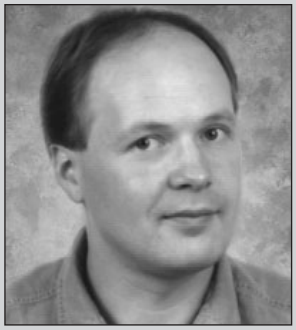
Gain a Real-World Understanding of How Your Applications Will Operate on a New Platform — Porting a J2EE Application to SAP Web Application Server

Volker Stiehl



Volker Stiehl joined Siemens in 1992, where he initially focused on server-side architectures, and then on J2EE-based integration and the integration of SAP R/3 systems with other legacy systems and databases. Currently Volker is an SAP NetWeaver consultant with special expertise in SAP Web Application Server, SAP XI, and SAP xApps.

(complete bio appears on page 92)

It is no secret that companies are cost-driven — if there is an opportunity to optimize your organization's efficiency, you have to seize it. A surefire way to reduce costs is to minimize application development time. The J2EE (Java 2 Platform, Enterprise Edition) standard was designed to address the challenge of building complex, business-critical applications with minimal effort. It takes over the ugly parts of developing business software, like transaction management and security, so that developers can concentrate on implementing business processes. And, as an industry standard, applications created using the J2EE architecture are almost universally supported, so you don't have to write different versions of the same application for different platforms.

While the J2EE standard can significantly reduce development time, in the past you still needed to maintain one type of server (e.g., BEA WebLogic, IBM WebSphere, Oracle Application Server, or the open source JBoss or JOnAS) for running your J2EE applications, and another (e.g., SAP Basis 4.6C or 6.10) for running your SAP-based business applications — until now. SAP Web Application Server (SAP Web AS) 6.30, and now 6.40, incorporate version 1.3 of the J2EE standard, so that you can use a single platform for developing and running your ABAP and J2EE applications, reducing your development and maintenance costs and optimizing the performance of J2EE applications that are already connected to your SAP systems.

This is great news for developing applications going forward, but what about the J2EE applications you've already developed on other platforms? Fortunately, the inherent platform independence of J2EE-based applications means that you can easily port them to a new server like SAP Web AS 6.30 or 6.40. Although this makes it

tempting to migrate all of your applications to the new server right away, it's best to start with one so you can accurately assess how the server will respond to your unique real-world conditions. Using an application migration I performed at my organization as an example, this article shows managers, decision makers, technical architects, and developers how to port a J2EE-based application from BEA WebLogic 6.1 to SAP Web AS 6.30, and details the lessons learned along the way. I'll also point out some potential problem areas and provide recommendations for ensuring a successful J2EE migration.[1] And I will outline the tools and best practices that will help you ensure the future portability of your J2EE applications.

> ✓ ***Note!***
>
> *While the example migration discussed in this article focuses on SAP Web AS 6.30, the Java capabilities are relatively unchanged in SAP Web AS 6.40, so the discussions and lessons in this article are valid for both releases.*

And even if you have no immediate plans to migrate to SAP Web AS, or are currently running your J2EE applications on a server other than BEA WebLogic, keep reading. By the end of this article, you will be able to evaluate whether it makes sense to port your J2EE applications to SAP Web AS, regardless of the platform you are starting from, and you will have a solid understanding of what to consider, how to start the porting project, what steps to take, and which tools to use to make the migration as cost-effective as possible.

Before we jump right into the details of porting a J2EE application to SAP Web AS, however, let's take a step back and review the J2EE application architecture.

[1] Note that once a customer has successfully ported a J2EE application to SAP Web AS, the application can be "SAP Web AS certified." For details, visit **www.sap.com/icc** or send an email to **icc@sap.com**.

Understanding the J2EE Application Architecture

Companies require a wide variety of applications for day-to-day business operations, which can add up to significant application development efforts. While standard software like SAP R/3 can go a long way toward minimizing your application development costs, it rarely covers 100% of your needs. Fortunately, SAP provides tools like the ABAP Workbench that make it easy to add functional extensions to your applications. Over time, however, business requirements have become more and more complex, in turn increasing application development time and pushing the limits of existing application development environments. The J2EE standard was designed to address this challenge by easing the development of complex Java applications.

J2EE is a distributed, multi-tiered application model. J2EE applications are generally distributed over the front tier (which displays information to the user), the middle tier (which implements the business logic), and the persistence tier (which houses the data used by the application), and consist of the components specified in the J2EE standard, in particular the JavaServer Pages (JSP), servlet, and Enterprise JavaBeans (EJB) technologies (see **Figure 1**).

The J2EE application I will use to demonstrate the porting process in the article is a Web-based application called Click4License (C4L) that was designed to centrally manage software licensing information for the entire company. Like many J2EE solutions, it requires access to several data sources — in this case, an LDAP server where every employee is uniquely identified, a relational database for storing the master data for users, computers, and licenses, a server for scripted software, a server for unscripted software, and a server for sending workflow emails to users, IT staff, heads of cost centers, etc.

C4L is a typical implementation of a multi-tiered J2EE 1.3-based Web application. It was created with the widely used open source framework Apache Struts, and makes extensive use of J2EE standard components, including JSPs, servlets, and EJBs, as

Figure 1 Data Flow in a Typical J2EE Application

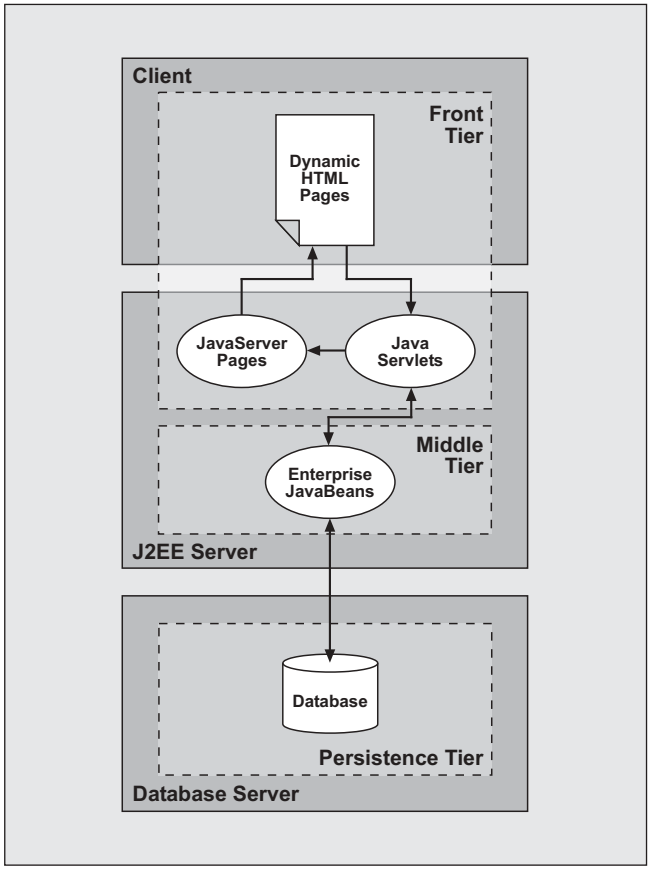
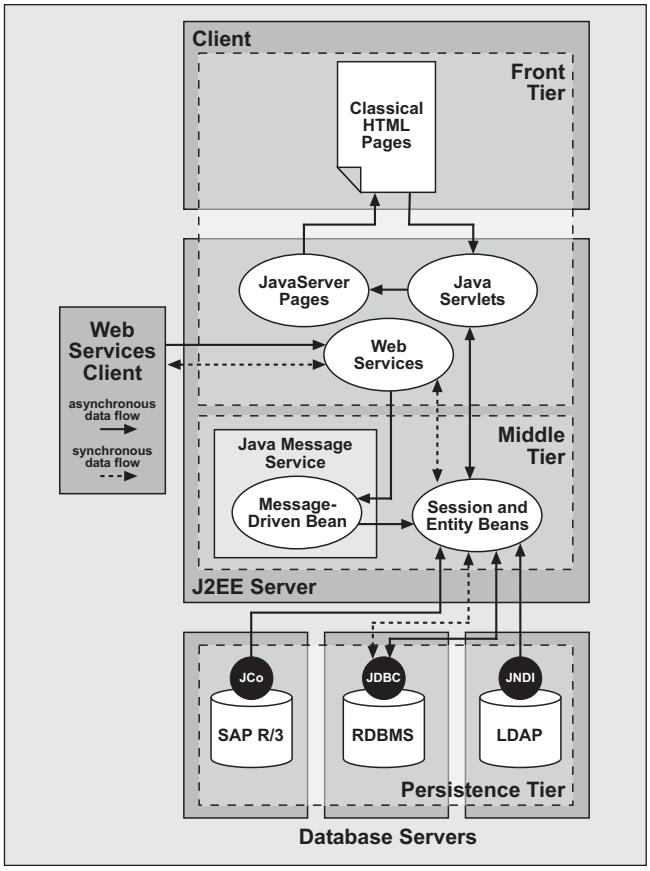


Figure 2 Data Flow in the Example Application



well as the Java Message Service (JMS). As you can see in **Figure 2**, the front tier uses classical HTML pages generated by JSPs and servlets to represent the user interface in a Web browser, along with a Web services[2] interface to upload and download master data and retrieve license information. The business logic is implemented in the middle tier using EJB technology — specifically, 31 stateless session beans, 26 container-managed entity beans, and, as you can see, 1 message-driven bean.[3] Access to the backend systems in the persistence tier is realized by the Java Naming and Directory Interface (JNDI) for the LDAP server, the SAP Java Connector (JCo)[4] for the SAP systems, and Java Database Connectivity (JDBC) for the relational database.

To run the C4L application, we needed a server that would support not only all of these technologies, but also a large number of users (12,000), computers (38,000), and licenses (250,000), while ensuring high availability and scalability. At the time we developed the C4L application (in early 2002), SAP supported J2EE 1.2, but our application required J2EE 1.3, for the EJB components in particular. For this reason,

[2] While SAP Web AS 6.30 and 6.40 support Web services, it is not part of the J2EE 1.3 specification itself. Web services will be part of the J2EE 1.4 specification, however.

