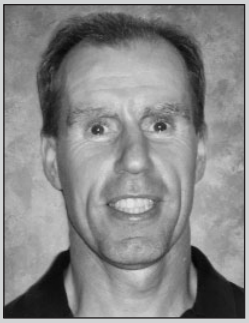
Developing Custom Applications for SAP Enterprise Portal — Advanced Java and .NET Options to Consider in Light of SAP NetWeaver

Patrick Dixon



Patrick Dixon is a manager with Deloitte specializing in SAP Enterprise Portal and Master Data Management (MDM). He has over five years of experience Web-enabling and integrating SAP systems with Enterprise Portal, ITS, Exchange Infrastructure (XI), and Plumtree Software solutions. Patrick was a key speaker at SAP BW and Portals 2004 and SAP Portals 2003.

(complete bio appears on page 148)

Writing a custom application for delivery via SAP Enterprise Portal (SAP EP) no longer means simply writing a custom ABAP report or module pool — SAP NetWeaver's native support for Java and .NET, in addition to ABAP, has dramatically expanded the menu of languages, tools, and approaches available to you for custom development. Each option has important pros, cons, prerequisites, and strategic implications that will affect your long-term costs and the lifetime of your code, and each yields an important lesson: All modern SAP teams need to define a deliberate development strategy to avoid ending up with an inventory of disparate applications that cannot be easily maintained or upgraded. Unfortunately, few companies have found time to do a true apples-to-apples comparison to serve as the basis for developing this strategy.

To help you make the right choices for your own environment, over the course of this two-part article series I compare your eight main options for developing custom content and applications for delivery via SAP EP:

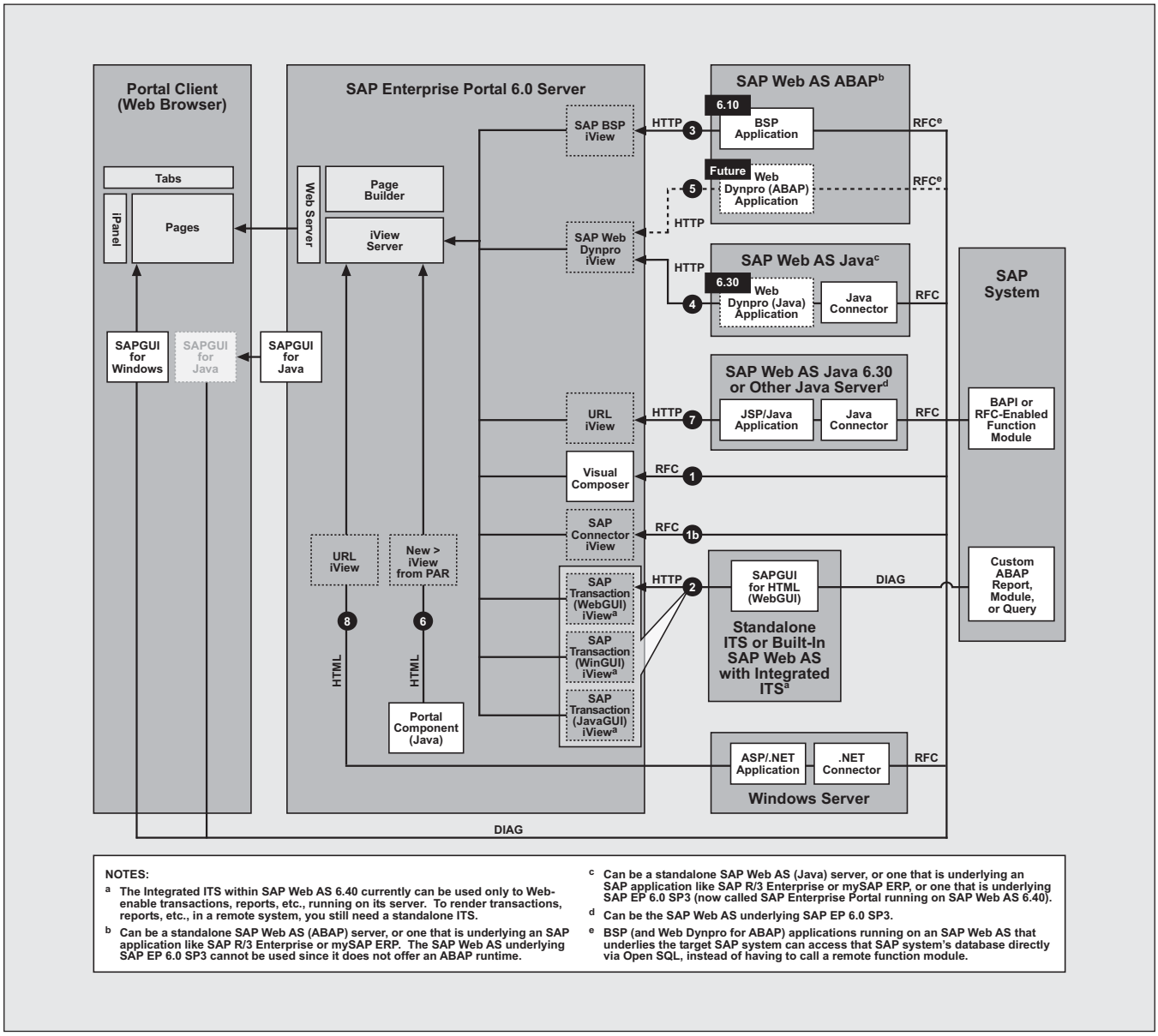
1. Write a custom SAP function module and deploy it via the SAP Visual Composer.

2. Write a custom SAP transaction or report and deploy it via an SAP Transaction (SAPGUI) iView.

3. Write a custom Business Server Pages (BSP) application and deploy it via an SAP BSP iView.

4. Write a custom Web Dynpro for Java application and deploy it via an SAP Web Dynpro iView.

Figure 1 Custom Development Options Overview



NOTES:

[a] The Integrated ITS within SAP Web AS 6.40 currently can be used only to Web-enable transactions, reports, etc., running on its server. To render transactions, reports, etc., in a remote system, you still need a standalone ITS.

[b] Can be a standalone SAP Web AS (ABAP) server, or one that is underlying an SAP application like SAP R/3 Enterprise or mySAP ERP. The SAP Web AS underlying SAP EP 6.0 SP3 cannot be used since it does not offer an ABAP runtime.

[c] Can be a standalone SAP Web AS (Java) server, or one that is underlying an SAP application like SAP R/3 Enterprise or mySAP ERP, or one that is underlying SAP EP 6.0 SP3 (now called SAP Enterprise Portal running on SAP Web AS 6.40).

[d] Can be the SAP Web AS underlying SAP EP 6.0 SP3.

[e] BSP (and Web Dynpro for ABAP) applications running on an SAP Web AS that underlies the target SAP system can access that SAP system's database directly via Open SQL, instead of having to call a remote function module.

5. Write a custom Web Dynpro for ABAP application and deploy it via an SAP Web Dynpro iView (future option).

6. Write a custom portal component in Java and deploy it via a PAR-based iView.

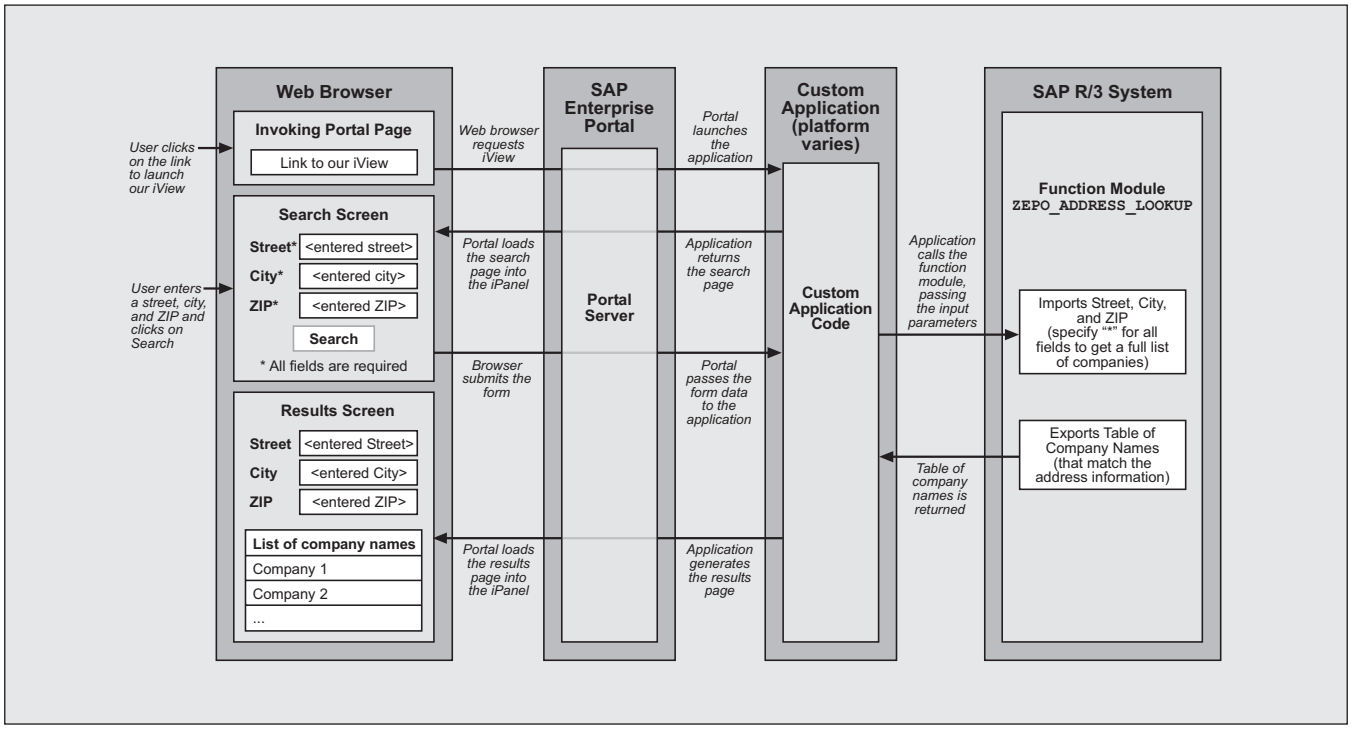
7. Write a custom J2EE application and deploy it via a generic URL iView.

