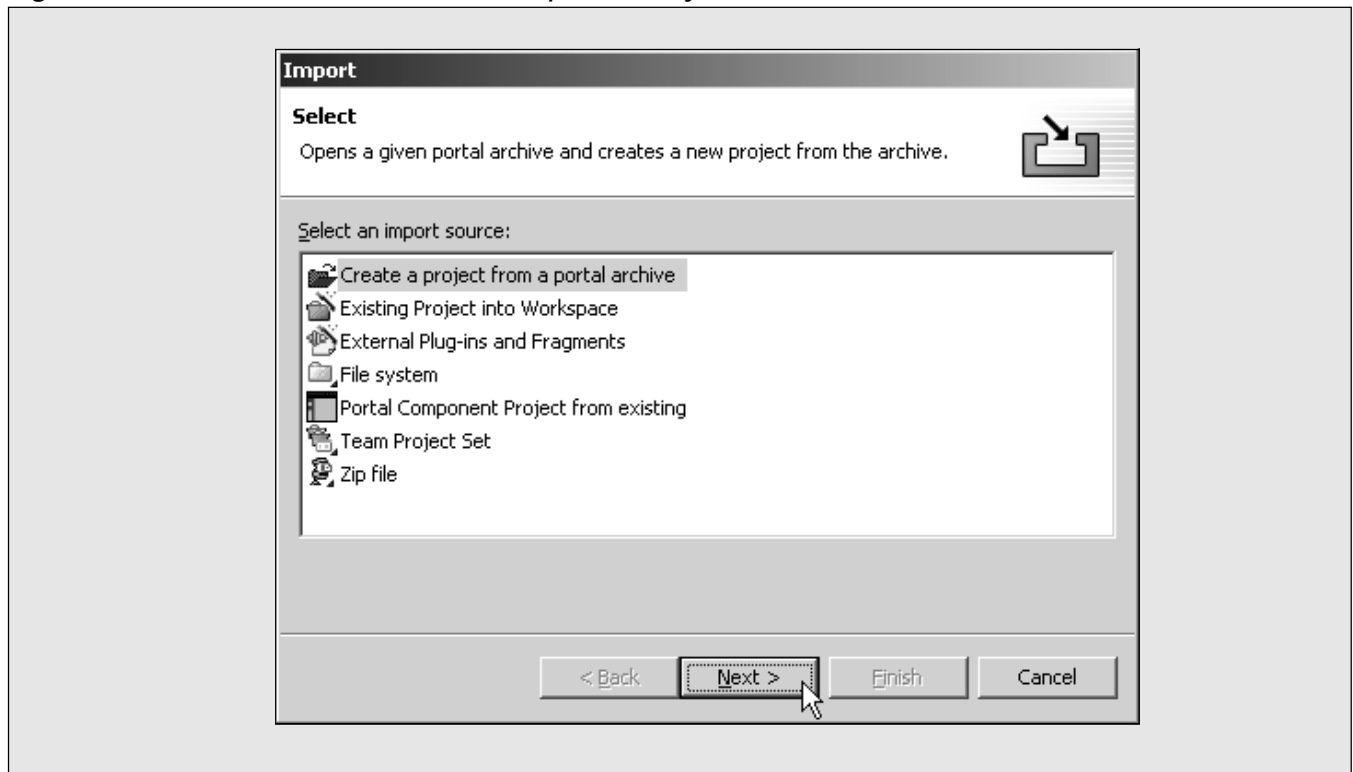


Figure 14

## Create a Development Project from a Portal Archive



A successful connection will enable the direct deployment of portal archive files from Eclipse to the PDK.<sup>9</sup> Other features of the SAP plug-in include wizards for building portal component development projects, together with templates for setting up the individual component application files. The SAP plug-in also enables you to import existing PAR files as development projects, which I will demonstrate next.

Before continuing with this article, if you haven't already done so, take a moment to download the PAR file for the HelloSAP example iView (*com.ibm.gcoe.HelloSAP.par*) from the download page at [www.SAPpro.com](http://www.SAPpro.com). Save the PAR file to your local machine.

### ✓ Note!

The SAP plug-in for iView development is only one of hundreds of plug-ins that are available for the Eclipse Platform. A good place to search for other plug-ins is at the unofficial “plug-in portal” at <http://eclipse-plugins.2y.net>.

In addition to the SAP plug-in for portal components, I use an open source plug-in called SolarEclipse, which supplies additional editor formatting for web application development. It provides support for HTML, JSP, and XML files. You can download the SolarEclipse plug-in at <http://sourceforge.net/projects/solareclipse>.

<sup>9</sup> The SAP plug-in for Eclipse can only connect with the PDK SP4 or later. If you are using a previous PDK release, you can still use Eclipse to build your project and make the portal archive. From within the PDK, you can upload and deploy the PAR file with the *Archive Uploader* component on the *DevTools* page.

Now choose *File* → *Import...* from the Eclipse menu. In the Import wizard dialog that appears (see **Figure 14**), make sure that the option *Create a*

Figure 15

## Import the Portal Archive

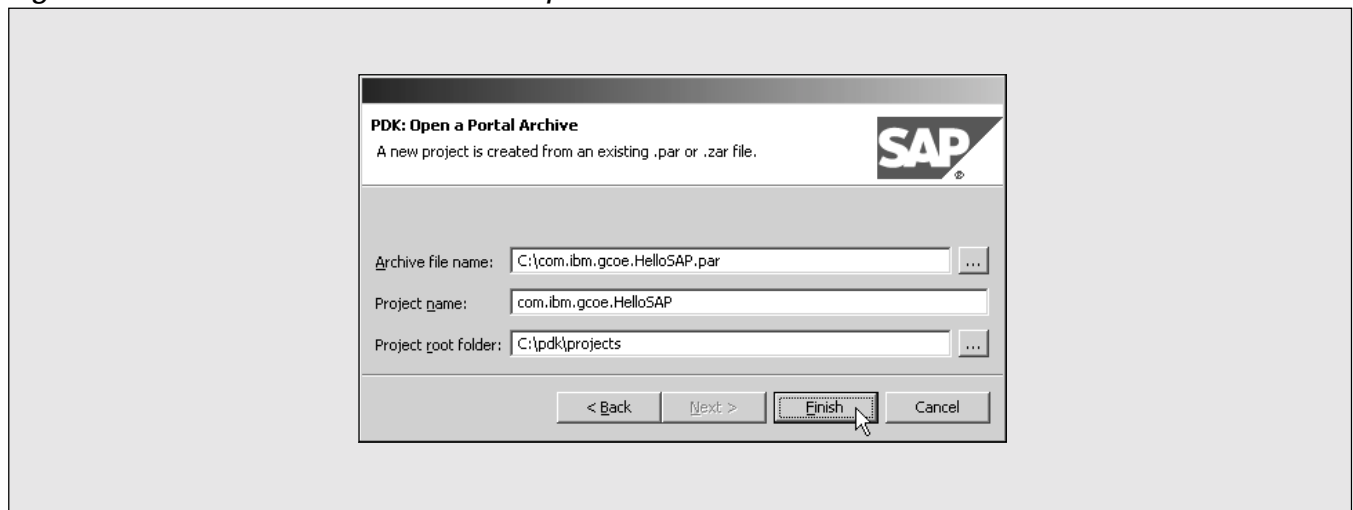
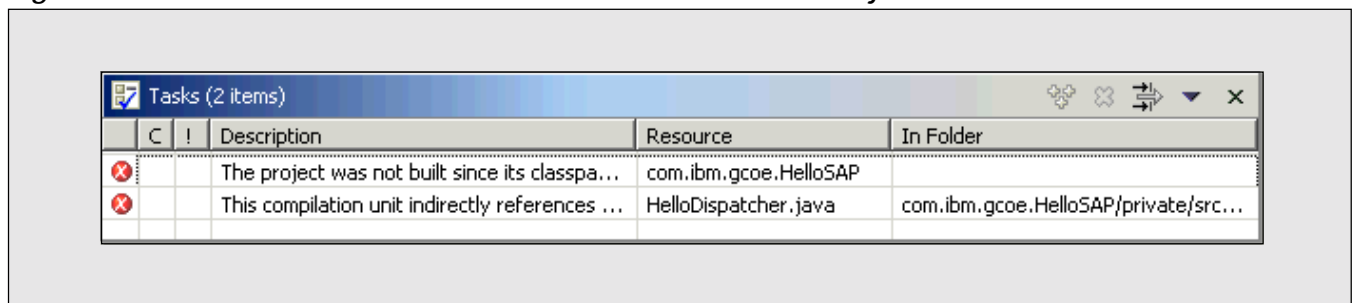