[3] There are three types of EJBs: session beans for the business logic, entity beans for persisting business data, and message-driven beans for asynchronous application behavior. For more on EJBs, see the articles by Karl Kessler and by Katarzyna Fecht, Adrian Görler, and Jürgen G. Kissner on pages 3 and 111 of this issue, respectively.

[4] See **http://service.sap.com/connectors** for further information on the SAP Java Connector (JCo). In addition, *SAP Professional Journal* has published an extensive set of articles regarding the effective use of JCo, including "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).

we chose to use BEA WebLogic 6.1 as the application server. With SAP Web AS 6.30, however, SAP introduced a J2EE 1.3-compliant Java stack. We chose to migrate to SAP Web AS to consolidate our system landscape onto a single platform and to take advantage of the enhancements it offers. Let's take a closer look at these advantages and why it makes sense to migrate to SAP Web AS.

Why Port a J2EE Application to SAP Web AS?

Until the 6.30 and 6.40 releases of SAP Web AS,[5] most SAP customers were forced to maintain a J2EE-compatible server like BEA WebLogic or IBM WebSphere, or the open source JBoss or JOnAS, for their J2EE applications, in addition to an SAP server for their business applications, using either IBM Access Builder or SAP JCo to connect the two — not an ideal scenario, as the server maintenance cost is, in effect, doubled, and the frequent connection openings inevitably affect performance. Another issue was the lack of support for the full J2EE application development lifecycle — from the development environment, to version and configuration management, to deployment — which led many developers to create their own development environments based on the open source framework Apache Ant, a time- and resource-consuming endeavor.

SAP Web AS now includes a number of useful features and enhancements that promise to significantly improve developer productivity through the use of sophisticated tools and wizards, which reduce errors, as well as full support for the software development lifecycle. In particular, these include:

- Support for Web services
- A database-independent abstraction layer for persistence
- The ability to easily develop user-friendly Web frontends
- A comprehensive J2EE development process

Support for Web Services

If your application requires integration capability that is independent of operating systems and programming languages, like our C4L application required for managing software licenses for all machine types across the company, an efficient Web services infrastructure may be essential.

From a development perspective, it makes sense to follow a bottom-up approach for the generation of Web services to save you time and to enhance the reusability of the application logic. With this approach, you first implement the application's business logic as a stateless EJB session bean. The stateless session bean is then used to automatically derive the WSDL[6] description file for the Web services layer, which in turn allows the application logic in the bean to be accessed by a variety of clients. The SAP NetWeaver Developer Studio[7] proxy generator then uses the WSDL description file to create a client proxy for the Web service.

A Database-Independent Abstraction Layer for Persistence

Although the J2EE standard comprises Java Database Connectivity (JDBC) for relational persistence and container-managed EJB entity beans for object-relational persistence,[8] it is still a challenge to write applications that are completely database-independent,

[5] While SAP introduced support for the Java programming language and the J2EE 1.2 standard in SAP Web AS 6.20, its scalability and usability regarding J2EE applications were limited.

[6] Web Services Description Language.

[7] SAP NetWeaver Developer Studio, introduced with SAP Web AS 6.30, is the new SAP development environment for Java applications. For a detailed introduction, see the article by Karl Kessler on page 3 of this issue.

[8] Java object persistence will be covered in a series of forthcoming *SAP Professional Journal* articles, beginning with the article by Katarzyna Fecht, Adrian Görler, and Jürgen G. Kissner on page 111 of this issue. See also the article "Achieving Platform-Independent Database Access with Open SQL/SQLJ — Embedded SQL for Java in the SAP Web Application Server" (January/February 2004).

and thus portable. While it is theoretically possible to write fully database-independent applications if developers adhere strictly to the SQL statements supported by all vendors, it is highly dependent on the discipline of the developers involved. Every database vendor offers proprietary enhancements to differentiate its products, and more often than not, especially in large teams working in distributed locations, developers will use them, "breaking" the portability of the application (more on this later in the article). This is where SAP's persistence engine, the Open SQL Engine introduced with SAP Web AS 6.30, comes into play.

The Open SQL Engine offers a database-independent abstraction layer to support the development of applications that are, in truth, database-independent. You provide the commands, which must be Open SQL statements, for executing the database operations, and the persistence engine handles the rest — optimizing the statement for the particular database in use, for example. As a result, you can change the database without modifying the application code, making the application available to a wide range of supported databases. (The idea behind this concept is not new: it was first introduced by SAP for ABAP development with the ABAP Engine.)

> ✓ ***Note!***
>
> *For more details on the Open SQL Engine and Java persistence, see the article "A Guided Tour of the SAP Java Persistence Framework — Achieving Scalable Persistence for Your Java Applications" on page 111 of this issue.*

The Ability to Easily Develop User-Friendly Web Frontends

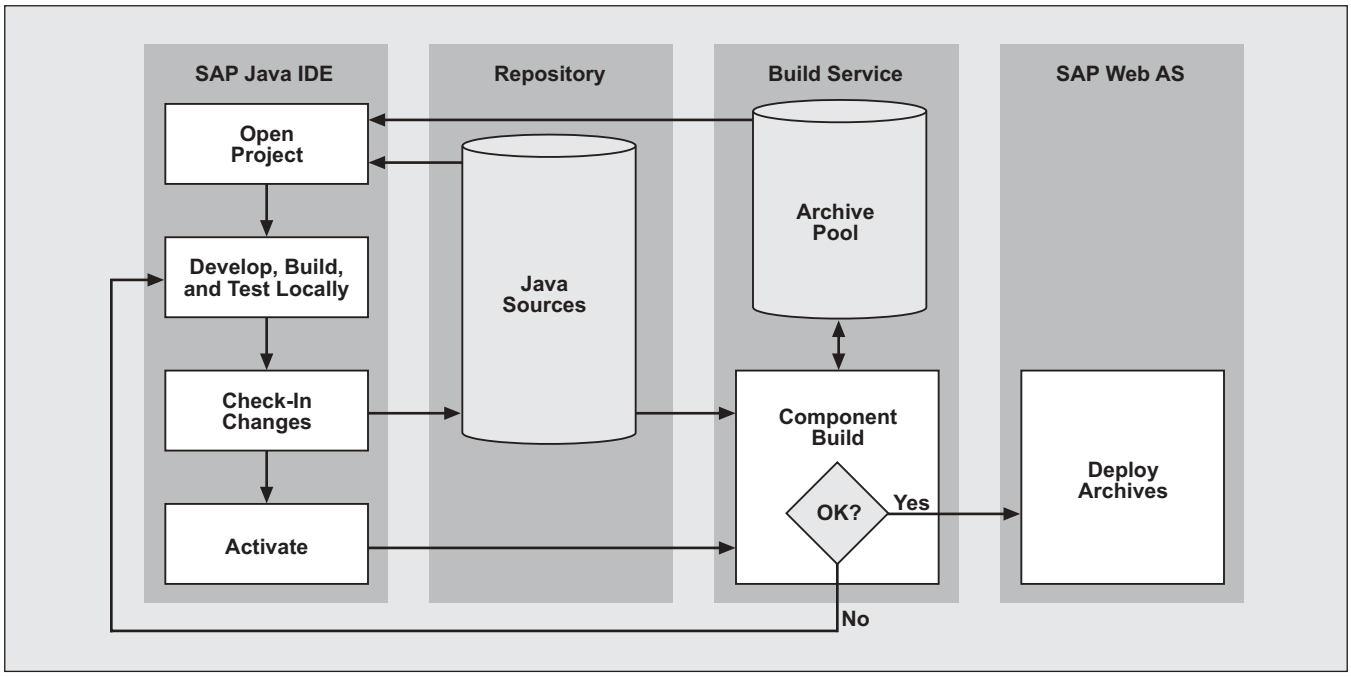
One of the challenges of developing applications with J2EE is building a user-friendly Web frontend. For instance, adding user-requested features such as client-side input checking and inter-field dependency checks can result in large portions of JavaScript programming that may not be portable across browsers, forcing you to maintain different coding for different browsers. Multiple design and prototype cycles can also drain your development resources. Considering the different available browsers and their idiosyncrasies (Netscape Navigator and Microsoft Internet Explorer sometimes require different implementations of the same HTML tag, for example), developing sophisticated frontends for different implementations of HTML and JavaScript can quickly lead to errors, project delays, and increased costs.

But a barebones design is not an option — users are now accustomed to comfortable client applications. The introduction of the Web Dynpro technology with SAP Web AS 6.30 represents a quantum leap in developer productivity. Web Dynpro includes development tools that allow you to design user-friendly frontends in a WYSIWYG format (such as dragging and dropping user interface components on the screen and binding them to data sources, for example). This approach delivers comfortable GUI controls ready for use with a minimum of programming effort, and provides less opportunity for programming error.

A Comprehensive J2EE Development Process

While J2EE application servers themselves continue to improve almost every year, professional development environments for creating complex server-side J2EE applications have not kept pace. This has changed recently, especially with frontend development, but many J2EE IDEs are still not as comfortable and productive as Microsoft's Developer Studio for C#, for instance. Some J2EE IDEs still lack tools that cover the complete development cycle, including configuration and transport management. As a workaround, most developers create their own tools and frameworks. While the open source build tool Apache Ant is helpful, building a sophisticated development environment on your own still means a lot of work (and a lot of money). And in most cases, a homegrown IDE will need to be re-tailored for each new project's requirements.