8. Write a custom ASP/.NET application and deploy it via a generic URL iView.

Figure 1 provides a graphical overview of these options and how they fit into an SAP EP landscape. The first installment[1] of this two-part article series

[1] "Developing Custom Applications for SAP Enterprise Portal — Starting with the 'Right' Options in Light of SAP NetWeaver" (*SAP Professional Journal*, March/April 2005).

Figure 2 *Example Application Flow*



> ✓ ***Note!***
>
> *Since most customers have by now upgraded from SAP EP 5.0 to SAP EP 6.0, I will limit the discussion to SAP EP 6.0. Most of the discussion still applies to SAP EP 5.0, but note that the menu paths and iView types will differ.*

covered options 1-5; options 6-8 are covered here in this second installment. As in the first installment, we use the different options to build the same simple application to give you a clear picture of the platform and skill set requirements, level of effort, and ease of coding involved in each option. This article will also introduce you to an SAP EP feature called "eventing" that enables your iViews to communicate with one another, so that when a user performs an action in one iView (e.g., enters a value), other iViews can be notified and take appropriate action (e.g., refresh a chart based on that value). Eventing allows you to build complex applications on the client side without overloading the portal server.

For your reference, let's briefly review the example application we'll be using to illustrate the options.

Example Application Overview

The example application is a simple search scenario: the user enters an address and clicks on the Search button; the application retrieves the company with that address from the SAP system. As you can see in **Figure 2**, the root of the application is a function module in SAP R/3 called `ZEPO_ADDRESS_LOOKUP`,

Figure 3 SAP R/3 Address Lookup Function Each Example Will Call

```
FUNCTION ZEPO_ADDRESS_LOOKUP.
*"----------------------------------
*"*"Local interface:
*"  IMPORTING
*"     VALUE(STREET) TYPE  STRING
*"     VALUE(CITY) TYPE  STRING
*"     VALUE(ZIP) TYPE  STRING
*"  TABLES
*"      EPOCOMPANY STRUCTURE  ZCOMPNAMES OPTIONAL
*"----------------------------------


DATA: A TYPE STRING, B TYPE STRING,
C TYPE STRING.

A = 'NO ORGANIZATION NAME FOUND FOR YOUR INPUTS. PLEASE TYPE '.
B = ' * AS WILDCARD FOR ALL THE INPUT FIELDS'.

CONCATENATE A  B INTO C.

TRANSLATE STREET TO UPPER CASE.
TRANSLATE CITY TO UPPER CASE.
TRANSLATE ZIP TO UPPER CASE.

IF STREET EQ '*' and CITY EQ '*' AND ZIP EQ '*'.
    EPOCOMPANY-COMPANYNAME = ' DELOITTE USA LLP '.
    INSERT TABLE EPOCOMPANY.
    EPOCOMPANY-COMPANYNAME = ' MICROSOFT USA '.
    INSERT TABLE  EPOCOMPANY.
    EPOCOMPANY-COMPANYNAME = ' HEWLETT AND PACKARD  '.
    INSERT  TABLE EPOCOMPANY.

ELSEIF STREET EQ 'VANENBURG' AND CITY  EQ 'HYDERABAD' AND ZIP EQ
'500082'.
    EPOCOMPANY-COMPANYNAME = 'DELOITTE CONSULTING INDIA PVT LTD'.
    INSERT  TABLE  EPOCOMPANY.
    EPOCOMPANY-COMPANYNAME = 'BAAN INDIA PVT LTD'.
    INSERT  TABLE EPOCOMPANY.

ELSEIF STREET EQ 'HITECH' AND CITY   EQ 'HYDERABAD' AND ZIP EQ '500082'.
EPOCOMPANY-COMPANYNAME = 'MICROSOFT INDIA PVT LTD'.
INSERT  TABLE EPOCOMPANY.

ELSEIF STREET EQ 'USA' AND CITY  EQ 'USA' AND ZIP EQ '50000'.
EPOCOMPANY-COMPANYNAME = 'SUN MICRO SYSTEMS'.
INSERT TABLE  EPOCOMPANY.
ELSE.
EPOCOMPANY-COMPANYNAME = C.
INSERT  TABLE EPOCOMPANY.
ENDIF.

ENDFUNCTION.
```


which takes in the values in the search fields and returns a table of matching company names. The application generates the search HTML page, parses the address values submitted by the user, calls the function module, and generates the results page.

Figure 3 shows the code for function module `ZEPO_ADDRESS_LOOKUP`. As you can see, the function checks whether any asterisks (*) were entered as wildcard values. If so, three dummy company names are added to the return table `EPOCOMPANY`. If not, the function tries to match the input address information to some hard-coded values.[2] If no matches are found, a "no matches found" message is returned in the table, along with a reminder that you can retrieve a list of all company names by entering asterisks in the input fields.

Developing the Examples

My previous article covered your best options for developing complex, SAP-centric applications (writing a custom SAP function module, transaction, or report), and the easiest options available to you for developing Web applications that run exclusively in an SAP environment (writing a Business Server Pages or Web Dynpro application). But sometimes you need more control over your Web applications than BSPs or Web Dynpro will allow, or perhaps portability is an important requirement for you, or maybe you want to avoid having your Java or .NET developers learn a new development approach. In these cases, you will probably want to turn to one of the following techniques:

6. Write a custom portal component in Java and deploy it via a PAR-based iView.
7. Write a custom J2EE application and deploy it via a generic URL iView.
8. Write a custom ASP/.NET application and deploy it via a generic URL iView.

[2] The values are hard-coded for simplicity and to ensure the example will work the same on various system implementations.

This second installment of the article series covers the pros and cons of these three approaches (options 1-5 were discussed in the previous installment) as they would apply to building the example application outlined in the previous section. For your reference, the pros and cons of all eight options are summarized in a download available at www.SAPpro.com.

Option #6: Write a Custom Portal Component in Java and Deploy It via a PAR-Based iView

Portal components (option ❻ in Figure 1) are specialized Java components, built with the Portal Development Kit (PDK),[3] that run within a container on the portal server called the "portal runtime," which serves as a gateway between the user and the backend systems containing the portal content.[4] A specialized SAP EP wizard lets you define iViews based on the portal archive (PAR) files generated and deployed by the PDK. Portal components use portal-specific classes to access portal data such as user role information. These classes also expose details of the current portal theme, so your iView can tailor its output to be consistent with other iViews. Like other Java applications, portal components communicate with backend SAP systems by calling function modules via the SAP Java Connector (JCo).[5]

[3] The PDK is available for download at https://www.sdn.sap.com/sdn/downloadarea.sdn. For more on the PDK, see the article "PDK Installation and Customization for SSO Access to SAP Systems: Essential Lessons for Developers and Implementation Teams" in the November/December 2002 issue of *SAP Professional Journal*.

[4] For more information on the SAP EP architecture, see my article "Integrating SAP Transactions, Reports, and Data into Your SAP Enterprise Portal — A Guided Tour of Your Options, Which to Use, and When" (*SAP Professional Journal*, July/August 2004) and also Rizwan Uqaili's article "Designing a Solid, Lasting Landscape for Your SAP Enterprise Portal Implementation: Identifying Your Key Requirements and Understanding Your Design Options" (*SAP Professional Journal*, September/October 2004).

[5] SAP JCo is available for download from http://service.sap.com. For more on JCo, see the *SAP Professional Journal* articles "Repositories in the SAP Java Connector (JCo)" (March/April 2003); "Server Programming with the SAP Java Connector (JCo)" (September/October 2003); and "Tips and Tricks for SAP Java Connector (JCo) Client Programming" (January/February 2004).