Figure 16

## Initial Errors in the HelloSAP Project



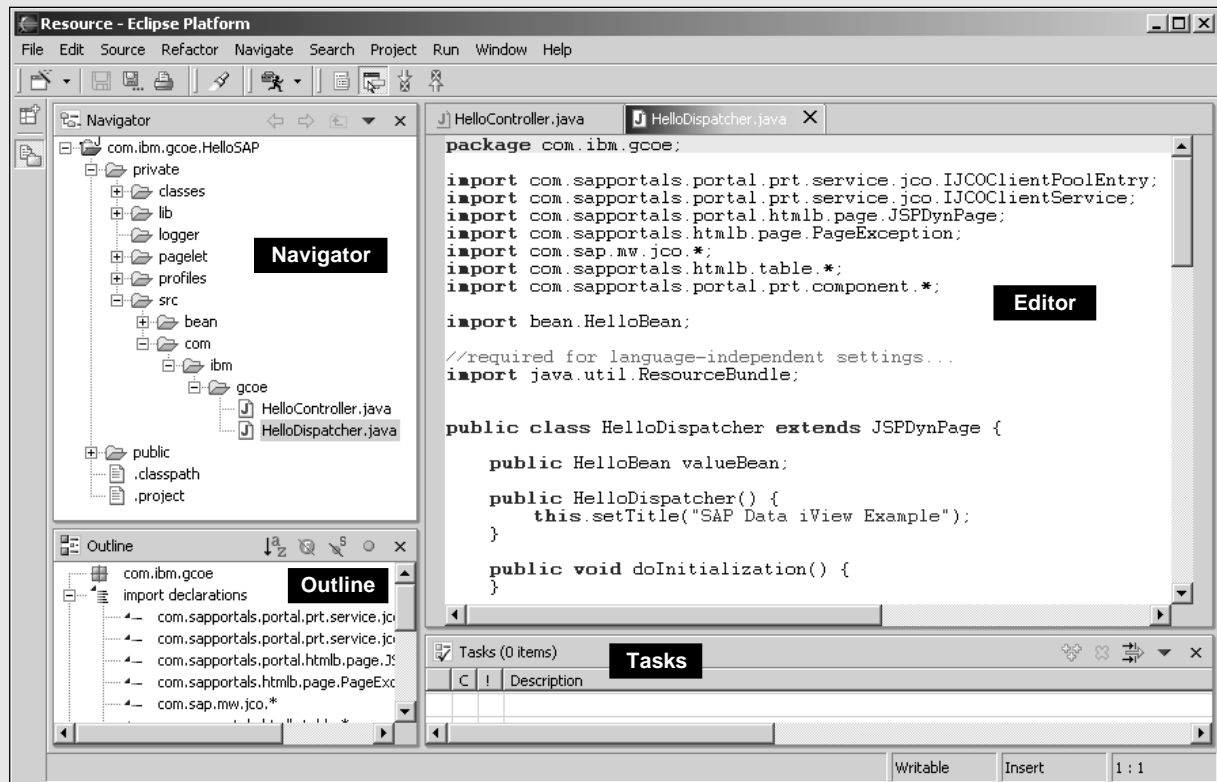
project from a portal archive is highlighted and select the *Next* pushbutton.

In the second step of the wizard, shown in **Figure 15**, choose the browse pushbutton (...) for the *Archive file name* field. Navigate to the *com.ibm.gcoe.HelloSAP.par* file and select it. Provide a name for the project (mine is named *com.ibm.gcoe.HelloSAP*) and set the project root folder (for me, *C:\pdk\projects*). Select the *Finish* pushbutton to import the PAR, and the HelloSAP project will be created in the Navigator view of the Eclipse Workbench. The sidebar on the next page provides a short tour of the Eclipse Workbench user interface using the imported project.

After you import the PAR file, your project may display two errors in the Tasks view of the Eclipse Workbench, as shown in **Figure 16**. This occurs because the *.classpath* file is missing the path to a resource library for the SAP Java Connector (JCo). (Refer back to the sidebar “Project Build Files” on page 106 for a further description of the *.classpath* file.) To resolve the build error, you need to manually provide this information to the project. Open the *.classpath* file by double-clicking on it in the Navigator view. It should contain a number of *<classpathentry>* attributes for the different resource libraries, but a reference to the *sapjco.jar* library is missing. If you are using Tomcat 3.3.1 (as in my example), *sapjco.jar* is located at *tomcat\lib\apps*.

## The Eclipse Workbench User Interface

The layout of the default Eclipse Workbench consists of four panes, or views (see the screenshot below): Navigator, Editor, Tasks, and Outline.



In the top left corner is the **Navigator** view, which I have already used in the article to introduce the development project file structure. To the right of the Navigator is the **Editor** view, where each file, regardless of file type, is opened and displayed. Go to the `private\src\com\ibm\gcoe` folder in the Navigator and open the two Java source files (`HelloController.java` and `HelloDispatcher.java`) by double-clicking on them. In the screenshot above you can see the code for `HelloDispatcher.java`. When multiple files are open simultaneously, you can navigate between them using the corresponding title bars at the top of the Editor view.

Below the Editor view is the **Tasks** view, where you can maintain a task list to keep track of activities. All problems or syntax errors in the project will be listed in this view as well. The fourth view in the default layout, the **Outline** view under the Navigator, displays the structure of the active file in the Editor area. If the active file is a Java source file, like `HelloDispatcher.java`, the Outline will show the structure of the complete package, including class definitions, variable declarations, and methods.

Together, the organized views of the Eclipse Workbench user interface are referred to as a "perspective." The default layout is set to the *Resource* perspective. Other perspectives include *Java*, *Debug*, and *CVS* (Concurrent Versions System). You can learn more about the Eclipse Workbench and the features of the other perspectives under *Help* → *Help Contents*.

For a 4.x release of Tomcat, the missing library is under *tomcat\common\lib*. Add a new attribute with the appropriate value, for example:

```
<classpathentry kind="lib"
  path="C:/pdk/tomcat331/lib/
  apps/sapjco.jar"/>
```

Save the project. It should now build successfully.

## Developing the HelloSAP iView

Now that you have set up the development project, let's examine how you can use Eclipse to build an SAP portal component based on the J2EE platform. To support the development process, SAP provides a group of interfaces and classes that can be collectively referred to as the "DynPage framework." Client programmers<sup>10</sup> can use these APIs as the foundation and structure for their portal components. This section briefly introduces the DynPage framework and then moves on to describe how the HelloSAP portal component implements it. I will highlight the key pieces of source code and offer a number of tips for working effectively with the Eclipse Workbench.

### The "DynPage Framework"

By definition, a framework provides a reusable structure for software development. The main objective of a framework is to support the creation of applications in a manner that increases development efficiency and at the same time improves quality and maintainability. This objective is achieved by building applications that are based on the framework's collection of interfaces and classes. The client programmer can

then exploit the framework by implementing the interfaces and extending the classes in a predefined, yet flexible way.

Within the EP, the DynPage framework defines the overall structure of an SAP portal component and, with support from the overall Enterprise Portal Client Framework (EPCF), manages its runtime functionality. At the technical level, the DynPage framework is primarily based on two programming interfaces:

- *com.sapportals.portal.prt.component.IPortalComponent*
- *com.sapportals.htmlb.page.DynPage*

The central abstraction of an SAP portal component is the *IPortalComponent* interface, which handles the request from the client, supports the integration of portal services, and delegates control in the application. All SAP custom-developed iViews in Java are based on this interface, and they usually extend a class that supplies the implementation (e.g., *com.sapportals.portal.htmlb.PageProcessorComponent*). In the HelloSAP portal component, the *HelloController* class extends *PageProcessorComponent* to implement the *IPortalComponent* interface.

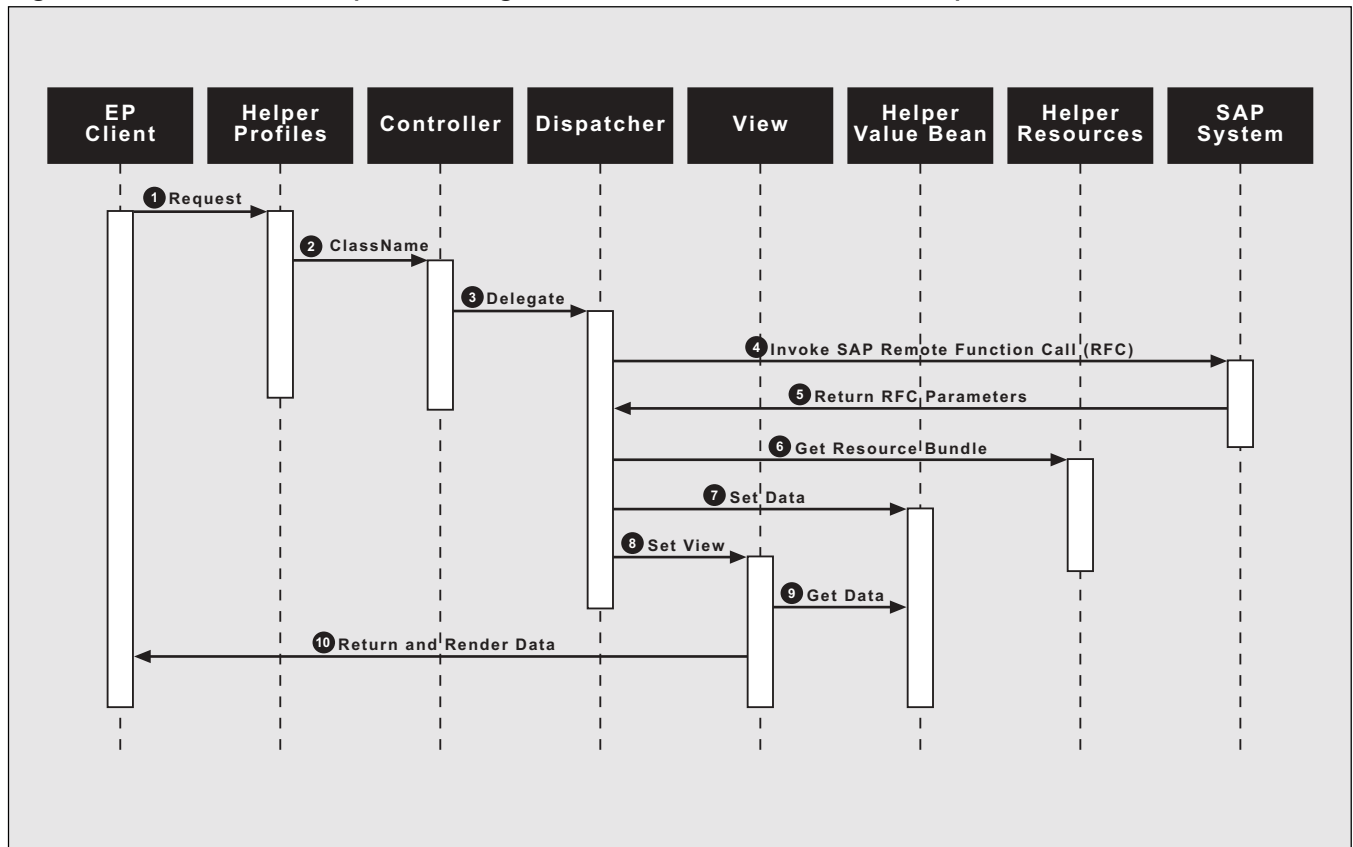
The second major piece of the framework, the abstract class *DynPage*, provides support for event handling, application flow, and view management. As an extension of the *DynPage* class, *com.sapportals.htmlb.page.JSPDynPage* contributes additional support for JavaServer Pages technology to the portal component. Since its output is rendered in a JSP page, the *HelloDispatcher* class of the HelloSAP portal component extends *JSPDynPage*.