Figure 3 Overview of the Java Development Infrastructure (JDI) Delivered with SAP Web AS



Another challenge of server-side J2EE application development is the large number of files that make up a J2EE application, including JavaScript files, Java libraries and classes, HTML files, JSP files, and pictures, each of which might be used by many developers, and all of which must be kept synchronized. As a result, development processes often rely on nightly builds, where all files are recompiled to maintain synchronization, which can take a long time for large projects and can result in long build-repair cycles. The actual programming and testing of J2EE applications also takes place on the developer's local system, meaning that to make changes, the relevant files must be transferred to the developer's seat (which must also be configured properly), corrected, and then transferred back to the production system. This process can lead to chaotic situations, especially if large development teams in different locations are all working on a single project.

Using the tested ABAP Workbench concept, SAP Web AS includes the Java Development Infrastructure (JDI), a complete development infrastructure for professional J2EE development, as depicted in **Figure 3**. As you can see on the left-hand side, when a developer opens a project, all the needed libraries and components are automatically retrieved to the developer's local machine from the repository, along with the Java sources, a step that previously was manual (and very error-prone). The developer makes changes, builds, and tests locally, and then manually "checks-in" the changes to the repository and activates them. The Build Service then works behind the scenes to verify all dependencies for the changed sources and starts an "incremental build" in which only the *dependent* sources are compiled — in the past *all* files were compiled, leading to long build-repair cycles. If this build process finishes successfully, the newly created archives are deployed on SAP Web AS. The Build Service makes the sources and archives available to other developers only after a successful build, so that the team is always working with a running system; if there is an error during the build, the developer gets an error message and no deployments are executed on the server.

As you can see, there are several good arguments for migrating your applications to SAP Web AS, just as there were for us. But before you start porting them all, it's best to start with one so you can first evaluate how the server will behave under high load, for example, and then estimate the amount of effort the

porting project will require. The porting effort is highly dependent on the proprietary extensions used to develop the existing applications. These vendor-specific extensions are usually not supported across platforms, so you will need to identify where they are used and then estimate the amount of effort that will be required to redevelop the functionality on the new platform (more on this later in the article).

You're probably thinking, "This sounds time-consuming. Why not just use a synthetic benchmarking tool?" While J2EE server vendors provide such benchmarks, the tools and results are proprietary — i.e., they are often not reusable across different applications and platforms — and tend to concentrate on specific components of the application, like EJB performance, rather than on how well the components interact. Standard J2EE benchmarks, like SPECjAppServer2002,[9] provide a more comprehensive view, and can be useful for an initial evaluation, but by nature cannot mirror your real-world requirements. So, in order to evaluate a new application server like SAP Web AS 6.30 or 6.40, it is a good idea to migrate an actual application you intend to run on the new platform.

So which of your applications should you migrate to evaluate the server? That depends on the J2EE technologies used in your application and what is most important to you: Is it EJB or servlet performance? Do you have a lot of Web services, or is JMS critical for you? Depending on the answers to these questions, simply port the smallest of your applications that use the required technology, or extract an autonomous part of a larger application.

In the next sections, I'll show you how to port an application from BEA WebLogic 6.1 to SAP Web AS 6.30 or 6.40 (while my own migration was to 6.30, the steps are essentially the same for 6.40), and point out some potential trouble spots based on my own migration experience, as well as ways to enhance the portability of your applications for future migrations. While the porting project steps will look a little different if you are porting from a server other than BEA WebLogic, like IBM WebSphere or JBoss, and their details are highly dependent on the features you used to implement your particular application's functionality, you will gain a solid understanding of how porting projects are organized and the considerations to keep in mind.

[9] For more on the Standard Performance Evaluation Corporation (SPEC) and supported standard benchmarks, visit **www.specbench.org**.

Porting a J2EE Application to SAP Web Application Server

Porting an application to SAP Web AS consists of a simple eight-step process:

1. Plan for the migration.
2. Adjust the J2EE application and the target system deployment descriptors and configure the target system.
3. Create a deployment project.
4. Create an enterprise archive (EAR) file to contain the application's Web archive (WAR) and Java archive (JAR) files.
5. Reference any required preconfigured global libraries that already exist on the target system.
6. Deploy any required custom-built global libraries that do not yet exist on the target system.
7. Deploy the EAR file.
8. Verify and test the deployment.

Step 1: Plan for the Migration

From a theoretical point of view, J2EE migration projects shouldn't be that difficult — if all vendors stick to the J2EE standard when developing their server products, it should be easy to port applications with minimum effort. The problem lies in differing interpretations of the J2EE specification. It is a relatively young specification with a broad approach that covers difficult server-side development frameworks (like the one for object-relational persistence), and understandably does not yet cover all the critical aspects of development in complete detail.

Estimating the Migration Effort

There are a number of considerations involved in estimating porting efforts:

☑ **Extent to which proprietary extensions were used to develop the application:** Each extension used will require a portable alternative (see the next bullet item). Sun Microsystems offers the Java Application Verification Kit (AVK) for the Enterprise to help you locate proprietary extensions used in applications (the AVK is discussed in more detail later in the article). For a rough estimation, you can ask participating team members to identify the extensions they used and the number of times they used them.

☑ **Extent to which the application adheres to the J2EE standard:** If the application departs from the standard significantly, how much effort will be required to write a portable alternative? Sometimes you can find open source alternatives for a given feature. If there is no alternative, estimate the effort for implementing this feature on your own. The complexity of the application will also affect the porting effort. For instance, someone with broad JSP/servlet/database knowledge can port a simple JSP/servlet application in a matter of days, while porting an EJB-based application can require at least two people with EJB knowledge and months to complete.

☑ **Other ancillary conditions that may affect total porting time:**

- **Type of source platform:** The less comprehensive the knowledge of the target platform, the more the effort may increase. Even different Unix derivatives lead to different effort levels.
- **Type of J2EE APIs used:** The porting effort will increase with the number of different J2EE technologies used in the application. Effort levels also differ between technologies. Starting with

To make up for this, vendors offer proprietary extensions to help developers improve their productivity. However, this can cause applications to become vendor-dependent and can increase the migration cost if the target server does not support the extension in use (more on this later). This makes it difficult to estimate migration efforts — the time needed by the project participants to fulfill their work and the money required for new hardware or software — because it's hard to say which problems will crop up during the project. However, there are some general considerations I can point out for you, which are outlined in the sidebar above (I will also outline the problems we encountered while migrating our software licensing application in a later section).

Step 2: Adjust the J2EE Application and the Target System Deployment Descriptors and Configure the Target System

As mentioned before, J2EE vendors add proprietary features to their application servers to enhance J2EE programming capabilities. The J2EE specification provides vendor-specific deployment descriptors[10] that allow developers to configure and adjust these proprietary features.

[10] Deployment descriptors are required to inform containers about middleware needs, and are declared in a file. (In J2EE 1.3, deployment descriptors are written using XML.) Such descriptors define how containers perform lifecycle management, persistence, transaction control, and security services.

the costliest, the relative effort associated with the most popular J2EE APIs are: bean-managed entity bean > container-managed entity bean > stateful session bean > stateless session bean ≥ servlets > JavaServer Pages.

- **Number of EJB session beans, entity beans, and message-driven beans used:** The porting effort will increase in proportion to the number of beans used.

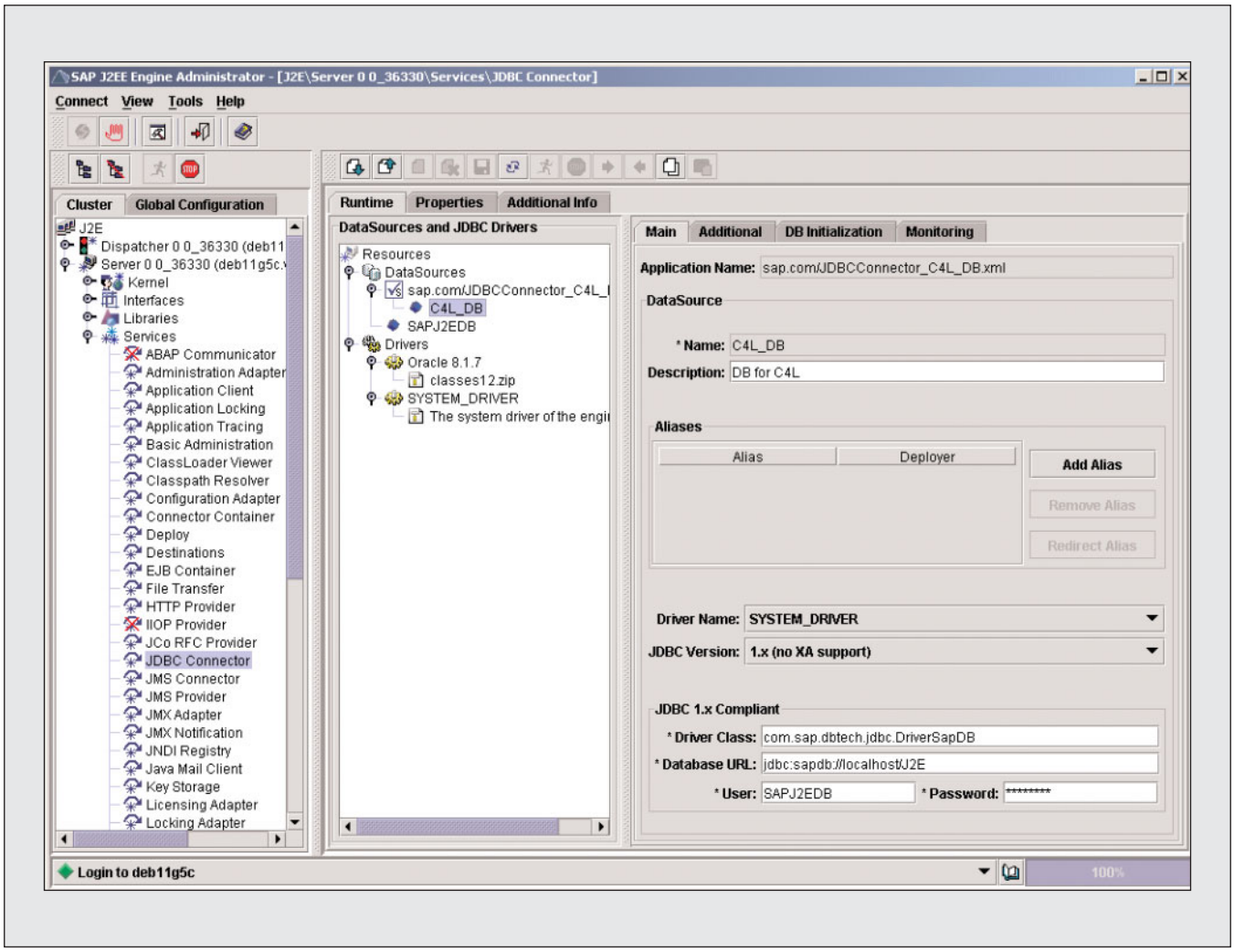
☑ **Extent to which the target server covers the application requirements:** To determine this, simply make a list of the technologies (and version numbers) used in your application and compare them with the feature list of the target application server. If it fits, you can start the porting project right away. If it doesn't fit, you have to decide if it makes sense to implement the missing features on your own (thus increasing the effort) or simply stop the porting project. In the case of our migration, for example, the following were required, all of which were provided by SAP Web AS:

- JavaServer Pages (JSP) 1.2
- Java Servlets 2.3
- Enterprise JavaBeans (EJB) 2.0, including stateless session beans, container-managed entity beans, a message-driven bean, and container-managed relations
- Java Naming and Directory Interface (JNDI)
- JavaMail
- Java Message Service (JMS)
- Web services

Each application server has its own set of features for addressing issues specific to that particular application server (like the primary key generation feature of BEA WebLogic; more on this later) as well as more common issues like load balancing, clustering, and monitoring. The specific steps for adjusting the features will depend on the particular feature and platform, so you will need to consult the relevant documentation. For our software licensing application, we needed to adjust the deployment descriptors for the application in order to use container-managed persistence. In addition to the standard *ejb-jar.xml* deployment descriptor specified by the EJB standard, SAP Web AS includes the vendor-specific deployment descriptors *persistent.xml* and *ejb-j2ee-engine.xml*.

The SAP-specific descriptor file *persistent.xml* is used to specify which entity bean fields map to which columns of which tables in the database. The EJB container uses this mapping when storing or retrieving container-managed fields from the database. Since the deployment descriptors are XML files, they can be adjusted using a simple text editor. Manually editing these XML files can be difficult, however, so vendors provide tools for handling them, such as the SAP Web AS Deploy Tool (more on this in an upcoming section). We also needed to configure a message queue for the Java Message Service (JMS) and data sources for object-relational persistence inside the target application server (SAP Web AS). This was done by using the SAP J2EE Engine Administrator of SAP Web AS.

Figure 4 The SAP J2EE Engine Administrator Screen



> ✓ ***Note!***
>
> *The SAP J2EE Engine Administrator is started via go.bat, which can be found in the SAP Web AS installation directory \usr\sap\<SID>\JC<instance number>\j2ee\admin\.*

Figure 4 shows the SAP J2EE Engine Administrator screen. Here, you can see a data source entry for the database connection *C4L_DB*, which was created as follows:

1. Select the *JDBC Connector* service from the tree on the *Cluster* tab in the left pane of the SAP J2EE Engine Administrator. On the *Runtime* tab in the right pane, choose the *DataSources* node under *Resources*.
2. Choose the toolbar button *Create New Driver or DataSource* (🗋).
3. Specify a unique name for the data source in the *Name* field, which will be used to look up the data source in the naming service of the J2EE Engine (*C4L_DB* in our case).
4. Choose a 1.x-compatible JDBC driver from the

Driver Name dropdown list of registered drivers. The drivers available for selection here are those listed in the *Drivers* subtree on the *Runtime* tab. You can add any vendor-specific JDBC driver to this list. Simply copy the ZIP or Java archive (JAR) file containing the driver to your file system, and then select the *Drivers* node on the SAP J2EE Engine Administrator *Runtime* tab, choose the *Create New Driver or DataSource* toolbar button (), enter a name for the driver (e.g., *Oracle 8.1.7* in Figure 4), and select the copied ZIP or JAR file from the pop-up that appears.

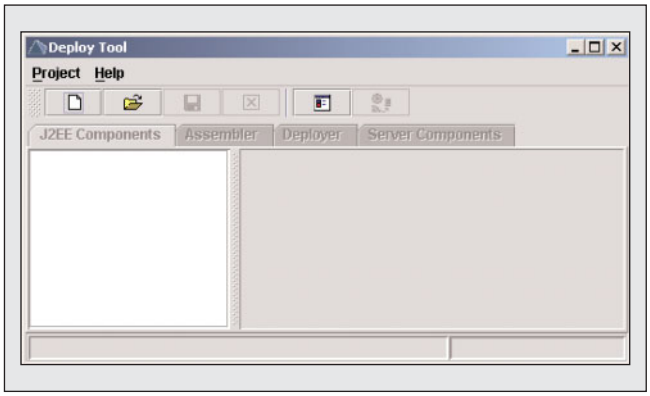
5. Choose 1.x from the *JDBC Version* dropdown list of supported JDBC versions.
6. Enter the fully qualified Java name of the driver's main class in the *Driver Class* field — e.g., *com.sap.dbtech.jdbc.DriverSapDB*.
7. Enter the database URL, which consists of a protocol identifier (always JDBC), a driver identifier (SAP DB, IDB, Oracle, etc.), and a database identifier (the format is driver-specific) — e.g., *jdbc:sapdb://localhost/J2E*.
8. Specify the user name for logging in to the database server.
9. Specify the password for the database user.
10. On the *Additional* tab, set the SQL type support.
11. Save your entries.

Step 3: Create a Deployment Project

When you are porting applications to SAP Web AS without adding code (which is the case when you simply want to evaluate an application server, as we are doing here), you can use the SAP Web AS Deploy Tool as the center of your deployment activities. (To add code, you would use SAP NetWeaver Developer Studio,[11] which is the SAP Web AS IDE introduced with SAP Web AS 6.30.)

[11] For a detailed introduction to SAP NetWeaver Developer Studio, along with a brief discussion of how to use it for deployments, see the article by Karl Kessler on page 3 of this issue.

Figure 5 The Deploy Tool Initial Screen



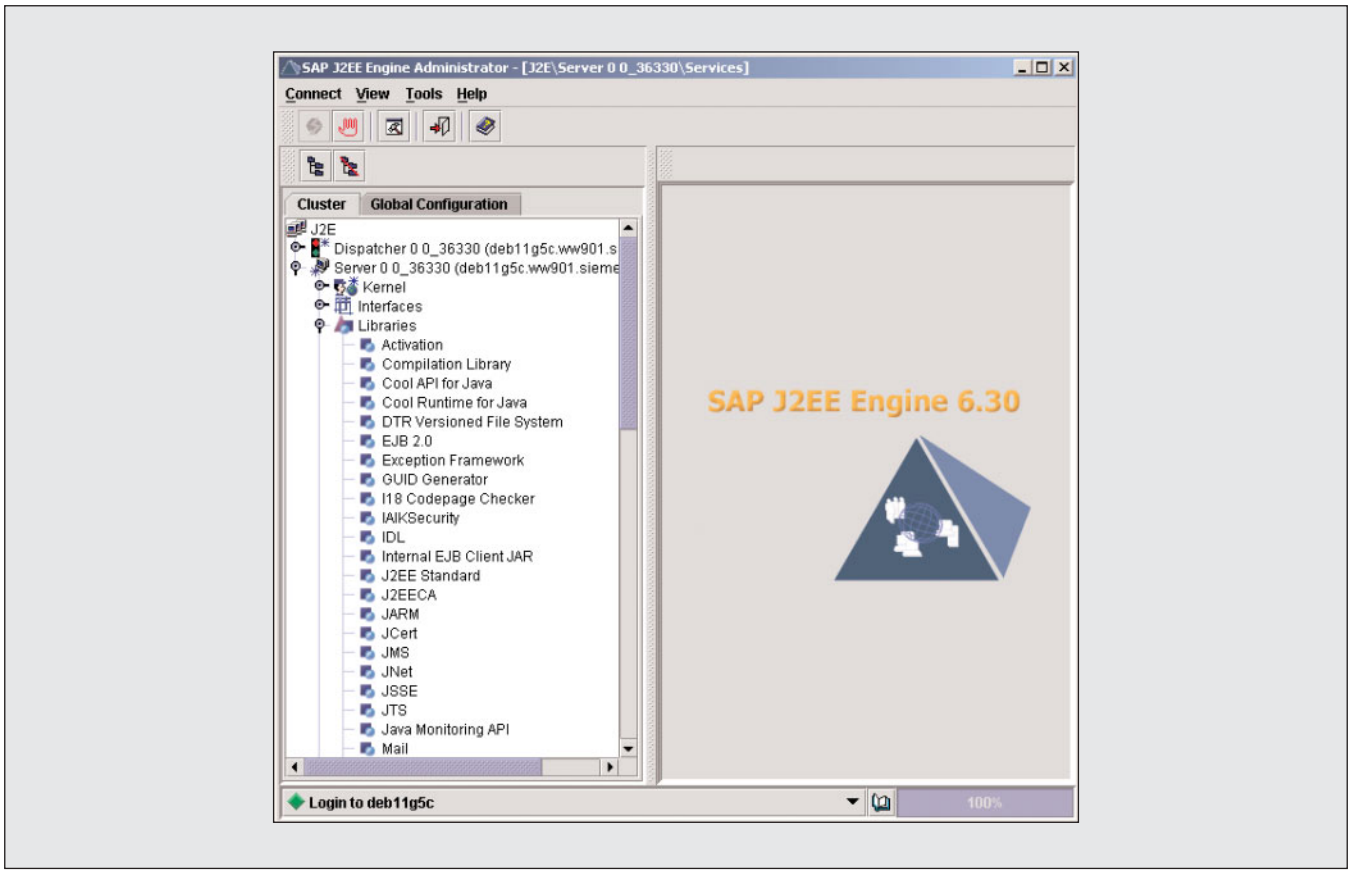
The Deploy Tool is a graphical user interface tool that generates and assembles archive components based on the J2EE application and deploys them on the J2EE Engine included with SAP Web AS 6.30 and higher. The Deploy Tool can be used to create J2EE components from the class files and deployment descriptors of an existing J2EE application, to assemble these components in an application enterprise archive (EAR), to edit its deployment properties, and to deploy the EAR on the specified J2EE Engine cluster elements.