> ✓ ***Note!***
>
> *Portal components are collections of Java components (e.g., servlets) and associated images, files, etc. Portal components are conceptually similar to servlets and Enterprise JavaBeans (EJBs)[6] in that the built-in functionality of the portal-specific classes handles many of the nasty configuration details for the component (e.g., networking, session management, user management, and scalability). Technically, portal components are subclasses of the portal runtime class com.sapportals.portal.prt.component. AbstractPortalComponent and are expected to override the main class's doContent method. When the portal server receives a request for your iView, the server calls the doContent method of the subclass and expects it to return the HTML the browser needs to render the iView.*

> ✓ ***Tip***
>
> *There is an abundance of Java code examples and standard components available for free on the Internet, which can help you quickly address common requirements and enhance your PDK-developed Java applications with features such as expandable search tabs and alerts.*

Writing portal components is a great option for teams with Java experience for writing a Web application that accesses data in both SAP and non-SAP systems. Portal components are fast because they are locally hosted on the portal server and they have full access to the portal user and theme information for easy integration into your portal. Plus, the PDK includes a rich library of predelivered SAP HTML Business (HTMLB) for Java classes (in package com.sap.htmlb) that simplify coding by automatically generating the HTML code for "SAP-looking" user interface elements. This lets Java programmers generate professional-looking Web pages — including form fields, tables, and more — without having to know much HTML or JavaScript, and the interface automatically shares the same look and feel as other SAP EP iViews.

One of the downsides of developing Web applications as portal components is Java itself — the Java development language is much more complex than ABAP, and this is especially true with regard to accessing SAP data via SAP JCo. Another downside is that portal components can run only on SAP EP,[7] so consider one of the other options — a Web Dynpro application (discussed in the previous article), a J2EE application, or an ASP/.NET application — if you are developing applications that you would like to be accessible from outside the portal.

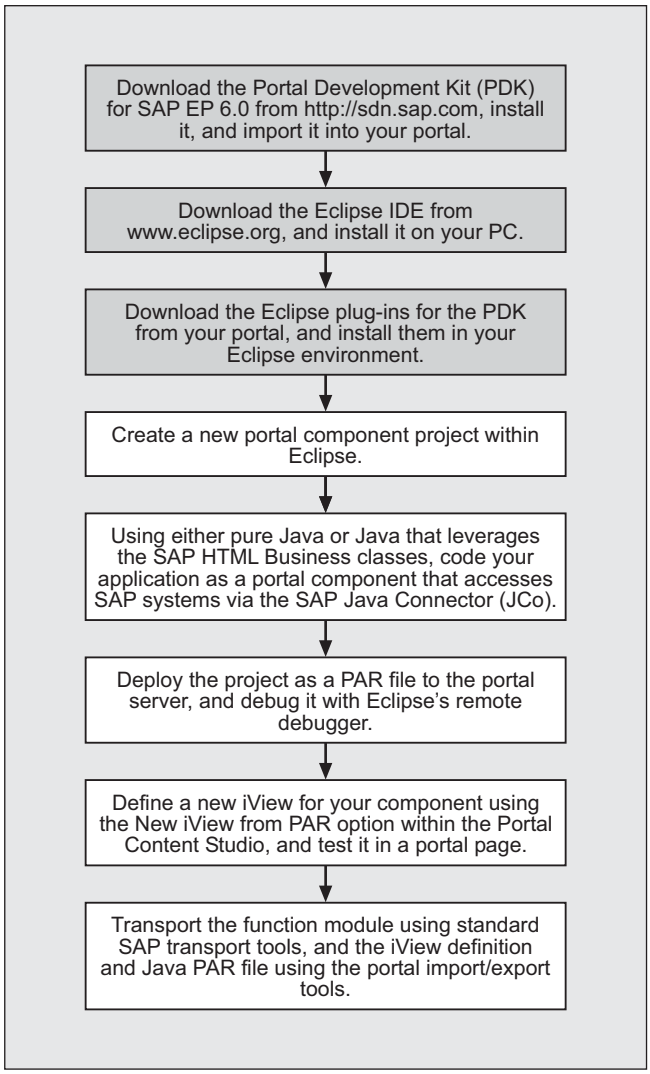
The steps required to develop portal components with the PDK are summarized in **Figure 4** (note that the items in gray are one-time setup steps). We'll create the example application both with and without using the HTMLB for Java tags to give you a clear picture of how the approaches differ. Follow these steps:

1. **Download and install the PDK for SAP EP 6.0.** The PDK is available for download for free as an SAP EP business package at https://www.sdn.sap.com/sdn/downloadarea.sdn. Simply navigate to the SAP EP download section, download the file, and follow the instructions. The basic process is to copy the PDK package into the portal import directory on your portal server (usually user/sap/<servername>/global/config/pcd/1import), import the package (select System Administration → Transport → Import), and then assign the Java Developer role (com.sap.portal.pdk.JavaDeveloper) to your user ID. Thereafter, when you log on to the portal, you'll see a Java Developer role with

[6] For more on EJBs, see the article "Persist Data for Your J2EE Applications with Less Effort Using Entity Beans with Container-Managed Persistence (EJB CMP)" in the July/August 2004 issue of *SAP Professional Journal*.

[7] Due to their reliance on portal runtime classes such as AbstractPortalComponent.

Figure 4 Writing and Deploying a Portal Component (Option 6)



documentation, code examples, and specialized tools available to you for viewing and managing portal components. You'll also see a Downloads option, where you can download the Eclipse plug-ins you'll need (more on this in a moment).

2. **Download and install the Eclipse IDE.** Download the Eclipse IDE from www.eclipse.org and install it on your PC (or your developer's PC). This should be straightforward and uneventful. Check the current SAP PDK documentation for the recommended version of Eclipse.

> ✓ ***Note!***
>
> *As explained in my July/August 2004 article, the PDK is offered for download in two forms. The PDK for SAP EP 5.0 is delivered as a set of Eclipse plug-ins that let you build portal components and run and debug them locally on a J2EE (Tomcat) server, and then transport them to your portal server. To improve team coordination and ease administration of custom portal components, SAP folded the Eclipse plug-ins into a more comprehensive business package for SAP EP 6.0. This new package — called the PDK for SAP EP 6.0 — includes administration tools and documentation, and an area where your Java developers can download the Eclipse plug-ins. In addition, the PDK for SAP EP 6.0 plug-ins have been reconfigured to deploy and debug your components on the portal server rather than locally.*

> ✓ ***Note!***
>
> *These steps are described in detail in the PDK and Eclipse installation instructions. Be sure to refer to the latest, specific instructions that come with the PDK business package and Eclipse. I've included an overview of them here just to give you an idea of what's involved.*

3. **Download and install the PDK plug-ins into the Eclipse IDE.** As installed in step 2, the generic Eclipse IDE doesn't include the portal-specific wizards, classes, menu options, and documentation you'll need to build portal components. This functionality is provided by SAP as a series of plug-ins that are downloaded and installed as part of the PDK. To access the plug-ins, log on to your portal with a user ID that has the Java Developer role described earlier and follow the menu path Java Developer → Development → Downloads. Download the Eclipse PDK plug-in, extract the ZIP file to your Eclipse plug-ins folder (usually

Figure 5 The Portal Component Code Without HTMLB

Code to Generate the Search Page

```
package com.deloitte.sapep;

import com.sapportals.portal.prt.component.*;

public class SearchForm extends AbstractPortalComponent
{
public void doContent(IPortalComponentRequest request, IPortalComponentResponse
response)
      {
response.write(" <html> <h2> <center>COMPANY NAME DETAILS</center></h2><body><form
name='comdetail' action='http://eportal:50000/irj/servlet/prt/portal/prtroot/
pcd!3aportal_content!2fcom.deloitte.epo.EPO!2fEsap' ><br><br><br> </center></h3>
</head>");
            response.write("<center><table border='7' bgcolor='grey'> ");
response.write("<tr><td colspan=2><b> SEARCH FORM</b> </td>
</tr><tr><td>Street</td><td><input type='text'name='street' value=''></td></tr>");
            response.write("<tr><td>City</td><td><input type='text'name='city'
value=''></td></tr>");
            response.write("<tr><td>ZipCode</td><td><input type='text'name='zipcode'
value=''></td></tr>");
            response.write("<tr><td><input type='submit'
value='Search'></td></tr></table></form> </center></body></html>");
      }
}
```

Code to Generate the Results Page

```
package com.deloitte.epo;

// import statements ... (omitted for printing)
public class Result extends AbstractPortalComponent
{
public void doContent(IPortalComponentRequest request, IPortalComponentResponse
response)
{
String street = request.getParameter("street");
String city = request.getParameter("city");
String zipcode = request.getParameter("zipcode");
IConnectorGatewayService cqService =
(IConnectorGatewayService)PortalRuntime.getRuntimeResources().getService(
"com.sap.portal.ivs.connectorservice.connector");
ConnectionProperties cp = new ConnectionProperties(request.getLocale(),
request.getUser());
```

under C:\eclipse\plugins), and start Eclipse to finish the installation. You must then configure the connection to your *development* portal server — for testing and debugging — by going to Window → Preferences → SAP Portal Plugin Settings.