### The Application Flow of the HelloSAP iView

At runtime, all of the different files in the portal component project work together within the construct of the DynPage framework. The best way to illustrate this runtime interaction is with a sequence diagram, which is shown for the example HelloSAP

<sup>10</sup> In his book *Thinking in Java*, Bruce Eckel divides Java developers into "class creators" and "client programmers." Class creators, for example, include the experts at SAP who wrote the API for the DynPage framework. You and I, on the other hand, are client programmers. We use the interfaces of the framework as support for our own development efforts. I recommend Eckel's book, available for free at [www.mindview.net](http://www.mindview.net), as a solid reference for Java basics and the language's object-oriented concepts.

**Figure 17** Sequence Diagram for the HelloSAP Portal Component



portal component in **Figure 17**. For the rest of this section, I will follow the flow of this diagram in describing the details of the HelloSAP iView.<sup>11</sup>

### Step 1: The EP Client Request

The processing of the portal component begins with the request from the client. The portal user logs on to the EP, and the iView is launched on the portal page. As I mentioned earlier, the web component request path of the portal component actually points to the

component-specific profile (i.e., the implementation profile) from the deployed PAR. Based on my settings for the PDK, the direct URI to start the HelloSAP iView is:

```
http://de-walapp963:8133/
irj/servlet/prt/portal/
prtroot/com.ibm.gcoe.
HelloSAP.HelloController
```

### Step 2: Helper Profiles and the ClassName Property

The HelloSAP development project contains two profiles:

- *default.properties*
- *HelloController.properties*

<sup>11</sup> My HelloSAP portal component makes use of terms (controller, dispatcher, view, helper) that are often used to describe abstract design patterns for distributed applications. In many ways, the DynPage framework can be seen as an implementation of a pattern called “Service-to-Worker.” For more information about patterns in Java, I recommend Sun’s Enterprise BluePrints site at <http://java.sun.com/blueprints/enterprise>.



**Listing 1: default.properties**

```
ServicesReference=htmlb, jco, jcoclient
tagLib.value=/SERVICE/htmlb/taglib/htmlb.tld
```

**Listing 2: HelloController.properties**

```
ClassName=com.ibm.gcoe.HelloController
```

The *ServicesReference* property in the default profile (see **Listing 1**) is a standard property that specifies the portal services required for the iView at runtime. Each portal service is a collection of classes that offers a specific piece of functionality to the portal component. Portal services in the HelloSAP example include those for HTML Business for Java (*htmlb*), the SAP Java Connector (*jco*), and the JCo Client (*jcoclient*). At runtime, the APIs for these services are loaded to the iView Runtime for Java (IRJ) web component.

The second property in the default profile, *tagLib*, is a custom property that the developer must define and maintain. Its value determines the file location of the tag library descriptor (TLD) for the HTMLB tag library. As I will demonstrate in Steps 8 and 9, the JSP page of the portal component accesses the TLD to render the HTMLB output. The value of the property in the example is the default location of the *htmlb.tld* file, which is located at *tomcat\webapps\irj\services\htmlb\taglib*.

The component-specific implementation profile, *HelloController.properties*, is shown in **Listing 2**. It

**Figure 18** Profile Property Settings for HelloController

**Profile Property Settings**

Title:

ComponentType:

Class/JSP Name:

Authentication Requirement:

Personalization Class:

Personalization Behaviour:

Personalization Link:

Resource Bundle Name:

EPCF Level:

Isolated in PDK:

Custom profile properties:

Name	Value	Inhe...	Pers...	Des...	Plain...

add delete OK Cancel

contains only a single property: *ClassName*.<sup>12</sup> You can maintain this property in the *Class/JSP Name* field on the dialog shown in **Figure 18** (for details on modifying component profiles, see the sidebar “Using Eclipse to Modify Portal Component Profiles” to the right). The value of the *ClassName* property must be

<sup>12</sup> Strictly speaking, the *ClassName* property can also be maintained in the default profile, allowing it to serve as both the default and an implementation profile at the same time. Since this practice is less transparent, I always recommend creating a separate implementation profile, even if the PAR contains only a single portal component, as in the HelloSAP example.

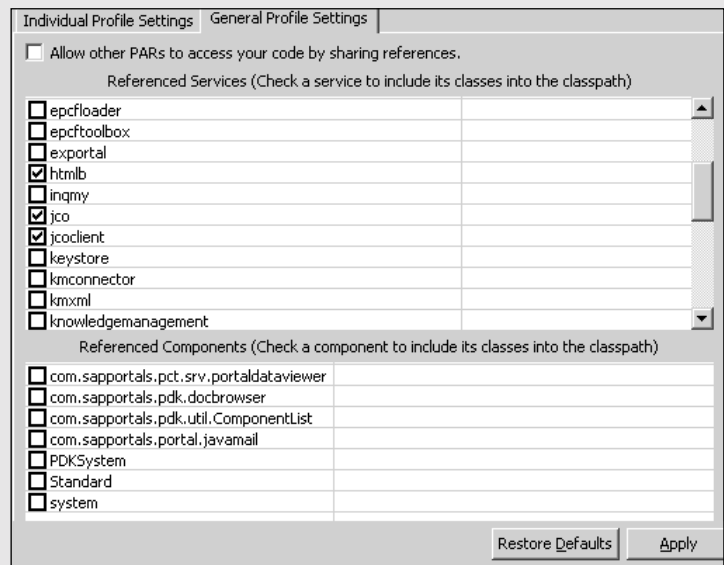
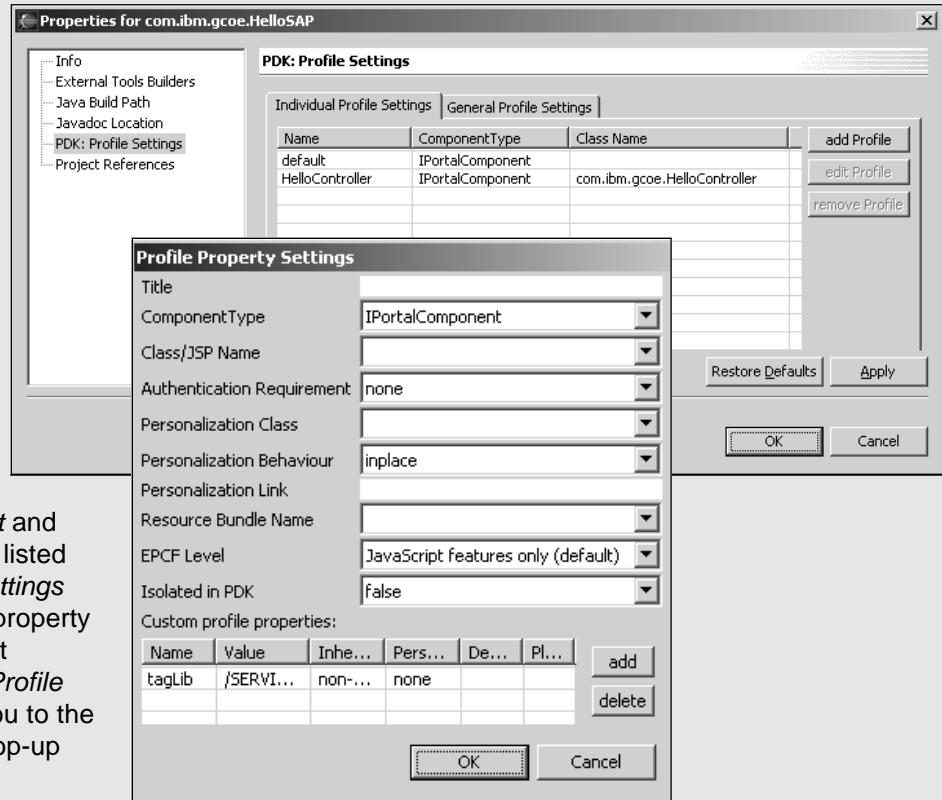
## Using Eclipse to Modify Portal Component Profiles

You can use the project properties dialog in Eclipse to maintain the portal component profiles. To open the properties dialog for the project, select the project name in the Navigator view and choose *File* → *Properties*. On the *Properties* screen (shown at the upper right), select *PDK: Profile Settings*.

As you can see, the *default* and *HelloController* profiles are listed on the *Individual Profile Settings* tab. To modify the profile property settings, choose the default profile and select the *edit Profile* pushbutton, which takes you to the *Profile Property Settings* pop-up shown to the right.

Note that for *default.properties* I have not defined an implementation class in the *Class/JSP Name* field, because the default profile does not trigger an individual portal component. Instead, you will recall that the properties in the default profile apply to all of the portal components in the PAR. You can also see the *tagLib* property at the bottom of the pop-up, which has been created as a custom profile property. Close the pop-up and switch to the *General Profile Settings* tab of the *Properties* screen (shown to the right).

On this tab, you will find the portal services in the *default.properties* file: *htmlb*, *jco*, and *jcoclient*. When you use this tab to manage the portal services, the proper entries are directly made in (or deleted from) the *ServicesReference* property of the default profile. This interface also ensures that the required libraries for each portal service are maintained in the *.classpath* build file.



### Listing 3: *HelloController Delegates to HelloDispatcher*

```
public class HelloController extends PageProcessorComponent {  
    public DynPage getPage() {  
        return new HelloDispatcher();  
    }  
}
```

set to the fully defined implementation class of the portal component, which, for the HelloSAP example, is *com.ibm.gcoe.HelloController*. Normally, you should be able to select the appropriate class from the field's dropdown list.