When the Deploy Tool is first started, the screen shown in **Figure 5** appears, showing four tabs where the necessary information for deployment must be entered.

> ✓ ***Note!***
>
> *The J2EE deployment tool is started via DeployTool.bat, which can be found in the SAP Web AS installation directory under \usr\sap\<SID>\JC<instance number>\j2ee\deploying\.*

But before you specify the details of the deployment, you need to create a "deployment project," which is the basis of all deployment activities. Select *Project → New Project* on the Deploy Tool initial screen, which creates a deployment project file ending with the extension *.dlp*.

Figure 6 Server Libraries Delivered with SAP Web AS



Step 4: Create an EAR File to Contain the Application WAR and JAR Files

You can only deploy complete enterprise archive (EAR) files to SAP Web AS. EAR files consist of Web archive (WAR) files for Web applications and Java archive (JAR) files for Enterprise JavaBeans (EJB) development. The *J2EE Components* tab of the Deploy Tool provides for the creation of WARs and JARs, which contain simple Java class files and deployment descriptors that must be added explicitly out of the directory where the application was developed. Since we are porting an existing J2EE application, the necessary JAR and WAR files already exist, so we don't need to use this tab.[12]

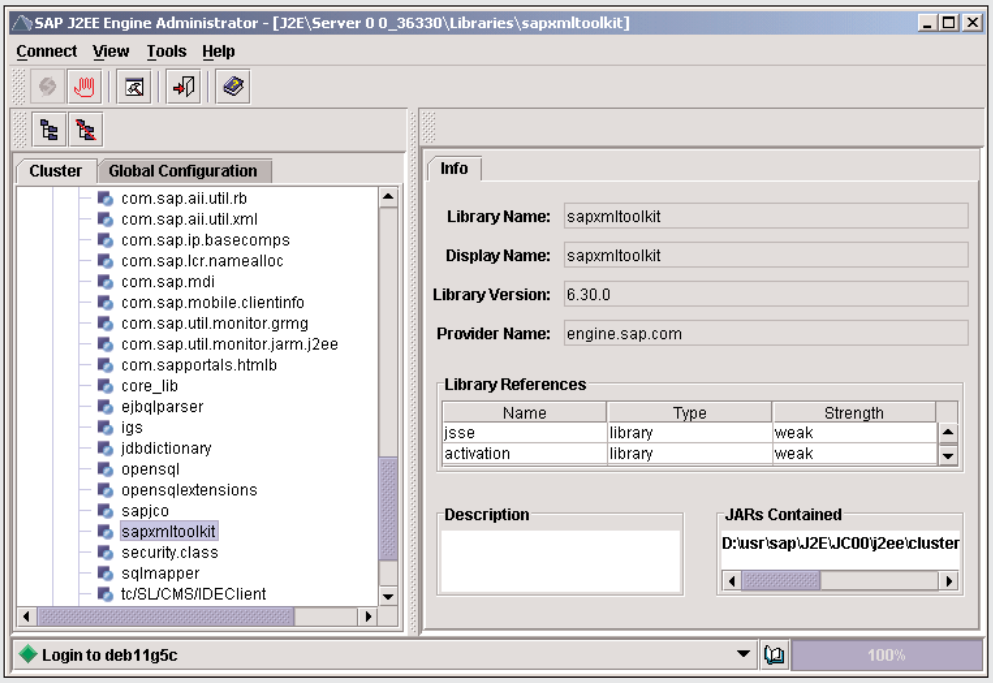
[12] For more details on creating WAR and JAR files with the Deploy Tool, go to the SAP Web AS 6.30 or 6.40 help at **http://help.sap.com** and navigate to *SAP NetWeaver Components → SAP Web Application Server → J2EE Technology in SAP Web Application Server → Development Manual → Deployment: Putting It All Together → Deploy Tool*. Note that once you have migrated to SAP Web AS 6.30 or 6.40, you would use SAP NetWeaver Developer Studio instead.

The *Assembler* tab of the Deploy Tool is used to create the EAR file that deploys the components of the J2EE application on the target SAP Web AS system. Add the archives that make up the EAR file — the WAR and JAR files — via the menu path *Assemble → Add Archive*.

You can also add existing archives from other sources to ease the deployment of simple Web applications delivered in WAR files. For example, the Apache Struts framework provides an infrastructure that allows developers to plug in their business logic and Web pages — the flow from one Web page to the next is described inside preconfigured XML configuration files, with no additional programming required. The accompanying library that assembles these files can be easily deployed with the application using the *Server Components* tab of the Deploy Tool (see step 6 on page 82).

Finally, the menu selection *Assemble → Make*

Figure 7 Accessing the Library Information



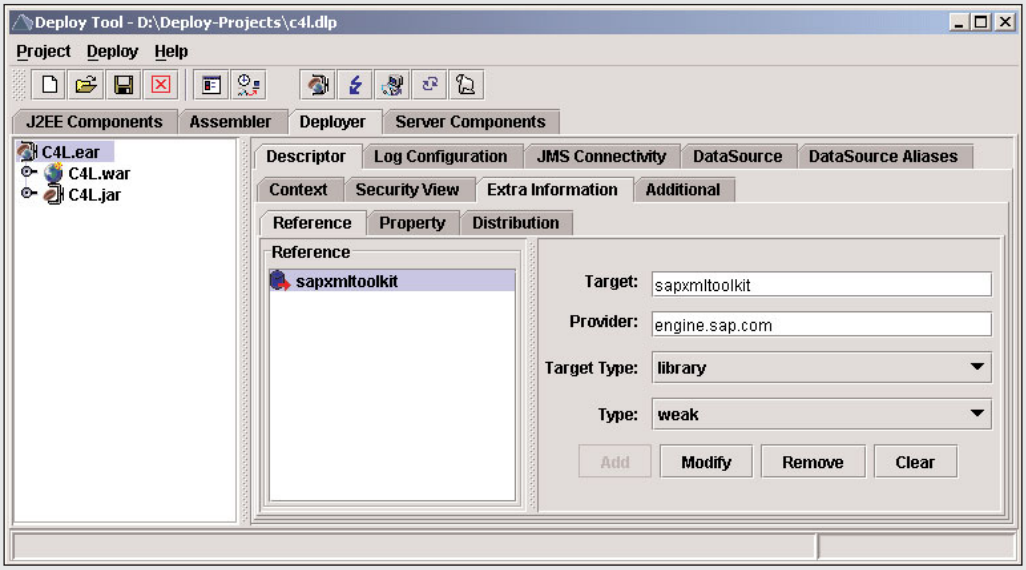
EAR creates the EAR file. If you don't need to reference or deploy any preconfigured or custom-built global libraries, the EAR file is now ready for deployment on the target application server, and you can skip to step 7. Otherwise, you must complete steps 5 and 6 before deploying the EAR file.

Step 5: Reference Any Required Preconfigured Global Libraries That Already Exist on the Target System

Some libraries required by your application may already exist as part of SAP Web AS. SAP Web AS 6.30 and 6.40 deliver a number of preconfigured libraries, such as an XML library, ready for use with different applications. To avoid redeploying these libraries for each application over and over again, it is a good idea to reference them. For example, almost every application needs a way to handle XML files, so it makes sense to reuse a single XML library for every application that requires XML access instead of maintaining a separate library for each application. For an overview of these server libraries, go to the SAP J2EE Engine Administrator, and in the tree on the left select *Server* → *Libraries* (see **Figure 6**). You can select these libraries directly, rather than having to deploy them (more on this in step 6). The question is: How do we tell the application server during deployment that our application wants to reuse an already-existing server library? In other words: How do we reference a library we see in the *Libraries* subtree of the SAP J2EE Engine Administrator?

Let us examine this question by using as an example the XML library *sapxmltoolkit*, which is reused in nearly every modern enterprise application. The Deploy Tool needs the library name and the library provider name for every library to be reused, which can be found by selecting the library node in the left pane of the SAP J2EE Engine Administrator, as shown in **Figure 7**. The information will then appear in the *Info* tab on the right, in the fields *Library Name* and *Provider Name* — *sapxmltoolkit* and *engine.sap.com*, respectively. On the *Deployer* tab of the Deploy Tool, select the EAR file (*C4L.ear* in the example), follow

Figure 8 Adding Library Reference Information to the Deployer Settings



the tab hierarchy *Descriptor* → *Extra Information* → *Reference*, and add the target and the library provider names, as shown in **Figure 8**.

You can then verify the success of this referencing step by opening the automatically generated descriptor file for the application using the Deploy Tool — in this case, *application-j2ee-engine.xml* — to see if the reference was included. **Figure 9** shows the deployment descriptor code for the example, which includes the reference we just added (shown in bold).

By referencing existing server libraries in this way, you can easily reuse global general-purpose libraries and avoid redundant deployment, saving you from redeploying them or writing similar libraries for each application.

Step 6: Deploy Any Required Custom-Built Global Libraries That Do Not Yet Exist on the Target System

The *Server Components* tab of the Deploy Tool is used to deploy any global libraries that were custom-built by the developer to the target application server, so you can reference them the way you reference the preconfigured libraries that already exist on the target system. For example, if a collection of industry-specific calculations is often used by several enterprise applications, it makes sense to pack them in one library and deploy the library as a shared resource. There are no custom-built global libraries used by our application, so we do not use this tab.