4. **Create a new portal component project within Eclipse.** Once Eclipse is up and running, create a portal component project and add some portal components to it. The PDK includes several good tutorials that teach you how to create a portal

Figure 5 (continued)

```
IConnection client = null;
try
{
// client = JCO.createClient( "321", "morumpra","padma",
"EN","socw2saer18","P47","SPACE");
client = cqService.getConnection("R3System", cp);
IInteraction ix = client.createInteractionEx();
IInteractionSpec ixspec = ix.getInteractionSpec();
ixspec.setPropertyValue("Name", "ZEPO_ADDRESS_LOOKUP");
IFunctionsMetaData functionsMetaData = client.getFunctionsMetaData();
IFunction function = functionsMetaData.getFunction("ZEPO_ADDRESS_LOOKUP");
RecordFactory rf = ix.getRecordFactory();
MappedRecord input = rf.createMappedRecord("CONTAINER_FOR_INPUT_PARAMETERS");
IStructureFactory structureFactory = ix.retrieveStructureFactory();
input.put("STREET",street);
input.put("CITY",city);
input.put("ZIP",zipcode);
Object out = ix.execute(ixspec, input);
MappedRecord output = (MappedRecord)out;
IRecordSet rs = null;
Object result = output.get("EPOCOMPANY");
if(result != null && (result instanceof IRecordSet))
rs = (IRecordSet)result;
if(result != null)
{
response.write(" <html> <body><form name='comdetail'
action='http://eportal:50000/irj/servlet/prt/portal/prtroot/
pcd!3aportal_content!2fcom.deloitte.r10.hyd.eptraining!2fTestgood' ><br><br><br>
<head> <h3> <center>Company Details </center></h3> </head>");

response.write("<center><table border='7'><tr> <th> COMPANY NAME </th></tr> ");
String code;
while(rs.next())
{
code = rs.getString("COMPANYNAME");
response.write("<tr><td>" +code + "</td> </tr>");
      }
                        response.write("</form></table> </center></body></html>");
                  }
}
catch(Exception exe)
{
response.write(exe.getMessage());
}
}
}
```

component project and add portal components, so let's skip the details of this step and move on to the more interesting discussion: the code!

5. **Code your component.** I developed the example with and without using the HTMLB for Java classes, and I also show you two different options for developing the search/results interface — one that separates the search and results code and one that combines it. **Figure 5** shows the code for

Figure 6 The Portal Component Code with HTMLB

```
public void doProcessBeforeOutput() throws PageException
{

      Form myForm = this.getForm(); // get the form from DynPage
      GridLayout gl = new GridLayout();
      gl.setId("myGrid");

      gl.setCellSpacing(2);
      // gl.setWidth("40%");
      // gl.setDebugMode(true);
      myForm.addComponent(gl);

if(onclick.equals("NO"))
{
        TextView tv1 = new TextView("COMPANY NAME DETAILS");
        TextView tv2 = new TextView("");
        TextView tv3 = new TextView("SEARCH FORM");
    Label lstreet = new Label("Street");
    gl.addComponent(7,120,tv1);
        gl.addComponent(8,120,tv2);
        gl.addComponent(9,120,tv3);
        InputField street = new InputField("street");
        street.setValue("");
            gl.addComponent(10,121,street);
            gl.addComponent(10,120,lstreet);

        Label lcity = new Label("City");
                  InputField city = new InputField("city");
                  city.setValue("");
            gl.addComponent(12,121,city);
            gl.addComponent(12,120,lcity);
            Label lzip = new Label("ZipCode");
                    InputField zip = new InputField("zip");
                    zip.setValue("");
                        gl.addComponent(14,121,zip);
                        gl.addComponent(14,120,lzip);

            Button cdisplay = new Button("cdisplay", "Search");

            cdisplay.setOnClick("onCompany");
            gl.addComponent(16,120,cdisplay);

// create your GUI here....
}//end of if
else
{
```

the example written using pure Java. As you can see, the code is split into two components, one that defines the search page layout and a second that handles the event that occurs when a user

Figure 6 (continued)

```
      IConnectorGatewayService cqService =
(IConnectorGatewayService)PortalRuntime.getRuntimeResources().getService(
"com.sap.portal.ivs.connectorservice.connector");
      ConnectionProperties cp = new ConnectionProperties(request.getLocale(),
request.getUser());
      IConnection client = null;

      try
      {
            client = cqService.getConnection("R3System", cp);
            IInteraction ix = client.createInteractionEx();
            IInteractionSpec ixspec = ix.getInteractionSpec();

            ixspec.setPropertyValue("Name", "ZEPO_ADDRESS_LOOKUP");
            IFunctionsMetaData functionsMetaData = client.getFunctionsMetaData();

            IFunction function =
functionsMetaData.getFunction("ZEPO_ADDRESS_LOOKUP");
                              RecordFactory rf = ix.getRecordFactory();

            MappedRecord input =
rf.createMappedRecord("CONTAINER_FOR_INPUT_PARAMETERS");
            IStructureFactory structureFactory = ix.retrieveStructureFactory();

            input.put("STREET",pstreet);
            input.put("CITY",pcity);
            input.put("ZIP",pzip);
            Object out = ix.execute(ixspec, input);
            MappedRecord output = (MappedRecord)out;

            IRecordSet rs = null;
            Object result = output.get("EPOCOMPANY");

            if(result != null && (result instanceof IRecordSet))
            rs = (IRecordSet)result;

            if(result != null)
            {
                  String code;
                  String name;
                  int i=12;
                  int y=120;

                  Button lcom= new Button("lcom", "COMPANY DETAILS");
                  //Label lcom = new Label("COMPANY DETAILS");
```

(continued on next page)

clicks on the Search button and displays the results. **Figure 6** shows the search page layout and event handling code consolidated into a single component that uses HTMLB for Java classes.

Figure 6 (continued)

```
                    gl.addComponent(10,121,lcom);

                    while(rs.next())
                    {
                    code = rs.getString("COMPANYNAME");
      Label city = new Label(code);
      gl.addComponent(i,121,city);
      i++;
                    }
              } //try
} //if-else
} // doProcessBeforeOutput
```

See the difference in code complexity? HTMLB keeps the code very clean and maintainable, and requires almost no knowledge of HTML or JavaScript. The pure Java code, on the other hand, is more unwieldy and requires additional HTML tags for the table definition and form field names. Let's take a look at the output: **Figure 7** shows the results of the pure Java code listed in Figure 5; **Figure 8** shows what the HTMLB-based code in Figure 6 produces. As you can see, the user interface generated by the HTMLB classes is more professional looking, automatically adopts an SAP look and feel, and offers additional functionality in many cases (e.g., paging between tables, which would otherwise take hours of development to reproduce). The pure Java code, on the other hand, requires the manual addition of stylesheets that can be difficult to coordinate.

The only downside to using HTMLB — aside from the investment in learning how to use it — is that it limits your control over the underlying HTML code and the ability to inject your own JavaScript routines into the code. So if you want more control, and are prepared to sacrifice the benefits of code automation, you're better off sticking with pure Java.

6. **Deploy and debug your component.** Deploying your component is easy, but it can be confusing for new developers. The idea is to have Eclipse generate a portal archive (PAR) file and transfer this file to the portal server so you can use it to define new iViews. The PAR file includes a compiled version of the Java classes that make up your component, as well as one or more deployment descriptor files that guide the portal's J2EE runtime in loading and running your component. To deploy the component, choose the menu item Export → PAR File and follow the wizard instructions that appear.

Once the PAR file is deployed to the portal server, you can debug it in two ways:

- **From within Eclipse:** Expand the Portal-INF folder within your Eclipse project, double-click on the portalapp.xml file, and click on the name of your portal component class under the components section. You'll see your component launch within the preview pane.
- **On the portal server:** Log on to the portal with a user ID that has the Java Developer role and navigate to Java Development → Component Inspector. Pull up your portal component class (which should now be in the list) and click on Start next to the class name.