### Step 3: Controller Delegates to Dispatcher

Based on the *ClassName* property in the *HelloController.properties* profile, the *HelloController* class is first instantiated in the IRJ.<sup>13</sup> The controller acts as the central point of access for all requests to the portal component. As **Listing 3** shows, for each request it receives, *HelloController* instantiates *HelloDispatcher*, returns an object reference of type *DynPage*, and finally delegates processing flow to it.

The dispatcher works to carry out the application logic of the portal component. It also provides event management and view delegation. Specifically, the *HelloDispatcher* class<sup>14</sup> extends the abstract class *JSPDynPage*, and as a result inherits three abstract methods from *DynPage*:

- *doInitialization()*

<sup>13</sup> The *DynPage* framework, which implements the necessary methods from the Java APIs, manages the hidden processes to load and instantiate the *HelloController* class automatically.

<sup>14</sup> While I haven't provided the complete listing for *HelloDispatcher.java* in this article, you can view the source from the imported PAR file in the Eclipse Workbench.

- *doProcessAfterInput()*
- *doProcessBeforeOutput()*

These methods provide the foundation for the structure of the portal component. The sidebar "Using the Portal Component Wizard in Eclipse" on the next page explains how to create a new portal component that extends the *JSPDynPage* class. Using the wizard automatically prepares the project's foundation code, including empty implementations of the methods.

If you are an ABAP developer, you will immediately recognize that the names for the methods were derived from SAP module pool (dialog) programming, where you use the PAI and PBO sub-routine modules. You will provide the same type of processing logic within the Java methods that you have come to know in these ABAP flow logic modules. The *doProcessAfterInput()* method handles user input, while the *doProcessBeforeOutput()* method prepares application data for output and sets the next screen. The *doInitialization()* method, in contrast, is called immediately when the class is executed.

In the *HelloDispatcher* class of the HelloSAP portal component, both the *doInitialization()* and the *doProcessAfterInput()* methods are empty implementations of the abstract methods from *DynPage*. The whole of the application logic in HelloSAP, then, takes place in the *doProcessBeforeOutput()* method.

## Using the Portal Component Wizard in Eclipse

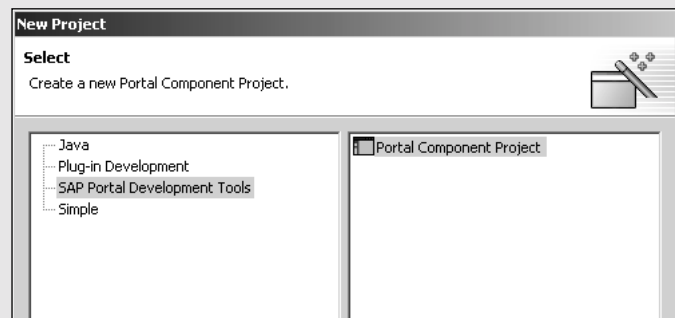
The SAP plug-in for the Eclipse Platform supplies a number of features to support the development of portal components, including a wizard to create a new portal component based on the `JSPDynPage` class.

### ✓ Note!

You should start Tomcat before working with this wizard. For details on installing and configuring Tomcat for use with the PDK, see my article in the November/December 2002 issue of this publication.

Start the Project wizard via the *File* → *New* → *Project...* menu path. Then follow these steps:

1. Select *SAP Portal Development Tools* from the options listed on the left-hand side of the project creation screen (see the screenshot to the right). Highlight *Portal Component Project* and then choose *Next*.



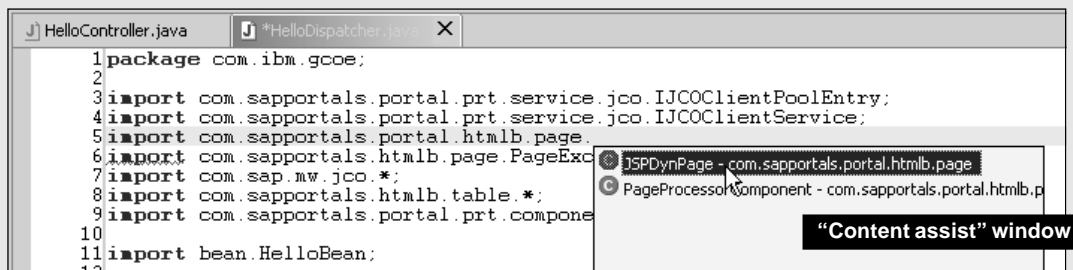
(continued on next page)

### ✓ Tip

While the dispatcher can be included directly within the controller, I prefer to define it as a separate class file. (This practice does not deliver any programmatic advantages, but I find that using two files helps to clarify the different roles of the two classes.) In order to instantiate it, then, `HelloController` must import the `HelloDispatcher` class as follows:

```
import com.ibm.gcoe.HelloDispatcher;
```

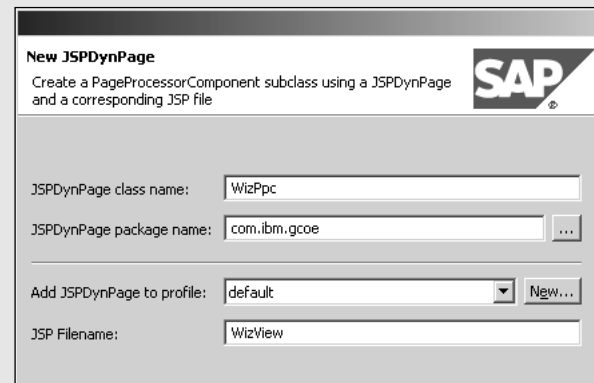
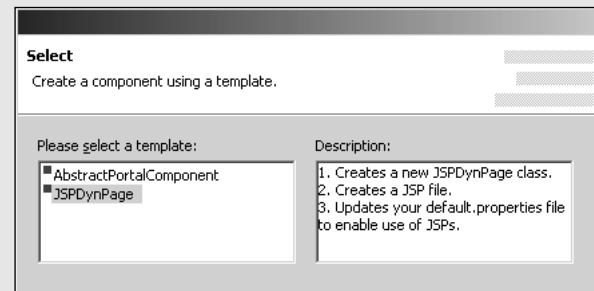
When adding an import directive in the Eclipse Workbench, you can make use of the “content assist” functionality. The screenshot below depicts the floating content assist window with options to complete the import directive for the `JSPDynPage` class in `HelloDispatcher`. Content assist also supports the automatic completion of other Java code, such as package directives or methods, with their required arguments.



(continued from previous page)

2. Give your project a name (the project name is *JspDynPageWizard* in the example here) and set its root folder (e.g., *C:\pdk\projects*), just as you did earlier when importing the portal archive.
3. Choose *Finish*, and the *Select* pop-up shown to the right will appear, where you can choose to create a portal component based on either the *AbstractPortalComponent* or *JSPDynPage* template.

The *AbstractPortalComponent* is a basic class for portal component development, and you can choose it to build iViews that do not require direct support from the DynPage framework. With *JSPDynPage*, you can build iViews that are based on the DynPage framework and that support rendering via JSP pages. If you choose the *JSPDynPage* template, as I have in the example shown here, you can provide the name of the class, the package name, and a JSP page in the next step of the wizard (see the second screenshot to the right).



#### Listing 4: Connect to SAP with Hard-Coded Parameters

```
jcoClient = JCO.createClient("100",
    "vieregger",
    "mypass",
    "EN",
    SAPhost,
    "00");
```

#### Step 4: Invoke the SAP Remote Function Call (RFC)

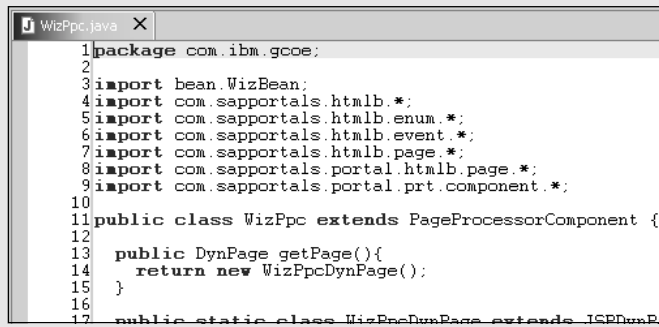
The first task in the implementation of *doProcessBeforeOutput()* is to access the SAP system and invoke the appropriate BAPI (*BAPI\_COMPANYCODE\_GETLIST* in the example here). The recommended method for connecting to

an SAP system is through client pooling, which is supported by the *jcoclient* portal service.<sup>15</sup>

<sup>15</sup> Matthias Edinger, SAP project manager for the PDK, has written an informative article on the technical details of the *jcoclient* service and how it handles client pooling. You can read the article at [www.iivestudio.com](http://www.iivestudio.com). You can also review the source code of the JCo Client Pool iView on the *Documentation* tab of the PDK. The HelloSAP portal component attains its connection to SAP in the same manner.

4. Select the *Next* pushbutton again to use the wizard to define the JavaBeans component, as shown in the screenshot at the lower right. Once you have made the appropriate selections on this screen, you can finally select *Finish*.