Step 7: Deploy the EAR File

The EAR file is deployed using the *Deployer* tab. You can specify the administration port of the application server (e.g., the default value is 50004) and then deploy the EAR by following menu path *Deploy* → *Deployment* → *EAR*, by selecting *Deployment* → *Deploy EAR* from the context menu of the EAR file, or by selecting the deploy button ( ). No other settings are necessary on this tab; simply accept the remaining defaults.

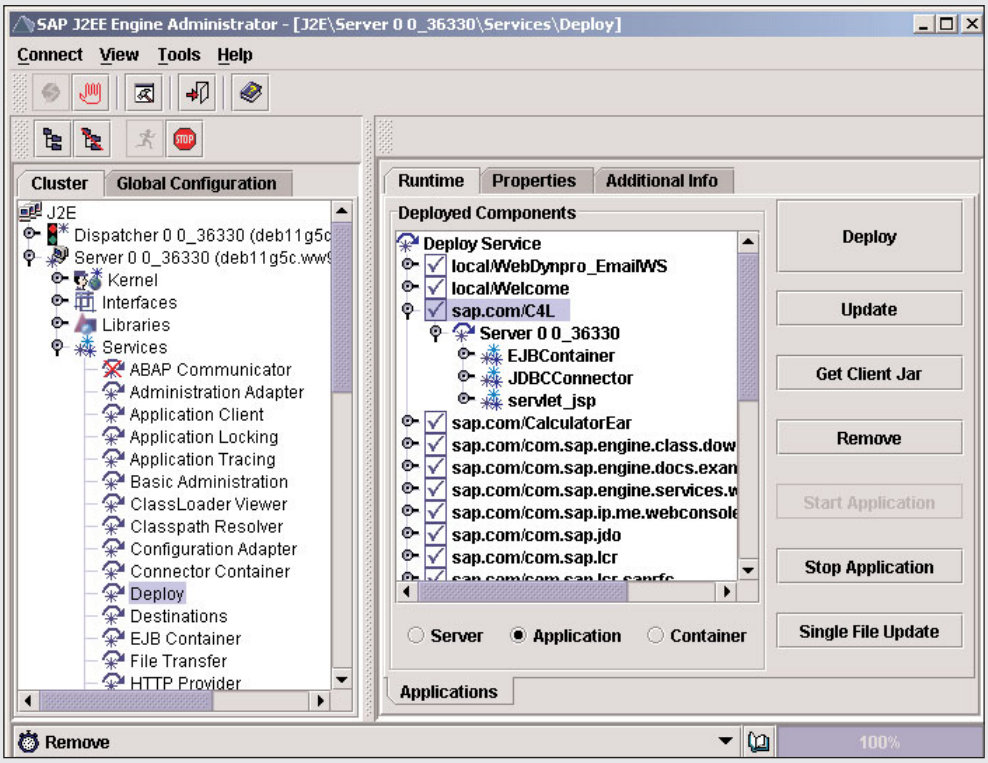
Step 8: Verify and Test the Deployment

You can verify the success of the deployment (i.e., whether the application and any deployed libraries are properly installed and running) via the SAP J2EE

Figure 9 The Library Reference Added to the Application's Descriptor File

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE application-j2ee-engine SYSTEM 'application-j2ee-engine.dtd'>
  <application-j2ee-engine>
    <reference reference-type="weak">
      <reference-target provider-name="engine.sap.com" target-type="library">
        sapxmltoolkit
      </reference-target>
    </reference>
    <fail-over-enable mode="disable">
    </fail-over-enable>
  </application-j2ee-engine>
```

Figure 10 A Successfully Deployed J2EE Web Application



Engine Administrator — if the deployment was successful, new entries will appear when you select *Server* → *Services* → *Deploy* for Web and enterprise applications or *Server* → *Libraries* for server components.

Figure 10 shows that the C4L application has been successfully deployed — the node *sap.com/C4L* appears in the *Runtime* tab display for the *Deploy* node. (Note that the radio button *Application* must be checked to display the application in the *Runtime* tab.)

At this point, your next steps depend on the errors identified by the Java stack traces that are triggered by exceptions thrown during testing. For J2EE applications, I recommend test tools such as JUnit or HttpUnit for Web applications, or Apache Cactus as an overall test framework for server-side Java code (servlets, EJBs, etc.). All of these tools can automate unit testing for large parts of the application.[13]

> ✓ ***Note!***
>
> *The following Web sites provide additional information about J2EE and the deployment process:*
>
> - ***http://java.sun.com/developer/technicalArticles/J2EE/Intro/index.html***
> - ***http://java.sun.com/j2ee/learning/tutorial/***

While testing our ported software licensing management application, we did experience some problems, which we will look at next. While our particular problems were specific to porting from BEA WebLogic 6.1 to SAP Web AS 6.30, the discussion will serve as a good starting point for identifying potential trouble spots in your own porting project.

Potential Porting Problems

The problems we encountered in our porting project fell into three distinct categories:

- Differing interpretations of the J2EE specification
- The use of proprietary extensions
- The use of special design patterns

[13] Details about these tools can be found at:
- **http://junit.org**
- **http://httpunit.sourceforge.net/**
- **http://jakarta.apache.org/cactus**

In the next sections, I'll describe the particular problems we encountered in detail, to give you a feeling for the direction you might want to take your own investigations.

Differing Interpretations of the J2EE Specification

Most of the problems we encountered when porting our software licensing application were caused by differing interpretations of the J2EE specification:

- BEA WebLogic does not check for certain mandatory exception declarations. SAP Web AS is much more exact in following the specification. For example, chapter 10.6.4 of the EJB 2.0 specification requires the enterprise lifecycle method *ejbCreate* for container-managed entity beans to throw a *javax.ejb.CreateException*. The *ejbCreate* method in our application was originally declared as depicted in **Figure 11** — as you can see, it throws a *RuntimeAccessorException* instead of a *javax.ejb.CreateException*.

  Be sure to check the deployment descriptors before deploying your application. The Java Application Verification Kit (AVK) for the Enterprise from Sun Microsystems (more on the AVK in an upcoming section) is a useful tool for detecting errors like this one because it checks the deployment descriptor for correctness. Using a tool like the AVK will save you from having to resort to the time-consuming trial-and-error method.

- BEA WebLogic does not check for incorrect transaction configurations (see chapter 15.4.8 of the EJB 2.0 specification). For example, with message-driven beans, the container handles the acknowledgement of received messages; therefore programmers should never use the JMS API[14] to

[14] JMS is part of the J2EE standard and comprises the asynchronous sending and receiving of messages. It's generally used for connecting systems in a loosely coupled fashion.

Figure 11 The Example Application's ejbCreate Method Exception Declaration

```
public abstract class ComputerBean
    extends GenericAttributeAccessDefaultImpl implements EntityBean
{
 ...
   public Integer ejbCreate(Map values) throws RuntimeAccessorException
   {
      this.setValuesForSpecifiedAttributes(values);
      return null;
   }
 ...
}
```

acknowledge messages. Programmers can, however, tell the container *how* to acknowledge received messages using the "acknowledge-mode" deployment descriptor. But this special descriptor can only be set for message-driven beans with *bean*-managed transaction demarcation, and the message-driven bean used by our software licensing application has *container*-managed transaction demarcation, so the acknowledge-mode deployment descriptor cannot be used. Although the descriptor was incorrectly set for the ported application, BEA WebLogic accepted the setting whereas SAP Web AS returned an error.

Again, use a tool like the AVK to check the deployment descriptors, which will save you a lot of time and effort.

- BEA WebLogic does not check the case-sensitivity of descriptor names; SAP Web AS, on the other hand, will refuse the deployment if the capitalization is incorrect. Chapter 22.5 of the EJB 2.0 specification states that the content of an XML element of the deployment descriptor is, in general, case-sensitive. The ported application used an incorrect descriptor name (the error is in bold):

```
<acknowledge-mode>auto-
acknowledge</acknowledge-mode>
```

The name should have been:

```
<acknowledge-mode>Auto-
acknowledge</acknowledge-mode>
```

This is also something that the AVK could easily have detected.

- BEA WebLogic does not check for correct parameters in EJB Query Language (EJBQL) expressions,[15] while SAP Web AS does. It is possible to formulate EJBQL expressions with subselects, which are represented using the keyword *IN(x)*, where *x* stands for a set of objects. Since *x* must be a set to search in, and you can't search inside a simple value, *x* cannot be a simple value (see chapter 11.2.6 of the EJB 2.0 specification), but this is exactly what happened in our software licensing application, so SAP Web AS did not accept the expressions during deployment.

  To address this problem, we had to correct the EJBQL inside the deployment descriptor. Once again, the AVK can detect such incorrect expressions for you.

[15] EJBQL uses a SQL-like syntax to select, update, or delete objects or values out of a database.

Figure 12 Accessing Entity Bean Members in the Example Application via Java Reflection