Figure 7 Search and Results Pages Generated by the Non-HTMLB Code

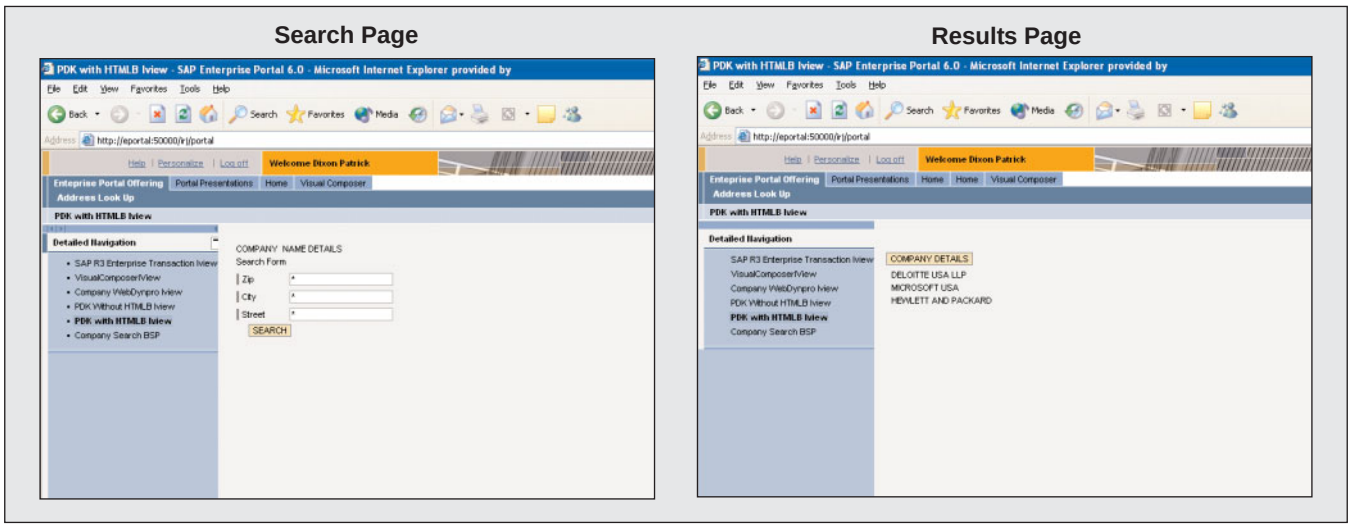
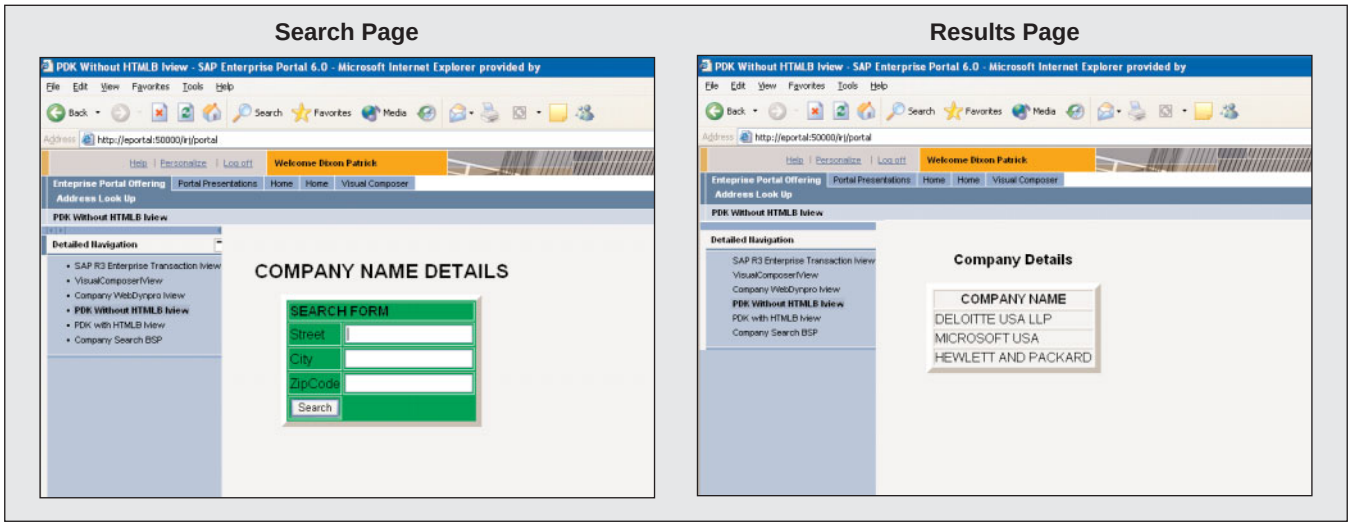


Figure 8 Search and Results Pages Generated by the HTMLB-Based Code



The first method is by far the easier of the two since you're already working in Eclipse. If the component doesn't work as expected, you can debug your code using Eclipse's remote debugger. The debugger can connect to an instance of your portal component class running on the portal server and let you step through the code line by line. More information on how to deploy and debug your portal components is available at http://sdn.sap.com. The SAP EP forums are particularly useful.

7. **Define a new iView for your portal component and test it.** Once you've deployed and debugged your portal component, you can build an iView based on the component and test it by integrating it into a portal page. In the portal administration tool, follow the menu path Content Administration → Portal Content, right-click on the PCD folder to which you want to add the iView, select New from Portal Archive → iView from the context menu, and define the iView by following the wizard's instructions. You can then assign the iView

to a page and test the component as usual. (Refer to my July/August 2004 article for more details on defining iViews.)

8. **Transport your application.** Your function module, iView definition, and PAR file must all be transported. The function module is transported using standard SAP transport tools (i.e., the SAP Change and Transport System). The iView definition and PAR file are both transported via the standard portal import/export tools.

In summary, developing portal components is a great option for SAP teams with extensive Java experience, who don't mind giving up code portability in exchange for performance, accessibility to portal server runtime data, and the ability to leverage HTMLB for Java. The Eclipse development platform is stable and vendor-neutral, and is easy for Java developers to learn (although not ABAP developers). Portal components can be developed either using pure Java and HTML, which provides you with more advanced control over the code, or using SAP-proprietary HTMLB for Java classes, which simplifies the code and enables you to instantly gain a professional SAP look and feel.

> ✓ ***Note!***
>
> *For details on installing and developing iViews with the PDK, see the session "Basics of Java iView Development" available at http://sdn.sap.com. Search for the session title to locate it.*

Option #7: Write a Custom J2EE Application and Deploy It via a Generic URL iView

Next on our journey of custom development options is to write a standard J2EE application (option ❼ in Figure 1) — i.e., as a set of servlets or Java Server Pages (JSPs). Both servlets and JSPs are staples in most Java developer toolboxes, and both can be developed using any one of a number of Java development tools, including SAP NetWeaver Developer Studio[8] (available from http://sdn.sap.com), IBM VisualAge for Java, and Borland JBuilder. If you have Java developers on hand and want to build a custom Web application that can access data in both SAP and non-SAP systems, chances are they'll be most excited about this approach since they'll have to learn the least to use it.

> ✓ ***Note!***
>
> *While Figure 1 depicts the custom servlet (option ❼) running on a separate server, both SAP EP 5.0 and 6.0 can host these applications on its J2EE Engine. Regardless of where you host your servlet, though, you'll integrate your deployed application as a generic URL iView in your portal server (since the portal server treats it as a foreign, external application). See my July/August 2004 article for details on creating generic URL iViews.*

The main advantage of developing J2EE applications instead of portal components or Web Dynpro for Java applications[9] is that you can run them on any Java server you like, and your Java developers don't have to learn the specifics of the portal component or Web Dynpro development models. And since you're developing pure, unadulterated Java (and not using proprietary extensions like HTMLB), you also have complete freedom as to which Java development tool to use. Finally, compared to Web Dynpro for Java, you gain complete control over the user interface code, although you lose the ability to automatically

[8] For more on SAP NetWeaver Developer Studio, see the article "Get Started Developing, Debugging, and Deploying Custom J2EE Applications Quickly and Easily with SAP NetWeaver Developer Studio" in the May/June 2004 issue of *SAP Professional Journal*.

[9] Web Dynpro for Java was covered in the first installment of this two-part article series, which appeared in the March/April 2005 issue.

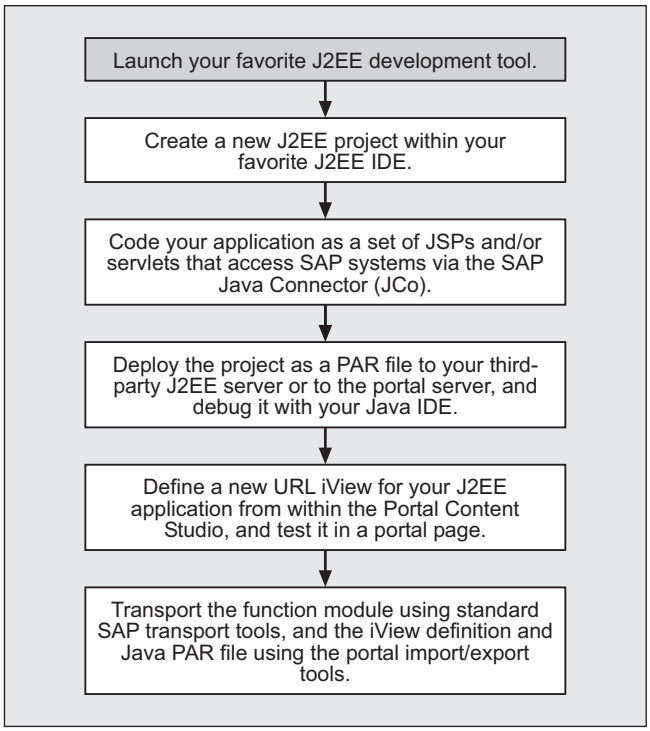
access user, theme, and other information provided by the portal server runtime. In the end, this approach is only appropriate for teams with extensive Java experience who want maximum code portability and are not afraid of maintaining the considerable amount of extra code involved in "going it alone."

> ✓ ***Note!***
>
> *Remember that there is an abundance of code examples and standard components available for free on the Internet that can help save you a lot of time when addressing common application requirements.*

> ✓ ***Note!***
>
> *Also remember that compared to ABAP, a downside to all of these Java options is that accessing SAP data by calling function modules with SAP JCo and formatting it for output is far more complex than accessing it with SQL in an ABAP program.*

The process for writing and deploying a J2EE application is almost identical to the process described previously for writing and deploying a portal component without using HTMLB for Java. **Figure 9** shows an overview of the steps involved (again, note that the gray step is a one-time step). The only real differences, aside from the specifics of the particular IDE you are using, are that you may be using a third-party J2EE server if you are using a non-SAP development tool to develop the application, and you deploy the J2EE application as a generic URL iView rather than a PAR-based iView (see my July/August 2004 article for details on creating generic URL iViews). The code for the example J2EE application would be the same as the non-HTMLB code shown in Figure 5 for the example portal component, except for one difference: the superclass would be HTTPServlet (if we chose to implement our demo as a servlet) instead of AbstractPortalComponent. The output would look exactly like the screens shown in Figure 7.