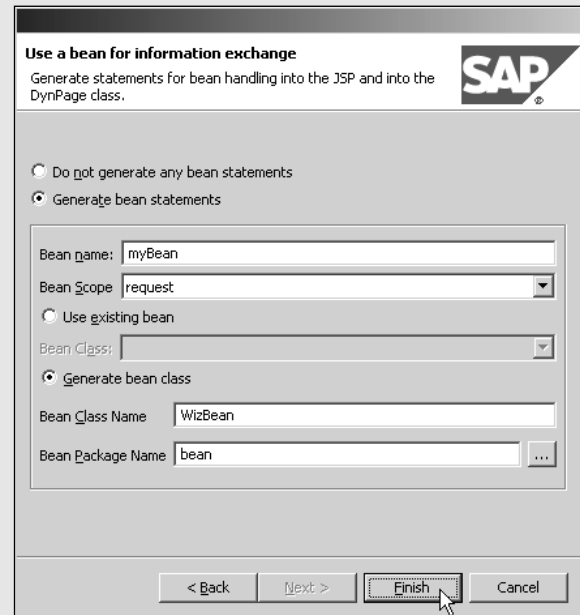
The wizard now creates the development project, with predefined templates for each of the files. The Java source for *WizPpc.java* (see the screenshot at the lower left), for example, extends the *PageProcessorComponent* and contains the implementation for the *getPage()* method. Note that the wizard creates only one Java source file for both the controller and dispatcher classes.



```

1 package com.ibm.gcoe;
2
3 import bean.WizBean;
4 import com.sapportals.htmlb.*;
5 import com.sapportals.htmlb.enum.*;
6 import com.sapportals.htmlb.event.*;
7 import com.sapportals.htmlb.page.*;
8 import com.sapportals.portal.htmlb.page.*;
9 import com.sapportals.portal.prt.component.*;
10
11 public class WizPpc extends PageProcessorComponent {
12
13     public DynPage getPage(){
14         return new WizPpcDynPage();
15     }
16
17     public static class WizPpcDynPage extends JSPDynPage {

```



**Use a bean for information exchange**  
Generate statements for bean handling into the JSP and into the DynPage class.

☐ Do not generate any bean statements  
☒ Generate bean statements

Bean name: myBean  
 Bean Scope: request  
☐ Use existing bean  
 Bean Class:   
☒ Generate bean class  
 Bean Class Name: WizBean  
 Bean Package Name: bean

< Back   Next >   **Finish**   Cancel

### Listing 5: Connect to SAP with Runtime User Parameters

```

IJCOCliientService jcoCliientService = (IJCOCliientService)
    request.getService(IJCOCliientService.KEY);
IJCOCliientPoolEntry jcoPoolEntry =
    jcoCliientService.getJCOCliientPoolEntry("P46_400", request);
JCO.Client jcoClient = jcoPoolEntry.getJCOCliient();

```

If you have previous experience with JCo, you are probably familiar with the *createClient()* method of the *JCO* class shown in **Listing 4**.

The arguments for the method provide the parameters, including a hard-coded user ID and password, to enable programmatic access to a remote SAP

system. Instead of using hard-coded parameters, *HelloDispatcher* determines the user information from the *request* object and utilizes the *jcoclient* service to read the SAP system destination information. Based on these parameters, the service can dynamically create the *jcoClient* object that enables the connection, as shown in **Listing 5**.

**Listing 6: Get the Request Object for the Portal Component**

```
IPortalComponentRequest request = (IPortalComponentRequest)
    this.getRequest();
```

**Listing 7: Create the Function Object for the RFC**

```
IRepository jcoRepository = JCO.createRepository("Repository",
    jcoClient);
IFunctionTemplate jcoFunctionTemplate =
    jcoRepository.getFunctionTemplate("BAPI_COMPANYCODE_GETLIST");
JCO.Function jcoFunction = new JCO.Function(jcoFunctionTemplate);
```

The method `getJCOClientPoolEntry()` requires two arguments: a *string* object with the SAP destination name (*P46\_400* in the example) from the *jcoDestinations.xml* file, and a *request* object, which is created as type *IPortalComponentRequest* (see **Listing 6**).

The next block of code (**Listing 7**) creates the *jcoFunction* object, which serves as the Java representation of the SAP Remote Function Call (RFC) or BAPI. To create the object, the name of the SAP BAPI (*BAPI\_COMPANYCODE\_GETLIST*) is supplied as the argument to the `getFunctionTemplate()` method.

The output of the BAPI in an SAP IDES system is shown in **Figure 19**.

If the RFC or BAPI required any input parameters, they would also be set here, but *BAPI\_COMPANYCODE\_GETLIST* can be executed directly:

```
jcoClient.execute(jcoFunction);
```

Once the BAPI has been executed in the SAP

**Figure 19** Execute the BAPI via SE37 in SAP

COMP	COMP_NAME
0001	SAP A.G.
1000	Giotto LTD
2000	IDES UK
2100	IDES Portugal
2200	IDES France
2300	IDES España
2400	IDES Italia
2500	IDES Netherlands
3000	IDES US INC
4000	IDES Canada
5000	IDES Japan
6000	IDES México, S.A. de C.U.
F100	Bankhaus Frankfurt
F300	Liberty Bank
R100	IDES Retail GmbH
R300	IDES Retail INC US



system, you can release the *jcoClient* object to put it back in the client pool:

```
jcoPoolEntry.release();
```

### ✓ **Note!**

*In order to maintain the simplicity of the HelloSAP iView, I have not included robust error checking and Java exception handling in the portal component code. For instance, if the BAPI execution results in an error, you should not release the pooled entry, but instead delete it. An example of this scenario is demonstrated in Matthias Edinger's JCo Client article on the iViewStudio web site ([www.iviewstudio.com](http://www.iviewstudio.com)). Also, after the actual execution of the BAPI (Step 5), I do not check the return code export parameter, which is important to determine the final status of the execution. You can find further examples using JCo, in addition to the complete *com.sap.mw.jco.JCO* API, in the PDK.*

## Step 5: Get RFC Return Parameters

After the successful call of the BAPI, any table or export parameters that are required for further processing must be retrieved from the *jcoFunction* object (see **Listing 8**).

The output of the BAPI in my example (shown in Figure 19) is provided in a table parameter, *COMPANYCODE\_LIST*, which can be accessed with the *getTable()* method. (Other methods provide

access to export fields and structures, or even to the entire export parameter list.) The table parameter is then set into the *jcoTable* object.

The BAPI table parameter is transferred from *jcoTable* to the *jcoModel* object. The *TableViewModel* interface, which is implemented by *JCOTableViewModel*, is part of the DynPage framework and supports the data format required for tabular output on JSP pages. The interface also defines a number of methods that enable runtime manipulation of the data in the *JCOTableViewModel*:

```
JCOTableViewModel jcoModel = new  
    JCOTableViewModel(jcoTable);
```

## Step 6: Get Resource Bundle

The resource bundle includes additional properties files with language-specific text for the portal component. Before the BAPI return data is rendered in the iView, the *HelloController* class will determine the language settings for the EP user and access the properties file for that language. I will demonstrate how to modify my HelloSAP example iView with language-independent features in the final section of this article.

## Step 7: Set Data

From *jcoModel* you can set the SAP table parameter data to the JavaBeans component, *HelloBean*. In the HelloSAP example, the bean class, which I describe

### **Listing 8: Get the BAPI's Table Parameters**

```
JCO.Table jcoTable =  
    jcoFunction.getTableParameterList().getTable("COMPANYCODE_LIST");
```

**Listing 9: Set the BAPI Data in the JavaBeans Component**

```
valueBean.setBeanModel(jcoModel);  
profile.putValue("idBean", valueBean);
```

in the sidebar on the next page, is instantiated as *valueBean*. The *setBeanModel()* method from the bean works to save the SAP table parameter from *jcoModel* to an attribute in the bean called *BeanModel* (see **Listing 9**).

After the data is set to the bean, the *profile* object uses the *putValue()* method to bind the *valueBean* object to the current application context, associating it with the string name *idBean*. Essentially, this binding enables the JSP page to locate the bean at a later time and render the saved data (in Step 8).

**Step 8: Set View**

Once the SAP system has been queried and the resulting data model has been set in the bean, the application flow is passed to the view (i.e., the JSP page) via the *setJspName()* method. The argument (*HelloView.jsp*) for the method determines the name of the JSP page to execute:

```
this.setJspName("HelloView.jsp");
```

**Step 9: Get Data**

The JSP page *HelloView.jsp* utilizes the HTMLB tag library and the *htmlb* portal service to render the output for the portal component. Access to the web controls of the HTMLB tag library is provided via the *<taglib>* element, where the attribute for the universal resource indicator (*uri="tagLib"*) defines the location of the *htmlb.tld* library descriptor file (you will recall that the URI value *tagLib* is a custom property maintained in the default profile and set to the directory location of the TLD file):

```
<%@ taglib uri="tagLib"  
prefix="hbj" %>
```

The value of the prefix attribute (*prefix="hbj"*) is freely definable. It sets the identifier for tags based on HTMLB. Within this JSP page, then, any tag that begins with *hbj* will look for its element and attribute values in *htmlb.tld*.

The standard tag indicator for JavaServer Pages technology is *jsp*. In the *HelloView.jsp* file, the *<jsp:useBean>* element serves to locate the JavaBeans component *valueBean*, which is acting as the data container object for the SAP BAPI table parameter:

```
<jsp:useBean id="idBean"  
scope="application"  
class="bean.HelloBean" />
```

You will recognize that the value of the *id* attribute is the string name (the first parameter; here, *idBean*) from the *putValue()* method in the *HelloDispatcher* class. Once the *<jsp:useBean>* element locates the *valueBean* object, the logic behind the element defines an object reference variable (with the name *idBean*) and stores a reference to the bean object in it. The end effect of *<jsp:useBean>* is to expose the “getter” and “setter” methods, and consequently the data, in the *valueBean* object.

The *HelloView.jsp* file uses four different elements from the HTMLB tag library. The elements *content*, *page*, and *form* describe the layout structure of the iView. The other element in the example,

## The Definition of the JavaBeans Component

The *HelloBean* JavaBeans component provides a data container that can only be accessed from within the HelloSAP portal component. Data that is retrieved and manipulated in the *HelloDispatcher* class (here, the SAP BAPI table parameter) can be written to the bean. Later, this data can be read and rendered by the *HelloView* JSP page. As I already mentioned, the *HelloBean* JavaBeans component is instantiated as a *valueBean* in the *HelloDispatcher* class:

```
valueBean = new HelloBean();
```

The standard format for a JavaBeans component defines a bean property (here, *beanModel*) and what are known as the property's "getter" and "setter" methods:

```
package bean;
import com.sapportals.htmlb.table.JCOTableViewModel;

public class HelloBean {

    public HelloBean () {

    }

    public JCOTableViewModel model;

    public JCOTableViewModel getModel() {
        return model;
    }

    public void setModel(JCOTableViewModel jcoModel) {
        this.model = jcoModel;
    }
}
```

In Step 6, the "setter" method *setBeanModel()* copies *jcoModel* into the *beanModel* property of *valueBean*. On the other hand, the "getter" method *getBeanModel()* allows the bean property *beanModel* to be read from other classes or components. I will use the *getBeanModel()* method (indirectly) in Step 8 to retrieve the data model for output on the JSP page.