```
Field[] fieldsInBaseClass  = baseClassRef.getDeclaredFields();

for (int i=0;i<fieldsInBaseClass.length;i++)
{
     String attributeName = (String)fieldsInBaseClass[i].get(null);
      this.addMapEntry(attributeName, classRef, classInstance);
}
```

The Use of Proprietary Extensions

As I have mentioned before, vendors offer proprietary extensions to help J2EE application developers improve their productivity, but if an extension is not supported by the target server, this means the application is not portable! You cannot run it on the target machine, and you have to develop an alternative on your own. Depending on the proprietary extension in use, this can add a significant amount of effort to the porting project.

For example, the BEA WebLogic primary key generation extension saves developers from having to worry about creating extra database tables to store the last assigned primary key for a certain entity. SAP Web AS does not support this extension, however, so we had to write a new, application-server-independent component for primary key generation. Since the component works on a new database table, where the last assigned primary key for each entity bean is stored, we had to adjust each *ejbCreate* method of each entity bean so that it used the new generated component for retrieving the next primary key.

The lesson here is to stay away from proprietary extensions. If you use them already, keep in mind that you will have to implement a portable alternative so that your application will be able to run on SAP Web AS. Every proprietary extension means additional effort for the migration project that must be considered. (I will discuss proprietary extensions and how to ensure portability in more detail shortly.)

The Use of Special Design Patterns

We also used a special design pattern in our software licensing application to access members of an entity bean directly via Java reflection instead of using accessor methods (see **Figure 12**). Unlike BEA WebLogic, SAP Web AS doesn't allow this kind of member access, so we had to completely rearrange the code. The lesson here is to follow the J2EE specification and don't deviate from it! SAP Web AS will detect any deviations and force you to rewrite the erroneous code to achieve complete J2EE compliance.

Implications for the Porting Project

We were able to fix all of the problems we encountered relatively easily and found that similar migration projects can be handled in a time frame of two to four weeks, assuming some J2EE knowledge. The elapsed time can become even shorter if the J2EE solution to be ported consists only of servlets and JSPs, which are older, and therefore simpler and more stable specifications. Migration of our servlet/JSP-based applications required almost no additional work: simply take the WAR file, wrap it in an EAR file, and deploy it to SAP Web AS via the Deploy Tool. These larger applications consisted of several hundred JSP files and used the Apache Struts framework for implementing the Model-View-Controller (MVC) design pattern[16]

[16] For more on the MVC design pattern, see the *SAP Professional Journal* articles "Build More Powerful Web Applications in Less Time with BSP Extensions and the MVC Model" (March/April 2003) and "Develop More Extensible and Maintainable Web Applications with the Model-View-Controller (MVC) Design Pattern" (January/February 2004).

and the Apache Cocoon framework for generating PDF files on the fly. However, even for these more simply structured applications, to avoid problems like the ones we encountered, the application has to stick strictly to the servlet/JSP standard as described in the Servlet 2.3[17] and JSP 1.2[18] specifications, respectively. This cannot be stressed enough.

Every project will face its own unique problems, so the ones I've outlined here can only give you a rough idea of the sources of possible errors. These issues are typical for these kinds of porting projects, however, and the discussion should lead you in the right direction when you encounter problems during your own projects. Nevertheless, it's always a good idea to have the J2EE 1.3 specifications at hand in case of ambiguities.

So what steps might we have taken from the outset of our application development to anticipate future porting activities? How portable is J2EE, really, and how can you ensure future portability? We'll look at these questions in the next sections.

How Portable, Really, Is J2EE, and How Can You Ensure Future Application Portability?

It should be clear that the problems we encountered during our migration were not showstoppers. While it was not possible to run the code immediately, the porting was a lot easier than in the old C or C++ days. (Remember the never-ending sequences of *#IFDEF* macro definitions for making code portable, starting with trivial issues such as the length of an integer on a given platform?) The only code changes we had to make were the alternatives to the proprietary extensions and special design patterns used. J2EE components such as servlets, JSPs, and stateless session beans are all portable without changing code or deployment descriptors — we only had to adjust the deployment descriptors for the entity beans.

[17] See **http://java.sun.com/products/servlet/download.html**.

[18] See **http://java.sun.com/products/jsp/download.html**.

While the implementation of the J2EE specification by different vendors is stable, and has gained broad acceptance and adoption by major server vendors, a significant challenge that remains to J2EE application portability is that vendors offer many proprietary add-ons to ease J2EE programming efforts, as I've mentioned before. While this might seem like a good thing, if you use these add-ons — such as the BEA WebLogic primary key generation facility we used for C4L, or its timer facility for regularly scheduling activities on the application server[19] — your application will lose its portability.

You're probably thinking, "But I want and need these extensions because they save me lots of time!" If a project gets into trouble because of a tight schedule, portability becomes less important than speed, and you will most likely use these features. It's the same with writing portable JDBC applications in which you use pure SQL statements — it's very likely that you will use proprietary SQL extensions of the database in use as soon as your project enters a critical phase. Whether an application remains portable or not, then, becomes dependent on programmer discipline.

> ✓ ***Tip***
>
> *While programmer discipline is largely up to the programmers themselves, it is a good idea to create a "developer handbook" that summarizes key "dos and don'ts." See* ***www.ambysoft.com/javaCodingStandards.html*** *for a useful example of a Java coding guide.*

If you know for certain that you are relying on a single vendor's products for developing and running your applications, perhaps because your company is the vendor, then it may be advantageous to use proprietary extensions, and portability may not be an issue. (Keep in mind, however, that most frameworks in the

[19] Note that these particular extensions are supported officially in the new J2EE 1.4 specification.

open source world are comparable to vendor extensions without sacrificing portability, since open source solutions are, in most cases, usable on all J2EE-compliant servers.) Or perhaps your company's IT strategy is based on "best-of-breed" products, and the application server in use may change overnight, or you may be writing J2EE-compliant applications for a mass market. In these cases, J2EE portability is of great importance, because you have to ensure that your solution can run on a wide range of J2EE-compliant application servers, which leads us to the next question: How do you ensure the portability of a J2EE application?

There are at least two techniques you can use to enhance the portability of your J2EE applications:

- The Java Application Verification Kit (AVK) for the Enterprise
- J2EE design patterns and best practices

The Java Application Verification Kit (AVK) for the Enterprise

The Java Application Verification Kit (AVK) for the Enterprise from Sun Microsystems helps developers:

- Test their applications for correct use of the J2EE APIs
- Test their applications for portability across J2EE-compatible application servers
- Avoid writing code that undermines the portability of their applications

To achieve true portability, your code must not use vendor-specific extensions to the J2EE platform, and your application must meet the requirements described in the J2EE specification. The Java BluePrints program (see **http://java.sun.com/blueprints** for details) provides an extensive set of guidelines, design patterns, and sample applications for building portable J2EE-based applications. However, it is still possible to inadvertently write nonportable applications. The Java AVK for the Enterprise is designed to help developers avoid such situations.

If an application passes the tests provided by the AVK, it is portable or easily portable to other J2EE-compliant platforms.

The AVK comprises the following tools:

- **J2EE Reference Implementation (RI):** The RI is a full-blown J2EE-compliant application server with certain extensions that allow logging of public API invocations at runtime. By correctly implementing the J2EE specification, the RI serves as an example of what a J2EE-compliant server should look like.

> ✓ ***Note!***
>
> *Although Sun attempts to implement the J2EE specification as closely and completely as possible in its RI, it should be noted that the RI is not a server for productive use — it was developed to test for J2EE compliance, and so performance, reliability, availability, and scalability issues are out of its scope.*

The RI is responsible for the execution of "dynamic tests." The developer deploys and runs an application on the RI. After deployment, the developer uses the application as usual (manually or via automatic test tools), trying to cover as much of the code as possible. As the application runs on the RI, the RI logs any API invocations and exceptions using its extensions. The dynamic verification process itself consists of two steps:

- The first step, called "introspection," generates a list of public APIs of the J2EE application (EJBs, servlets, JSPs) that should be called when the application runs on the server.
- The second step, called "instrumentation," logs the calls that were actually executed during runtime.

By comparing the list gathered by the "introspection" step with the list generated during the

"instrumentation" step, it is possible to generate API coverage metrics for the application.

- **Verifier:** The Verifier executes "static tests." It comprises over 2,000 test cases based on the J2EE specification and checks them against all the EAR files and deployment descriptors of an application. For example, the Verifier checks that:
  - All components named in the deployment descriptor are available
  - Statements expressed using EJBQL are semantically correct
  - JAR/WAR/EAR files are packed correctly
  - Method names and signatures are correct
  - Return values defined in the remote and local interfaces are exactly the same
- **Report Tool:** This tool summarizes the results from the static and dynamic checks. It compares the instrumentation list against the introspection list, calculates average numbers for the methods and Web components, and produces static as well as dynamic reports from the static checking and dynamic checking coverage results, respectively. From the information in these reports, you can derive the percentage of method calls passing the check and display a list of methods that did not pass the test.

An application is verified as portable if it fulfills the following criteria:

- EJBs and Web components pass all Verifier tests
- EJB components run in the RI and 80% of the methods are called without system exceptions