Figure 9 Writing and Deploying a J2EE Application (Option 7)



Option #8: Write a Custom ASP/.NET Application and Deploy It via a Generic URL iView

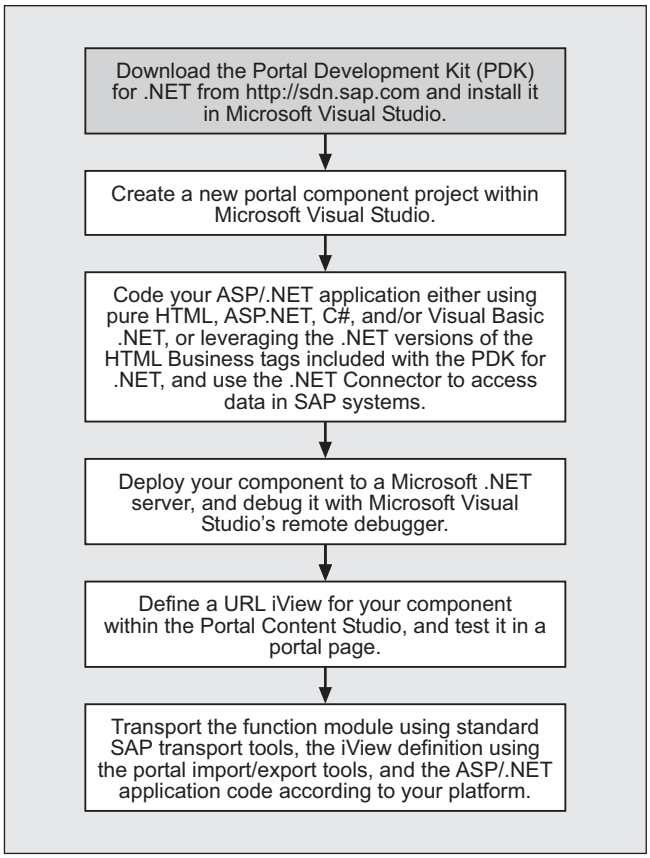
So what about SAP teams that have strong familiarity with Microsoft tools like Microsoft Visual Studio and languages like Active Server Pages (ASPs), ASP.NET, Visual Basic .NET, or C#? Can these developers build SAP EP applications using their existing expertise? Yes, they can (see option ❽ in Figure 1).

SAP has two offerings for Microsoft developers. First is the SAP .NET Connector for calling function modules and BAPIs on SAP systems. The .NET Connector, which supersedes the SAP DCOM Connector, is analogous to JCo and can be used with

both .NET and "non-.NET" versions of Microsoft development languages.[10] Second is the PDK for .NET, containing a library of SAP-specific DLLs, which is a recently released version of the PDK for SAP EP 6.0 that includes .NET versions of the portal classes, documentation, and wizards included with the standard version, including a set of HTMLB for .NET subroutines that simplify coding by automatically generating the HTML code for "SAP-looking" user interface elements. Just as the PDK for SAP EP 6.0 plugs into the Eclipse IDE, the PDK for .NET plugs into Microsoft Visual Studio. Unlike the PDK for SAP EP 6.0, however, the .NET version is not released as a business package that is installed on the portal server — you download it from http://sdn.sap.com and install it directly into your Microsoft Visual Studio tool. For convenience, the PDK for .NET includes the .NET Connector.

> ✓ ***Note!***
>
> *Since the .NET Connector is useful for more than just portal development, it is also available as a standalone download from http://sdn.sap.com.*

Figure 10 Writing and Deploying an ASP/.NET Application (Option 8)



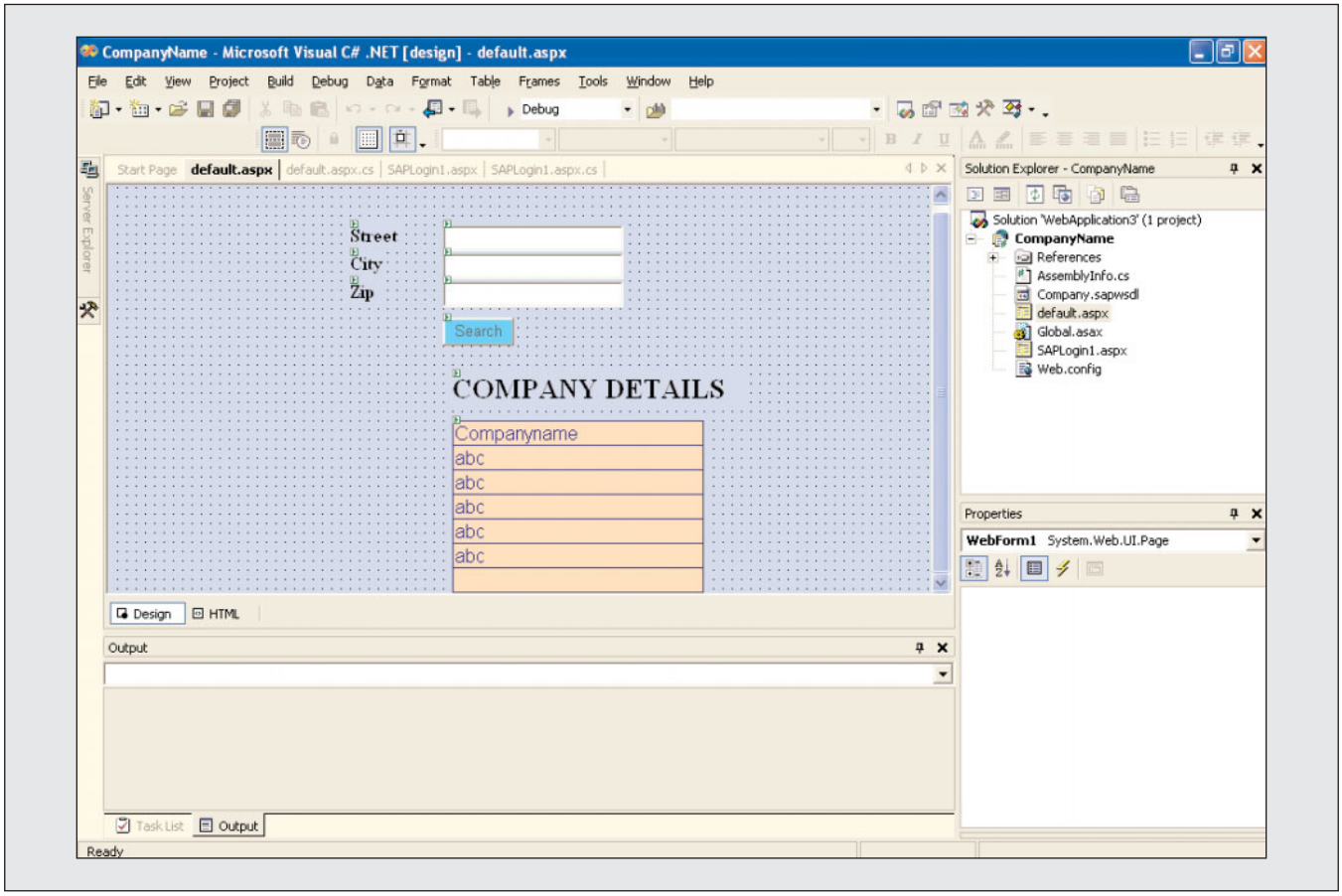
Since the Java and .NET tools parallel each other so closely, it won't surprise you to learn that developing applications with the .NET tools is almost identical to developing applications with their Java counterparts in terms of complexity, pros, and cons. Like Java applications, .NET applications can be easily integrated into the portal by using the generic URL iView template, and like Java applications, there is an abundance of code examples and standard components available for free on the Internet, which will save you a lot of time in addressing common requirements.

There are a few caveats, however. First, you need a Microsoft Internet Information Server (IIS) to host .NET applications — you can run Microsoft IIS on your portal server (if your portal runs on Windows) or you can use another Microsoft IIS in your landscape. Second, SAP's support for .NET seems (in my opinion) to have been added purely for the sake of compatibility. SAP has committed to ABAP and Java as a long-term, strategic focus, and (in my opinion) you can most likely expect the latest and greatest features to first become available to ABAP and Java developers, and only thereafter trickle down to the .NET stack.

The process for writing and deploying an ASP/.NET application is similar to the process described previously for writing and deploying a portal component. **Figure 10** shows an overview of the steps involved. As you can see, similar to portal components, you can create the example application both with and without using HTMLB tags. The only real differences between the process for portal components and the process for ASP/.NET components, aside from

[10] For more on the .NET Connector, see the article "SAP .NET Connector for C# Programmers" (*SAP Professional Journal*, July/August 2003).

Figure 11 The Design of the Search/Results Page in Microsoft Visual Studio



the specifics of the development tool in use, are that you do not load code to the portal server, and you deploy the ASP/.NET application as a generic URL iView rather than a PAR-based iView (for details on creating generic URL iViews, see my July/August 2004 article).

Figure 11 shows the design of the search and results pages in Microsoft Visual Studio. As you can see, they have been combined into a single page.