It is important to note that typical JavaBeans components do not have different names for the individual parameters of the "getter" and "setter" methods, as I have done with *beanModel* and *jcoModel*.

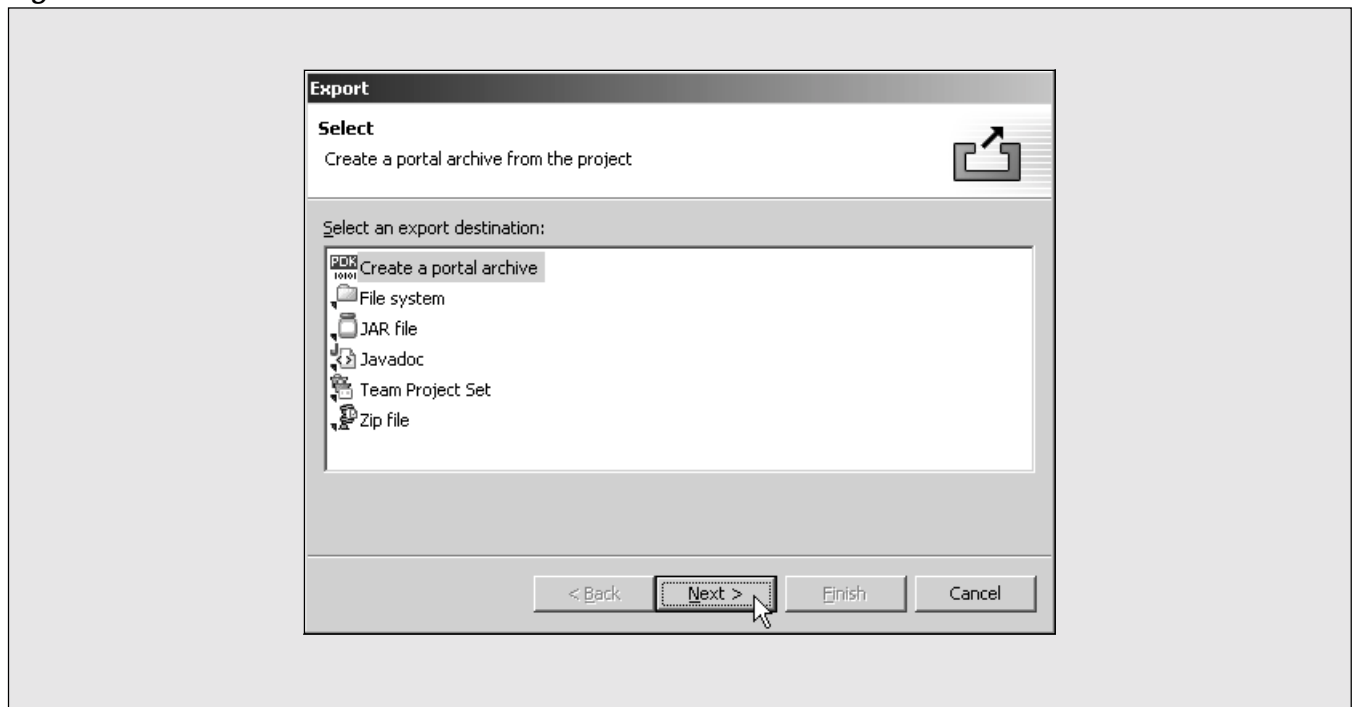
*tableView*, provides for the rendering of the bean's data in the EP:

```
<hbj:tableView id = "idTableView"
    model = "idBean.BeanModel" />
```

The *model* attribute in the *tableView* element provides direct access to the maintained properties in a JavaBeans component. To read from the *valueBean*, the value of the *model* attribute should be the value of the *id* attribute in the *<jsp:useBean>*

Figure 20

## Create the Portal Archive



element (*idBean*), with a reference to the property (*BeanModel*) you want to get from the bean. While the *getBeanModel()* method is not explicitly executed in the JSP page, the application logic behind the model attribute provides the same result.

The *tableView* element also supports a number of other attributes, which are beyond the scope of this article. You can find the complete list of possible attributes and their arguments, as well as further information about the other elements, on the *Documentation* tab of the Portal Development Kit.

### Step 10: Return and Render Data

The final step, as depicted in the sequence diagram shown in Figure 17, is the rendering of the data in the Enterprise Portal.

When it is served at runtime, the JSP page *HelloView.jsp* is translated into a Java servlet class

called *\_sapportalsjsp\_HelloView.java*.<sup>16</sup> Finally, the Tomcat servlet engine compiles and executes this servlet class to render the output of the portal component.

### Build and Deploy the Portal Archive

To build and deploy the HelloSAP development project, choose *File* → *Export...* from the menu. On the first screen of the Export wizard, which is shown in **Figure 20**, make sure that *Create a portal archive* is highlighted and select *Next*. Choose the HelloSAP project from the available list (see **Figure 21**) and select *Next* again. In the final step (shown in **Figure 22**), select all three checkboxes. The option *Deploy portal component to local portal* will upload and deploy to your PDK the newly created portal archive that was configured in

<sup>16</sup> The servlet class from the JSP page of the HelloSAP example is stored at *tomcat\webapps\irj\WEB-INF\plugins\portal\resources\com.ibm.gcoe.HelloSAP\work\pagelet*.

Figure 21

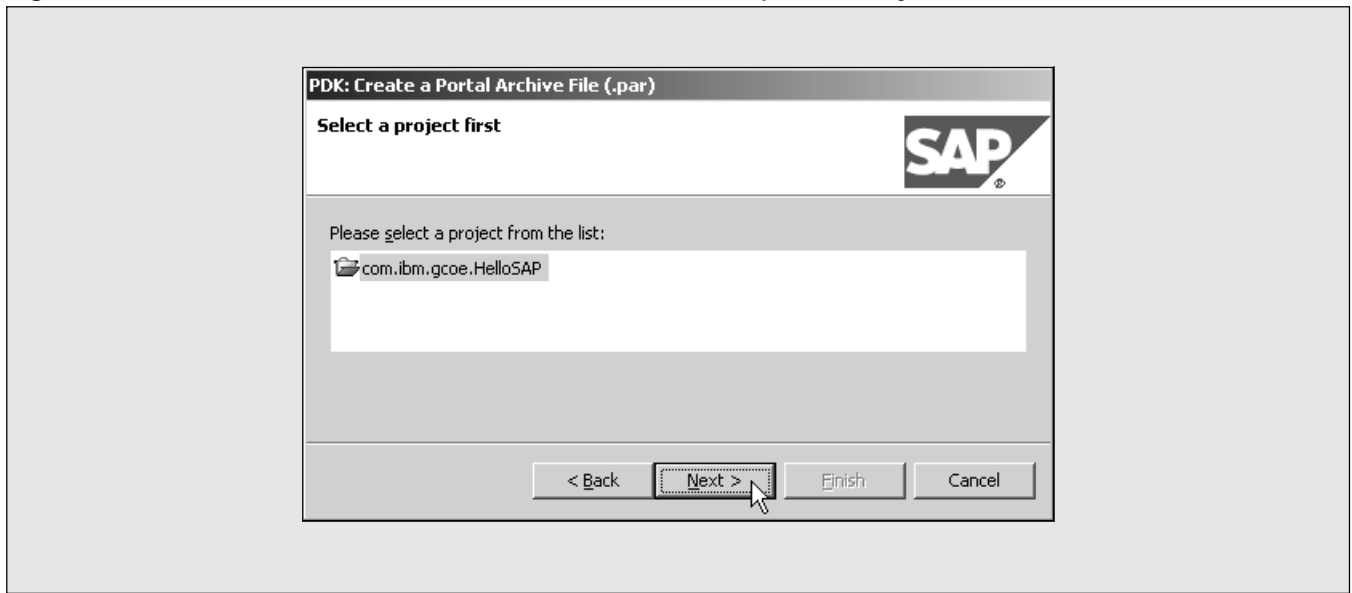
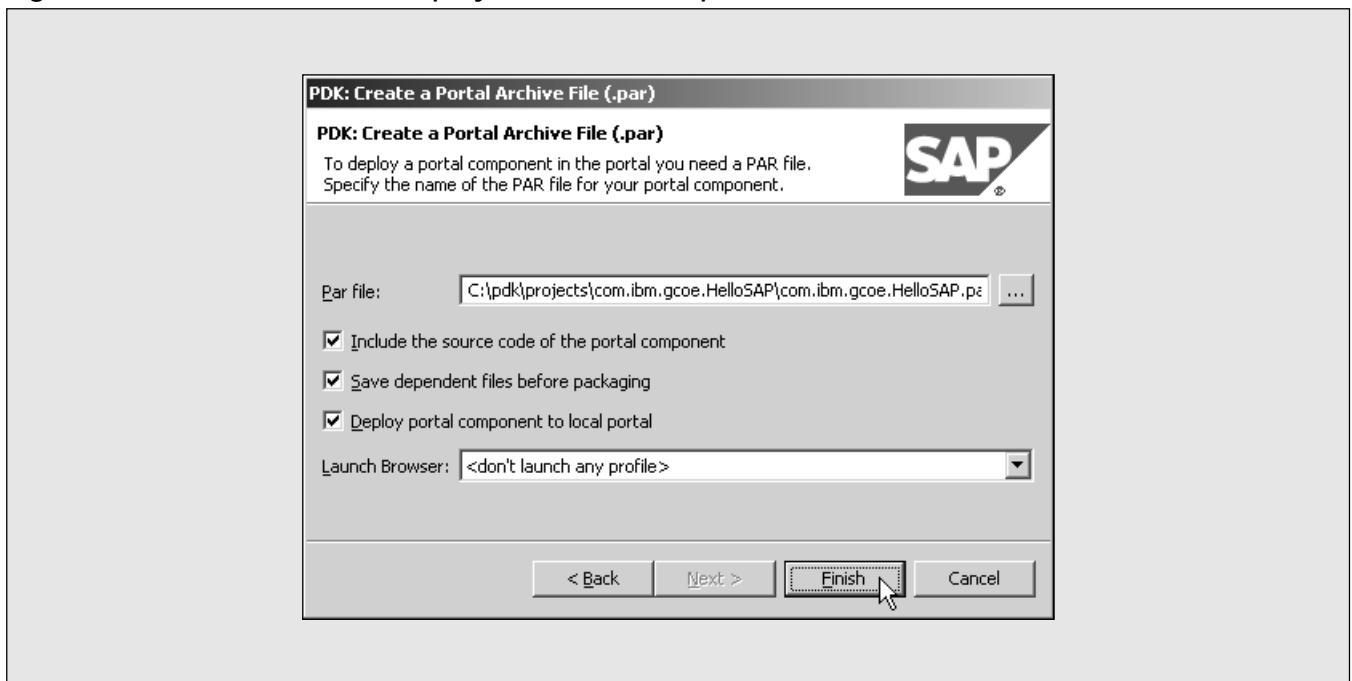
*Select the HelloSAP Development Project*