- All servlets and JSP pages can be called without compilation and runtime errors

If portability is an issue and you want to start developing a new J2EE-based application with portability in mind, it is a good idea to integrate the AVK verification process into your build process right from the beginning. On the other hand, if the J2EE application already exists and you want to find out how portable it is, start with the Verifier and analyze its output. After correcting the issues reported by the Verifier, you can port the application to the RI and run your tests on the RI server to get the coverage numbers. These numbers will guide you to the portions of code that require additional testing.

> ✓ ***Note!***
>
> *Using the AVK is no guarantee that your application can be installed on the J2EE server of another vendor and that it will run immediately and under all circumstances:*
>
> - *The AVK doesn't currently scan source code to find sequences that use vendor-proprietary APIs or that do not conform to the J2EE specification.*
> - *The AVK will throw an exception if vendor-specific code is called; since you cannot expect that all possible paths through your code will execute, there is still some degree of uncertainty about whether there is still a code path containing vendor-specific code.*
>
> *Although the AVK will aid in assessing and implementing true portability, you can expect to have some work left (such as integration testing) even after its use.*

One final note regarding the use of proprietary extensions: the AVK accepts the use of proprietary features, but only if you provide the features in a portable way for other application servers. What does this mean? I previously mentioned BEA WebLogic's proprietary primary key generation technique and how we solved the problems presented by its use. By providing server-specific solutions in the appropriate method, I can run this feature on both a BEA WebLogic server and on application servers that don't support it. This method of solving portability issues is called "code branching," where the same feature is developed twice — one version for a particular application server by using its proprietary extension, and a portable version for all other application servers.

> ✓ ***Note!***
>
> *The AVK can be downloaded from* ***http://java.sun.com/j2ee/avk/****. The download is free of charge if you simply want to check your programs internally. If you wish to reference a tested application as "Java Verified," then you will need to apply for the Java Verified Trademark License. Details about writing portable programs, as well as prerequisites, license costs, how to install the AVK, and so on, can also be found at this site, especially on the FAQ page (****http://java.sun.com/j2ee/verified/faq.html****).*

J2EE Design Patterns and Best Practices

During our porting project, we also learned that developer productivity can be optimized through the proper use of J2EE design patterns and best practices. Although these are broad and well-covered topics, let me point out some of the more important aspects and how they pertain to J2EE portability.

While I have scratched only the surface of some of the difficulties presented by J2EE programming, it should be clear that the development of large J2EE-based applications is very complex, and even programmers with deep Java knowledge may have difficulty with J2EE programming. For example, to cover the complete spectrum of J2EE programming, you will need experts in multiple domains — object-relational mapping, transaction handling, security, messaging, parallelism, legacy integration, and resource management, just to name a few, each of which comes with its own set of pitfalls. And keep in mind that the J2EE specification is still evolving — while this article focuses on J2EE 1.3, J2EE 1.4 has been released with even more extensions. This is where design patterns can help your porting efforts.

Design patterns are practical, proven solutions for challenges that every programmer faces during software development, especially during the development of J2EE-based applications. They can save you a lot of time, money, and frustration — you can take advantage of the experiences of others and use solutions that have already been successful in real-world applications. Another advantage is that design patterns are constructed to fit any J2EE-compliant application server, so they are highly portable.

Let's look at the "Session Façade," which is the most widely used EJB design pattern.[20] The Session Façade design pattern addresses the issue of how to properly partition the business logic between session and entity beans in order to execute the business logic in one bulk network call instead of multiple fine-grained invocations. The problem that this pattern addresses is depicted in **Figure 13**, which shows a direct entity bean access by remote clients.

Calls to session or entity beans are required to execute business logic on the server. As you can see in Figure 13, every fine-grained invocation of the deployed session/entity beans adds multiple network calls. What does "fine-grained" invocation mean in this regard? Let's take an online banking scenario as an example. To transfer money from one account to another, the following "fine-grained" steps must be called from a remote client:

1. Find the entity bean of the user who initiates the transfer.
2. Determine whether the user is authorized to transfer money.
3. If so, find the entity bean of the account from which money will be withdrawn.
4. Find the entity bean of the account to which money will be deposited.
5. Withdraw money from the account found in step 3.
6. Deposit money in the account found in step 4.

This simple scenario requires at least six bandwidth-draining network calls. The Session Façade pattern optimizes this situation. It simply

[20] The example shown here is an abbreviated version of one discussed in more depth in the book *EJB Design Patterns* by Floyd Marinescu.

Figure 13 Executing Business Logic Without the Session Façade Design Pattern

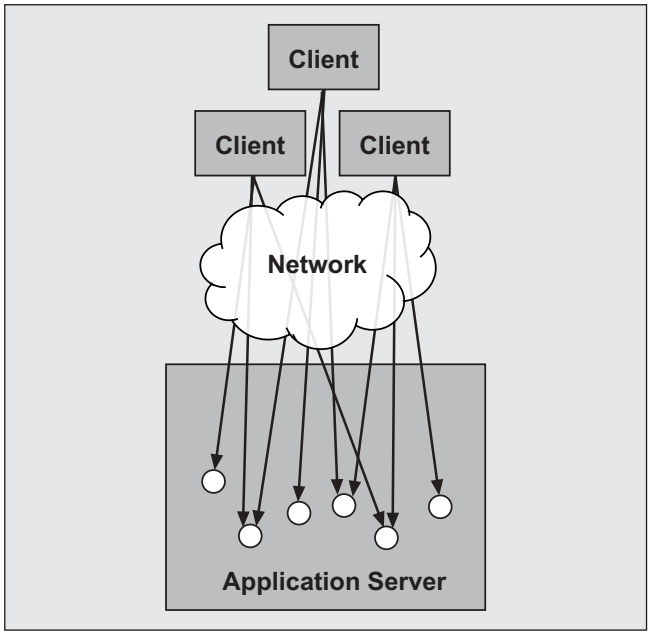
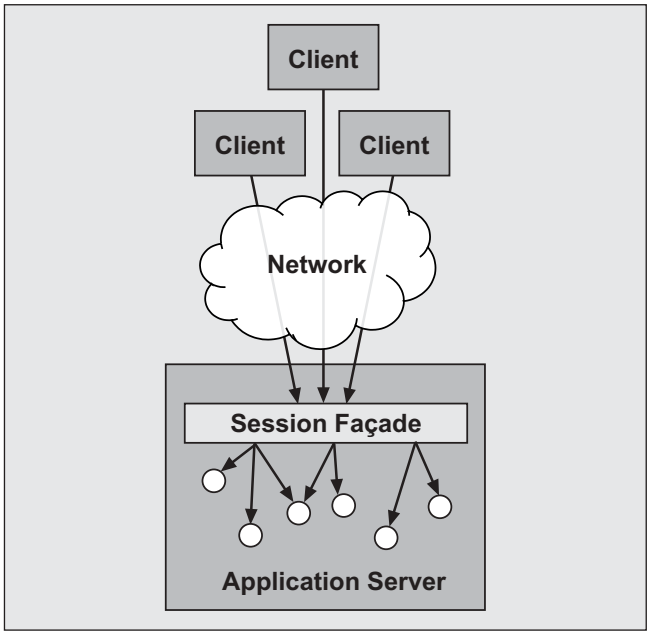


Figure 14 Executing Business Logic with the Session Façade Design Pattern



states that clients should never have access to entity beans explicitly. Entity beans should be wrapped in a layer of session beans (i.e., the Session Façade) as depicted in **Figure 14**.

The Session Façade enforces the execution of business logic in one network call and provides a clean layer for placing the business scenario in a single class. For our previous online banking scenario, this means that the logic for transferring money from one account to another will be encapsulated in a session bean with a method called *transferMoney(user, account1, account2, amount)*. This session bean is provided with all necessary parameters in a single network call (that's why it is called a "coarse-grained" call in comparison to the "fine-grained" calls of the first implementation). This not only optimizes network traffic, it's also good J2EE design practice to separate business logic from database access. This separation allows specialized teams to develop different parts of the application, and optimizes maintenance costs — for example, if you changed your database, you would only have to adjust the code in the database layer instead of modifying code in your business logic.

Design patterns are an important resource for saving development time and money. I won't delve any deeper into patterns here, since there are so many, but starting points for further research are the J2EE patterns home page (**http://java.sun.com/blueprints/corej2eepatterns/index.html**) or the home page for server-side application development at **http://theserverside.com**. You can also find books by searching for "design pattern," "EJB design pattern," "J2EE design pattern," or "best practices" on bookseller sites.

Recommendations for Ensuring a Successful J2EE Application Migration

Based on my experiences, here is a checklist of recommendations you can use to help ensure a successful J2EE application migration:

☑ Test your application using the AVK and correct the errors it reports. Ideally, you should integrate

this verification process into your developers' build process right from the beginning.

- ☑ Use automatic test tools, which ensure test result reproducibility and maximum code coverage.
- ☑ When designing complex J2EE applications, make use of established design patterns. Portable implementations of design patterns by companies or open source communities can often be used to save time and money.
- ☑ Check the deployment descriptors of your application for the original application server against proprietary extensions. If some are found, develop portable alternatives that don't use vendor-specific extensions to the J2EE specification.
- ☑ Use the SAP Web AS Deploy Tool to generate the EAR files. It's up to you to start with basic class files, pack them together in JAR and WAR files or with existing JAR and WAR files, and zip them into an EAR file. The Deploy Tool will then guide you through the individual deployment steps. For a detailed description of the Deploy Tool, go to the online help for SAP Web AS 6.30 or 6.40 at **http://help.sap.com** and navigate to *SAP NetWeaver Components → SAP Web Application Server → J2EE Technology in SAP Web Application Server → Development Manual → Deployment: Putting It All Together → Deploy Tool*. Remember that once you've migrated to SAP Web AS 6.30 or 6.40, you will use SAP NetWeaver Developer Studio instead.
- ☑ Before deploying the application, identify the global libraries upon which the application depends; you can reuse libraries that already exist on the target application server by referencing them in the Deploy Tool. If a special global library doesn't yet exist on the target system, but you know that it will be widely used by other applications, deploy this library via the Deploy Tool prior to deploying the application itself, so that you can reference it like the already-existing libraries. Existing libraries can be examined via the SAP J2EE Engine Administrator.
- ☑ Following deployment, test your application manually or by using an automated test tool like JUnit, HttpUnit, or Apache Cactus. (If you've developed automated tests for use with the AVK verification process, you can reuse those same tests here.) Follow the stack traces that pop up and repeat deployment and test steps after each correction. You don't have to repeat the AVK verification in each cycle (unless you have introduced the step in your normal development process), but you should reactivate it if the correction shows ambiguous behavior.

Conclusion

With its support of the J2EE 1.3 standard, SAP Web AS 6.30, and now 6.40, offer you the opportunity to harmonize your IT landscape by running your J2EE and business applications on the same platform — simultaneously reducing your maintenance and development costs, and leading to an optimized total cost of ownership. My hope is that with the lessons you've learned in this article, including why it might make sense to port your applications to SAP Web AS, the considerations involved, the potential pitfalls, and how to ensure application portability in the future, you won't have to let this opportunity pass you by.

Volker Stiehl received a degree in computer science from the University of Erlangen, Germany, in 1988. After working for a small company that specialized in computer process control, in 1992 he joined Siemens, where he initially worked on server-side architectures. With the advent of the Java programming language, Volker focused on J2EE-based integration and the integration of SAP R/3 systems with other legacy systems and databases. Currently Volker is working as an SAP NetWeaver consultant with special expertise in the SAP Web Application Server and SAP Exchange Infrastructure (XI) technologies, as well as the SAP cross applications (xApps) architecture. Volker can be reached at vstiehl@t-online.de.