Figure 12 shows the ASP.NET code that handles the event that occurs when a user clicks on the Search

Figure 12 The Example Developed Using ASP.NET, HTMLB for .NET, and the .NET Connector

Event Handler Code to Perform the Address Lookup

```
private void Button1_Click(object sender, System.EventArgs e)
{// Declare parameters here

Company proxy = new Company();
try
```

(continued on next page)

Figure 12 (continued)

```
{
proxy.Connection =  SAP.Connector.SAPLoginProvider.GetSAPConnection(this);
  // Call methods here
 proxy.Zepo_Address_Lookup(this.TCity.Text,this.TStreet.Text,this.TZip.Text, ref
this.zcompnamesTable1);
  // Now update Data Bindings. On WinForms this will be automatic, on WebForms
call the following line
  this.DataBind();
}
```

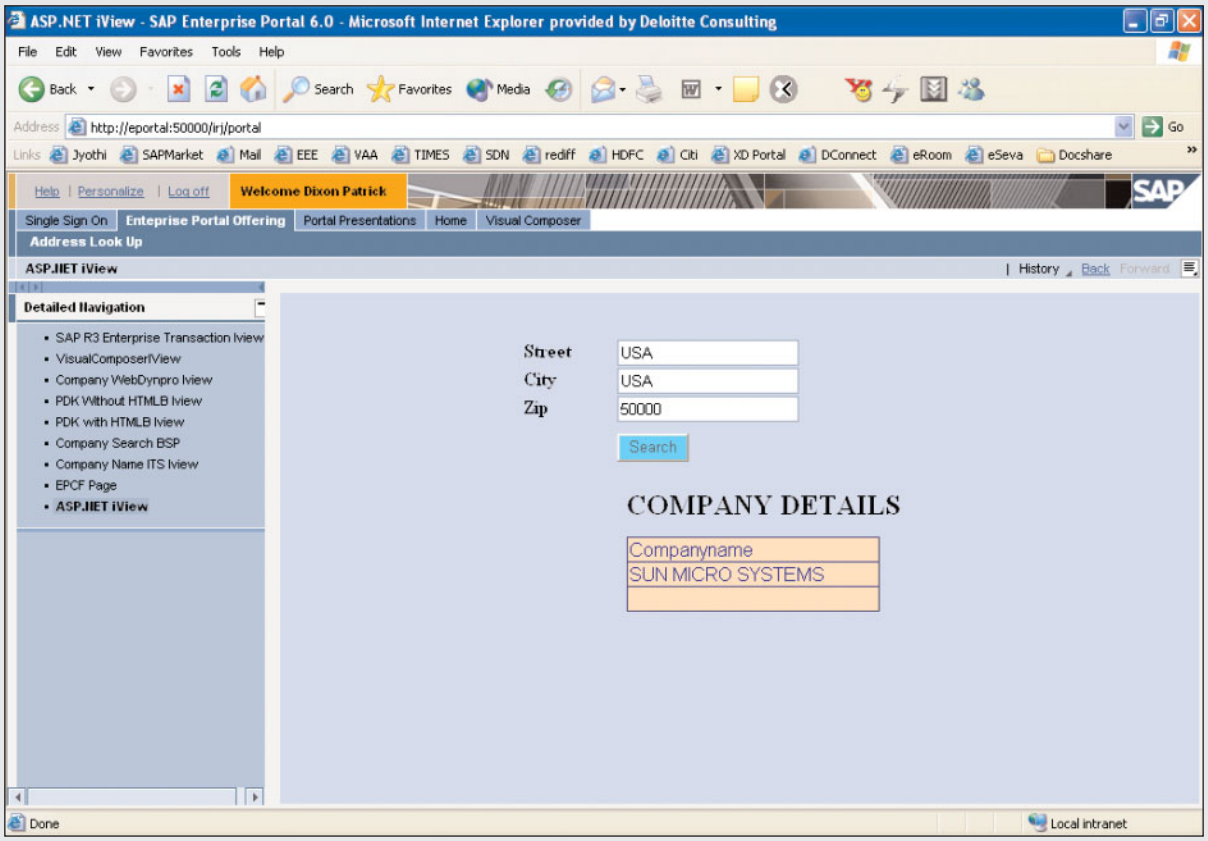
Code for the Search/Results Page

```
<%@ Page language="c#" Codebehind="default.aspx.cs" AutoEventWireup="false"
Inherits="WebApplication3.WebForm1" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
      <HEAD>
            <title>WebForm1</title>
            <meta name="GENERATOR" Content="Microsoft Visual Studio .NET 7.1">
            <meta name="CODE_LANGUAGE" Content="C#">
            <meta name="vs_defaultClientScript" content="JavaScript">
            <meta name="vs_targetSchema"
content="http://schemas.microsoft.com/intellisense/ie5">
      </HEAD>
      <body MS_POSITIONING="GridLayout" bgColor="inactivecaptiontext">
            <form id="Form1" method="post" runat="server">
                  <asp:TextBox id="TCity" style="Z-INDEX: 101; LEFT: 288px;
POSITION: absolute; TOP: 64px" runat="server"></asp:TextBox>
                  <asp:TextBox id="TStreet" style="Z-INDEX: 102; LEFT: 288px;
POSITION: absolute; TOP: 40px" runat="server"></asp:TextBox>
                  <asp:TextBox id="TZip" style="Z-INDEX: 103; LEFT: 288px;
POSITION: absolute; TOP: 88px" runat="server"></asp:TextBox>
                  <asp:Label id="Label1" style="Z-INDEX: 104; LEFT: 208px;
POSITION: absolute; TOP: 40px" runat="server"
                        Font-Bold="True">Street</asp:Label>
                  <asp:Label id="Label2" style="Z-INDEX: 105; LEFT: 208px;
POSITION: absolute; TOP: 64px" runat="server"
                        Font-Bold="True">City</asp:Label>
                  <asp:Label id="Label3" style="Z-INDEX: 106; LEFT: 208px;
POSITION: absolute; TOP: 88px" runat="server"
                        Font-Bold="True" Width="30px">Zip</asp:Label>
                  <asp:Button id="Button1" style="Z-INDEX: 107; LEFT: 288px;
POSITION: absolute; TOP: 120px" runat="server"
                        Text="Search" ForeColor="#404040"
BackColor="#80FFFF"></asp:Button>
                  <asp:DataGrid id=DataGrid1 style="Z-INDEX: 108; LEFT: 296px;
POSITION: absolute; TOP: 208px" runat="server" DataSource="<%# zcompnamesTable1
%>" Width="216px" ForeColor="Blue" ShowFooter="True" Font-Names="Arial"
BackColor="#FFE0C0" BorderColor="#0000C0">
                  </asp:DataGrid>
```


Figure 12 (continued)

```
                    <asp:Label id="Label4" style="Z-INDEX: 109; LEFT: 296px;
POSITION: absolute; TOP: 168px" runat="server"
                        Width="248px" Font-Bold="True" Font-Size="Large">COMPANY
DETAILS</asp:Label>
            </form>
      </body>
</HTML>
```

Figure 13 The Search/Results Page Generated by the ASP.NET Code



button and calls the address lookup function module `ZEPO_ADDRESS_LOOKUP` (via the .NET Connector), and the code that generates the layout.

Figure 13 shows the search/results page deployed via a generic URL iView, and the results generated by an example company lookup.

> ✓ ***Note!***
>
> *You may notice that, in contrast to Web Dynpro, the screen designer within Microsoft Visual Studio lets you modify the code it generates. This is why Figure 12 includes HTML tags (<asp: …> tags).*

In the end, this approach is most appropriate for SAP teams with strong ASP/.NET development skills, and very little in terms of ABAP and Java skills. In the long term, however, the Java options will be more strategic due to SAP's commitment to this direction, and SAP Web AS's built-in ability to host Java applications across platforms.

Armed with a solid understanding of your options for developing custom applications for your SAP EP implementation — from developing complex SAP-centric applications (covered in the previous installment of this article series) to developing highly customized, cross-platform Java applications and applications based on ASP/.NET — you are ready to make the right choices for your own environment. As you build your custom applications and deploy them via iViews, you will probably find that some of the deployments would be much more efficient if you could somehow link the iViews together — for example, if a value entered in one iView could automatically be added to another iView. In the next section, I show you how SAP EP's "eventing" feature enables your iViews to communicate with one another on the client side, enabling you to build complex applications without overloading the portal server.