Figure 22

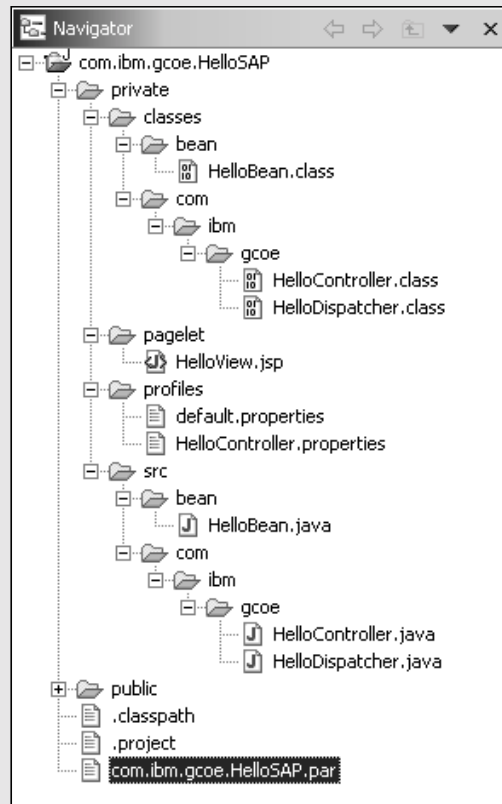
*Deploy the Portal Component to the PDK*

Window → Preferences. Do not launch a profile<sup>17</sup> and, finally, select the *Finish* pushbutton.

<sup>17</sup> Since JCo requires user parameters from the portal framework, it is not possible to test the portal component outside of the PDK.

You may need to refresh the Navigator view in the Eclipse Workbench in order to see the newly created PAR file. The PAR file is stored in the root directory of the HelloSAP development project.

Figure 23

*The com.ibm.gcoe.HelloSAP.par File in the Project*

**Figure 23** shows the file displayed in the Eclipse Navigator view. You can also go to the project root directory in Windows Explorer and open the *com.ibm.gcoe.HelloSAP.par* file with your extract utility. You can further open the JAR (which holds the CLASS files) and the SRC.JAR (which holds the Java source files). The contents from all of these files are shown in **Figure 24**.

**Test the HelloSAP iView in the PDK**

Part of the deployment process extracts the archive to the resources folder under *tomcat\webapps\irj\*

*WEB-INF\plugins\portal*. Once deployed, all of the portal components, including the HelloSAP example, can be accessed in the PDK from the *DevTools* → *ComponentInspector*. (While you do not have to be logged in to access the *ComponentInspector*, the HelloSAP portal component does require an authenticated user to execute properly.) On the *Overview* tab of the *ComponentInspector*, scroll down to find the HelloSAP example and select the *Add to Favorites* link, as depicted in **Figure 25**.

The *ComponentInspector* should change to the *My Personal Components* tab, and the HelloSAP example should be visible. Choose the *Start* link for

Figure 24

Open the PAR, JAR, and SRC.JAR Files

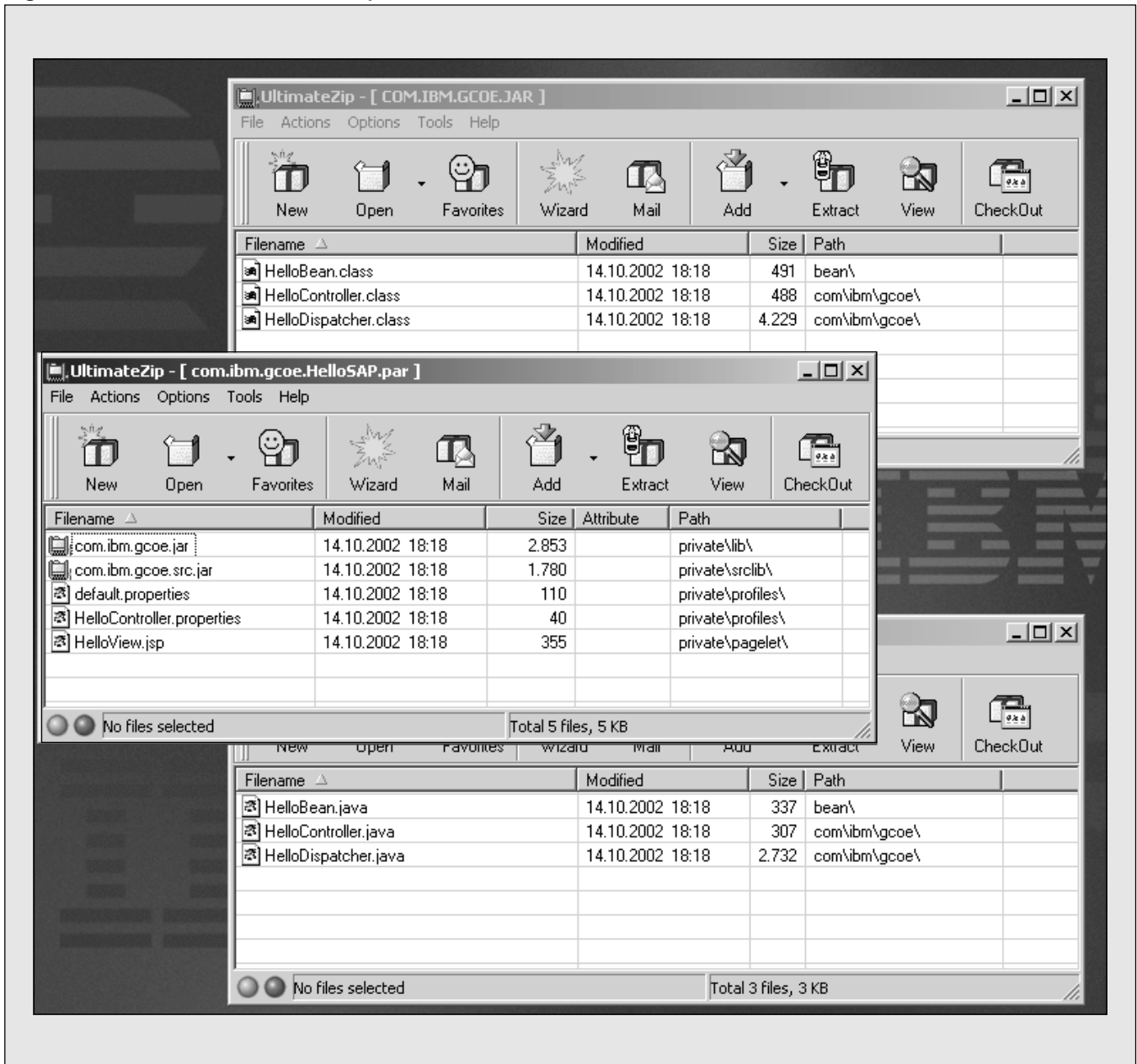


Figure 25

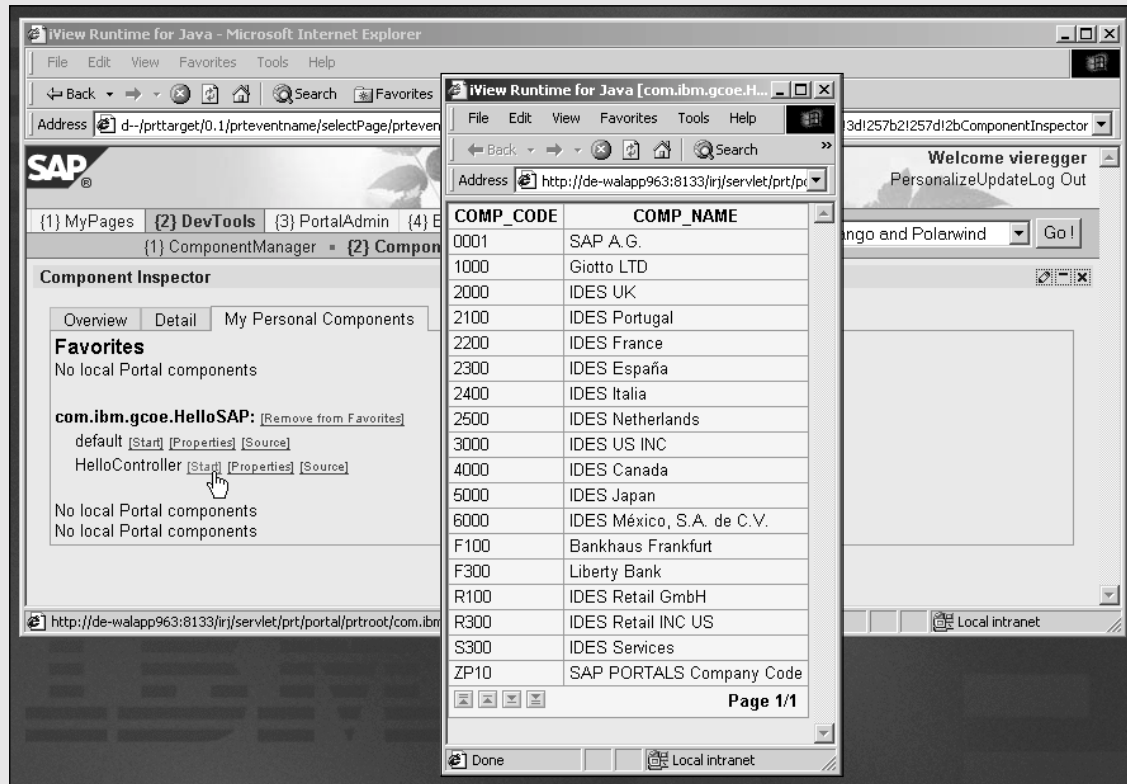
Add the HelloSAP Example to Your Favorites

```
com.ibm.gcoe.HelloSAP [Details] [Add to Favorites]
com.pwcconsulting.ecoe.jdbc.hsql [Details] [Add to Favorites]
com.pwcconsulting.ecoe.mysql [Details] [Add to Favorites]
```



Figure 26

Test the HelloSAP Portal Component in the PDK



*HelloController* to execute the iView in a new browser window. The successful execution of the HelloSAP iView is shown in **Figure 26**.

## Internationalization for the HelloSAP iView

I have never worked on an SAP project where internationalization (i.e., language-independent development for end users in different countries) has not been an integral part of the business requirements. Since the Java programming language was designed from its core to write internationalized applications, you can easily modify the HelloSAP portal component to

be language-independent. The standard API is provided in the *java.util* package, which you must import to your dispatcher class:

```
import java.util.ResourceBundle;
```

In this final section, I will make use of the Java API and SAP's internationalization portal service to provide language features in English and in German.

## Modify the Dispatcher Class for Internationalization

As you can see in Figure 26, the default column headings in the HelloSAP iView (*COMP\_CODE* and *COMP\_NAME*) are not very descriptive. You can

**Listing 10: Use the Internationalization Features for iViews**

```

ResourceBundle resource = request.getResourceBundle();
String varBukrs = resource.getString("t001-bukrs");
String varButxt = resource.getString("t001-butxt");
jcoModel.setColumnName(varBukrs, 1);
jcoModel.setColumnName(varButxt, 2);

```

modify the headings using the *setColumnName()* method of *TableViewModel*:

```

jcoModel.setColumnName("Code", 1);
jcoModel.setColumnName("Company
    Name", 2);

```

The first argument of the method sets the text, and the second argument determines the column number. The problem with this simple approach is that the column headings must be hard-coded in a single language. To use the internationalization

features of portal component development, you still use the *setColumnName()* method, but you must first retrieve the language settings of the user and then access the appropriate language resource file, as shown in **Listing 10**.

The *ResourceBundle* is retrieved with the *request* object (created in Step 4 of the previous section), and it includes the language settings for the current user and session. (See the sidebar below for more information about modifying your default language settings.) The *getString()* method of the *resource* object

**Language Settings for the Enterprise Portal and the Portal Development Kit**

When you first start the PDK, it will pick up the default language of your browser to determine the language settings for the welcome and login fields. The first screenshot below shows the default for English, while the second is German. The default language setting for Microsoft Internet Explorer is maintained under *Tools → Internet Options...* on the *General* tab. Choose the *Languages* pushbutton to access the pop-up shown to the right.

(continued on next page)

passes a predefined string (here, *t001-bukrs* and *t001-butxt*) and returns the required value (based on the *resource* object language setting) from the appropriate language resource file.

### Create the Resource Bundle in the Development Project

The resource bundle for internationalization is made up of a single default resource, plus individual resource files for each required language. The default resource file is named *localization.properties* and is used when the particular language setting for a user is not maintained in the resource bundle. Since I have maintained resource files for English and German, the

default resource file would be accessed, for example, if my PDK user's language were set to Spanish.

The entire resource bundle, including the default (*localization.properties*) and the individual files for each language (here, *localization\_en.properties* and *localization\_de.properties*), must be maintained in the root of the *src* subfolder of the development project (see **Figure 27**).

Inside each of the language resource files, a name/value pair is maintained for each piece of text in an iView. The HelloSAP example iView contains only text for the column headings, but you can also maintain the text for GUI elements, such as text fields and pushbuttons, in addition to standard HTML text, on the JSP page.

(continued from previous page)

When you are logged in to the PDK, the EP framework first looks for a mandatory language in the *workplace.properties* file at *tomcat\irj\WEB-INF\plugins\portal\system\properties*. If the *mandatorylanguage* parameter is maintained (as is the case with the default settings of my PDK), all of the portal components will be rendered in that language, regardless of the individual user's language settings:

```
request.mandatorylanguage=en
```

You can comment out the *mandatorylanguage* parameter with the number symbol (#) or just leave its value blank. In its place, insert the *defaultlanguage* parameter to maintain the PDK's default language:

```
request.defaultlanguage=en
```

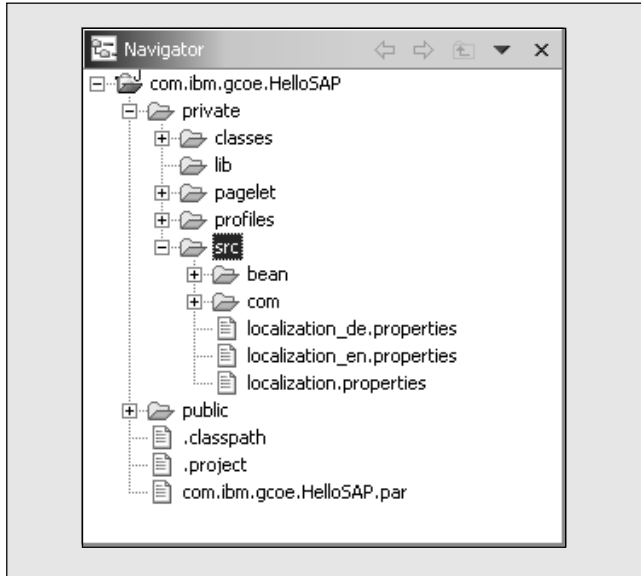
When no mandatory language is set, the framework will read the language settings of the logged-in user. If no user-specific setting is maintained, the framework will use the default language from *workplace.properties*.

You can modify the settings of your PDK user in the *KMUsers.properties* file, where the user-specific language is maintained via the *locale* parameter:

```
vieregger.locale=en
```

If I execute the portal component with locale *en*, the framework will read the resource file *localization\_en.properties* and render the iView in English. To work with the provided resource bundle for my HelloSAP example, you can change your user's locale parameter to *de*, the language abbreviation for German. Note that after making changes to either the *workplace.properties* or *KMUsers.properties* file, you must restart Tomcat for any modifications to take effect.

**Figure 27** The Resource Bundle in the HelloSAP Development Project



The language resource file for English (*localization\_en.properties*), which is the same as the default, is:

```
t001-bukrs=Code
t001-butxt=Company Name
```

The language resource file for German (*localization\_de.properties*) is:

```
t001-bukrs=Buchungskreis
t001-butxt=Firmen-Bezeichnung
```

The parameter name, as you can see, is used as the first argument in the *setColumnName()* method shown in Listing 10. The value of the parameter in the resource file, then, is maintained with the language-specific translation of the text. For example, the value of *t001-bukrs* is *Code* in the English resource file and *Buchungskreis* in the German. The English-language version of the HelloSAP iView is depicted back in Figure 1, in the introductory section of this article. **Figure 28** shows the German-language version of the HelloSAP iView.

**Figure 28** The German-Language HelloSAP iView

Buchungskreis	Firmen-Bezeichnung
0001	SAP A.G.
1000	Giotto LTD
2000	IDES UK
2100	IDES Portugal
2200	IDES France
2300	IDES España
2400	IDES Italia
2500	IDES Netherlands
3000	IDES US INC
4000	IDES Canada
5000	IDES Japan
6000	IDES México, S.A. de C.V.
F100	Bankhaus Frankfurt
F300	Liberty Bank
R100	IDES Retail GmbH
R300	IDES Retail INC US
S300	IDES Services
ZP10	SAP PORTALS Company Code

Seite 1/1

### Rebuild and Redeploy the Portal Archive

After you modify the *HelloDispatcher* class and create the corresponding resource files in *src*, you must rebuild and redeploy the PAR file. Once again, select the *File* → *Export...* menu path. The default settings should match your original deployment, so you can simply finish the wizard. Now go back to the PDK and start the HelloSAP iView on the *My Personal Components* tab. As a test, change the language settings for your PDK user to German and restart Tomcat. Finally, start the portal component again to view the German-language column names: *Buchungskreis* and *Firmen-Bezeichnung*.

## Conclusion

As SAP continues to move its mySAP Enterprise Portal to a pure J2EE-based environment, the role of

custom-developed iViews in Java will only increase in importance for your EP projects. Now that you have walked through the complete development process for an example Java iView, you should be ready to start on your own development projects in Eclipse. While the HelloSAP example iView only makes use of basic functionality from the DynPage framework and the HTMLB tag library, it still outlines the basic structure to access data from SAP. More advanced iViews might include dropdown lists or input fields, where users can insert data for BAPI input parameters. Going a step further, you could then set up personalization to save and restore these parameters for individual users. Further information on how to implement these features, as well as examples of other portal component services, is available in the PDK.

*Carl Vieregger is a senior consultant with IBM Business Consulting Services. Since 1998 he has been a member of the Global SAP Centre of Expertise (GCOE) based in Walldorf, Germany. After starting as a certified ABAP developer and business workflow specialist, his current focus is on portals and portlet application development. Recently, he co-delivered a lecture and workshop on the PDK at the SAP TechEd conference. Prior to moving to Germany, he graduated from Northwestern University in Evanston, Illinois, with a B.A. in International Studies and Slavic Languages & Literature. You can reach him at [carl.vieregger@computer.org](mailto:carl.vieregger@computer.org).*