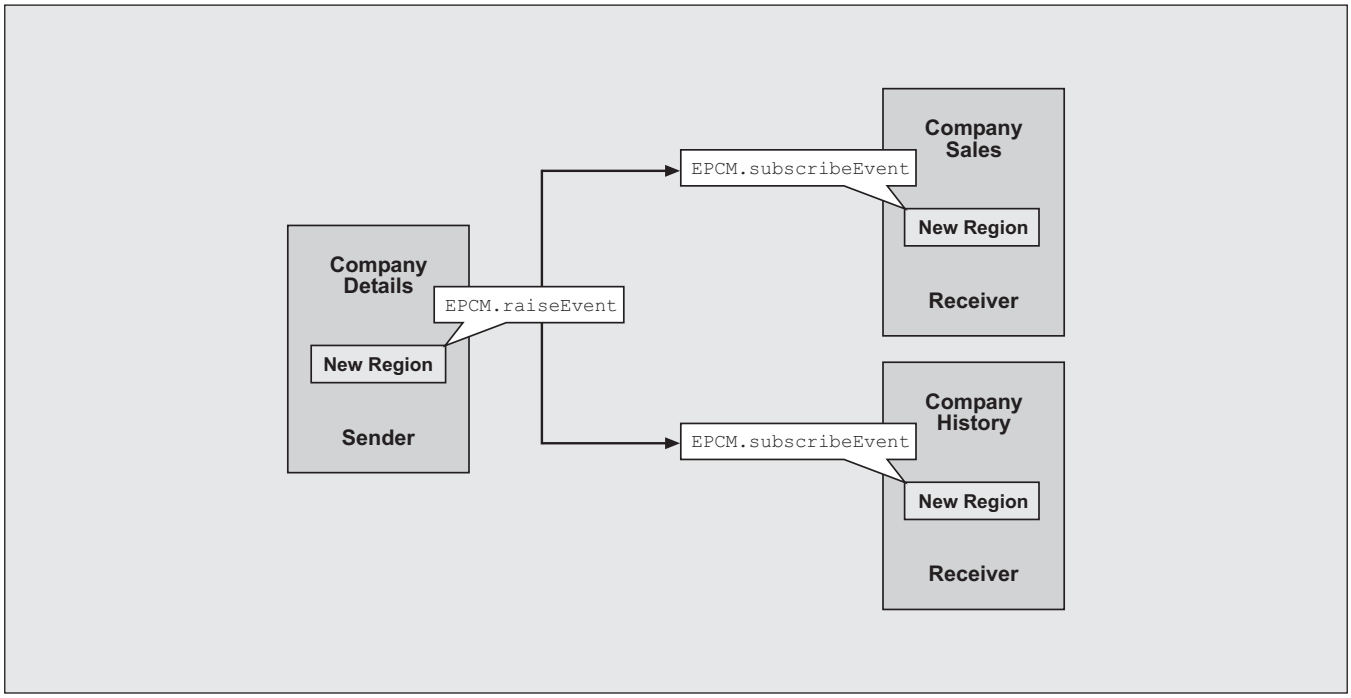
Enhancing Your SAP EP Implementation with Eventing

Eventing is a powerful feature unique to SAP EP with which your custom-developed iViews can "talk" to each other — when a user performs an action in one iView, all other iViews on the page can be notified and take appropriate action. In practice, this means that one iView (the sender) can drive the display of data in other (receiver) iViews. For example, when a user selects a new region from a Region dropdown menu in one portal iView, it can send an event containing the name (or value) of the newly selected region to all chart iViews on the page, which can react by refreshing their charts to reflect data for that region. This avoids users having to update the region for each graph individually. For this reason, eventing always becomes a project "must have" when I demo it to clients.

> ✓ ***Note!***
>
> *Detailed documentation on eventing is available at https://media.sdn.sap.com/html/submitted_docs/PDK_for_dotNET_10/How%20to%5CClient%20Side%20Events.htm. You can also find more detailed documentation and examples in the SAP Enterprise Portal Development Kit (PDK) section of the SAP NetWeaver Developer Studio help.*

Here's how it works (refer to **Figure 14**): The underlying mechanism of eventing is a library of JavaScript code called the Enterprise Portal Client Manager (EPCM). The EPCM methods provide access to the Enterprise Portal Client Framework (EPCF), which enables communication between iViews. The EPCM library is automatically downloaded with your portal pages, and makes a scripting object called EPCM globally available to all JavaScript code within your custom iView. To start receiving event notifications (when a new region is selected for a company in an iView, for example) all interested iViews (e.g., those containing company sales and company history information) must call the EPCM `SubscribeEvent` method to register themselves as listeners for particular event types. You only

Figure 14 *How Eventing Works*



> ✓ ***Note!***
>
> *For security reasons, the EPCF only allows events to be transmitted between iViews that originate from servers in the same domain.*

> ✓ ***Note!***
>
> *The fact that you have to insert code into the sending and receiving iViews is why this technique is (for the most part) only useful for custom iViews.*

need to call `subscribeEvent` once for a given type of event in order to receive event notifications. An unlimited number of iViews can subscribe to a given event type.

Here's an example of a `subscribeEvent` call:

```
EPCM.subscribeEvent(
"urn:com.dixon.RegionSelectiView",
"RegionChanged", onRegionChange );
```

The first parameter identifies the namespace of the event, the second is the event name, and the third is the name of the JavaScript method within your code that you want invoked when an event of that event type is raised. You are free to assign whatever values you want to the namespace and event name parameters — they just must match the values you specify when you raise the event. The role of the namespace is to prevent conflicts between iViews that might use the same event names.

Figure 15 Complete Example Eventing Code

Code Placed Within the Chart iViews

```
<SCRIPT language ="JavaScript">

EPCM.subscribeEvent( "urn:com.dixon.RegionSelectioniView", "RegionChanged",
onRegionChange );

function onRegionChange(eventObj)
{
var newRegion = eventObj.dataObject;

//Use the new region value to automatically refresh the chart—the code will
//depend on your design
document.elements.region = newRegion;
document.elements.forms[0]submit();
}
</SCRIPT>
```

Code Placed Within the Region Selector iView

```
<SCRIPT language ="JavaScript">

function doRegionChanged(newRegion)
{
   //raise the event, passing the new region value
   EPCM.raiseEvent( "urn:com.dixon.RegionSelectioniView", "RegionChanged",
newRegion );
}

</SCRIPT>

//The region drop down menu which triggers the JavaScript handler upon a value
//change
<SELECT Name="RegionDropDownMenu" onchange=" doRegionChanged(this.value)">
<OPTION value="NorthEast">North East
<OPTION value="SouthEast">South East
<OPTION value="MidWest">Mid West
…
</SELECT>
```

The next step is to raise the event by calling `EPCM.raiseEvent` to transmit an event notification when desired. The example shown in **Figure 15** is:

```
EPCM.raiseEvent(
"urn:com.dixon.RegionSelectioniView",
"RegionChanged", newRegion );
```


The first and second parameters should look familiar — they identify the event. The third parameter is the newly selected region.

The rest of the code in Figure 15 is relatively self-explanatory. The process starts when a user changes the value of `RegionDropDownMenu`. The browser triggers the `onchange` event, which we've coded to call the `doRegionChanged` JavaScript function, passing the value of the new region (note that you can pass multiple values by passing an array to `raiseEvent`). The function raises our event and returns control to the user. Behind the scenes, the EPCM object retrieves the list of subscribers for that event type and calls the requested handler function — in this case, we've asked the system to call `onRegionChange`. `onRegionChange` receives a standard event object containing the new region in its `dataObject` attribute. We use the `newRegion` value to refresh the chart by loading its value into an existing hidden field and resubmitting a form within the iView (this part of the code will differ depending on how your iView is coded).

> ✓ ***Tip***
>
> *Additional, detailed eventing examples are included with the Portal Development Kit (PDK), available for download at https://www.sdn.sap.com/sdn/downloadarea.sdn.*

In summary, the main benefit of eventing is that you can integrate data and applications from disparate systems that might otherwise be very expensive to integrate. Users enjoy a seamless experience and are usually unaware of the underlying data sources. You can eventing-enable any iView that allows you to modify its HTML and add the requisite JavaScript calls. The browser must have JavaScript enabled for eventing to work.

Conclusion

In light of SAP Web AS support for Java, .NET, and ABAP, and the release of myriad new development tools like SAP NetWeaver Developer Studio, the Portal Development Kit, and the Visual Composer, SAP teams now have a huge menu of options for developing custom applications. These options have emerged both to support enterprises with vastly different skill sets and to meet the need for more "Web-friendly" development models than traditional SAP Dynpro programming.

This article has concluded a whirlwind tour of your available options, and this series has hopefully provided you with a solid apples-to-apples understanding of how the available options differ in terms of prerequisites, development process, flexibility, and code complexity.

My recommendation for teams with extensive ABAP experience is to (for now) continue to develop complex, SAP-centric applications using traditional SAP module pools and reports and deploy them via SAPGUI for HTML iViews, until Web Dynpro for ABAP is released and matures. Web Dynpro for ABAP should then be strongly considered for Web-only applications,[11] since it will offer a much more native look and feel.

Microsoft shops should continue to use .NET Connector to access SAP systems, and start using HTML Business for .NET within the PDK to simplify their code and give their applications an SAP look and feel. Microsoft shops should also look into learning ABAP or Java, since these will be more strategic going forward from an SAP perspective.

Java teams should strongly consider using Web Dynpro for Java whenever possible, but will probably find the lack of customizability over the user interface (HTML) frustrating. In this case, developing portal components using HTML Business for Java classes is

[11] That is, applications that don't need to be accessed by SAPGUI users as well.

preferable to developing traditional J2EE applications, which require you to maintain a significant amount of additional code on your own. While using Web Dynpro for Java and portal components sacrifice application portability, the benefits far exceed the costs in most cases — applications require significantly less code, are more maintainable, and gain a standard SAP look and feel, and with portal components you also gain access to portal runtime variables and enhanced performance due to hosting within the portal server itself.

Patrick Dixon is a manager with Deloitte specializing in SAP Enterprise Portal and Master Data Management (MDM). He has over five years of experience Web-enabling and integrating SAP systems with SAP Enterprise Portal, SAP Internet Transaction Server (ITS), SAP Exchange Infrastructure (XI), and Plumtree Software solutions. Patrick was a key speaker on portal implementation and content integration at the SAP BW and Portals 2004 and SAP Portals 2003 events. Prior to moving to the US in 1996, he was a freelance consultant in Great Britain. Patrick graduated from Bath University in 1983 with a master's degree in computer simulation, and subsequently obtained an MBA from City University, London. You can reach him at padixon@deloitte.com.

