ten best practices for J2EE development

---

key learnings from a J2EE consulting project

Chris Peltz
Hewlett Packard, Co.
October 2001

10 best practices for J2EE development

key learnings from a J2EE consulting project

introduction

The software industry has developed a number of best practices and techniques for designing, developing, and deploying J2EE-based applications [ALUR, ROM, SIDE, PIC]. These best practices have been created to help guide the J2EE architect in designing software solutions that are robust, scalable, flexible, and maintainable. Recently, members of Hewlett Packard's software consulting organization had an opportunity to apply some of these best practices in evaluating a J2EE-based framework. This paper presents a few of the best practices that were helpful during this engagement. The techniques are organized into three categories: design, development, and deployment. While this is not intended to be an exhaustive list of best practices, it should provide the reader with a few important guidelines for J2EE development.

planning the design

1. **Provide the right level of documentation to fit the developer role.**
   Many development organizations are relying on outsourcing and distributed teams to develop their J2EE-based applications. With many of the developers remotely located from the design team, it is even more critical that sufficient design documentation be provided to the developer. In a typical n-tier architecture, there are often three developer roles required: user interface developer, business object developer, and application developer. Design documentation is usually created to meet the needs of the user interface or the business objects. Often overlooked are the specifications that map the user interface to the back-end business objects. At a minimum, the specifications should provide the following two pieces of information:

   1. **Field Mappings** - detailed mappings that relate each user interface field to the appropriate data element. These mappings can also help validate the design assumptions and can be used to populate data for testing purposes.
   2. **Business Logic** – this explains how to retrieve the right records from the right entity beans in the data. Once the entity beans have been identified, the appropriate business logic has to be developed to find the data and return it to the client.

   In addition to the design specifications, developers may also need access to the design model for the business objects. The design of the business objects can be done using UML and an associated modeling tool. Development teams should consider providing lightweight versions of these tools to the developer community. One such tool is TogetherSoft (www.togethersoft.com) ControlCenter, which is a Java-based modeling tool that can run from HP-UX, Windows, or other platforms.

   JavaDoc is another method used for documenting the classes, methods, and attributes of a J2EE project. If done well, this type of documentation can be very helpful to the developer building applications that use existing business objects. A development team should consider creating a set of standards that specify the guidelines for in-line documentation. Based on the team's needs, JavaDoc can include the purpose of each class, how to use it, any possible side effects, available methods and

attributes, cross references to related classes, and exception handling information. One should also consider package-level documentation, which can be used to describe each package, its use, and the list of classes contained within it. [JDOC] provides a very good starting point for developing JavaDoc standards.

That said, JavaDoc is only useful if the developer knows what they are looking for. A good analogy to this is a dictionary. A dictionary can be a very useful resource, but only if the reader knows the specific word or concept they are looking for. For the J2EE developer, JavaDoc is usually not sufficient to help them understand **how** to use the available business objects together. What's needed is a developer guide that explains how to construct an application and specifically covers what J2EE patterns or concepts to consider at each layer of the architecture.

2. **Consider the design of the user interface as well as the business objects.**
Because J2EE is often considered a back-end solution, development teams often focus on the business object design more than the user interface (UI) design. However, without the proper UI design, developers may end up spending more time than expected in developing the application. The important part of the design process isn't designing the look-and-feel – it's designing the relationships that exist between the various UI components. The goal is to minimize the dependencies that might exist between the UI elements. These dependencies are often created because of the data that has to be shared among two or more windows or views within the application.

A typical user interface screen might include multiple views, or panels, of data. The important part of the design is determining how these various views relate to each other. One approach is to have a higher-level parent view that is used to locate and manage the data, and a set of children views that display the data. A UI designer should attempt to maximize these parent-child relationships and minimize any sibling relationships (e.g., one view relying on data from another view). The following figure illustrates the difference between these two concepts:

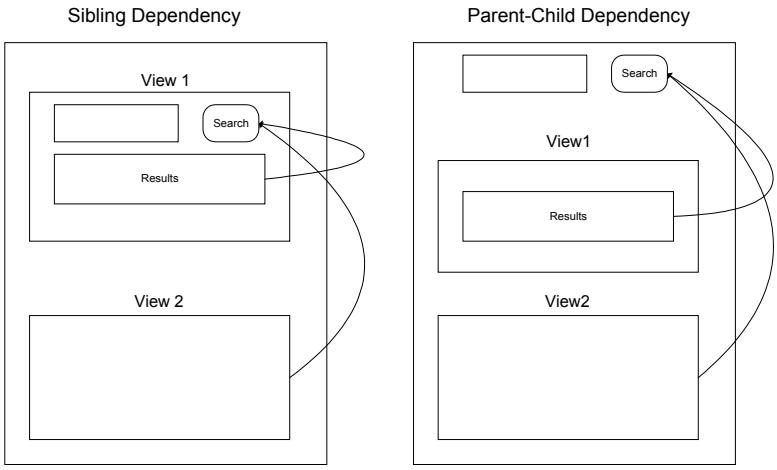

Figure 1. Managing Dependencies Between Views

By building the UI in this manner, one can encapsulate functionality and increase reuse of the graphical views to other parts of the application.

3. Provide a facility to make value objects viewable.

The ValueObject pattern can act as the model[1] for a business object, which can then be mapped directly to the user interface. If using this approach, a developer will often want the capability to view or display the contents of the value object. For example, this might be useful for verifying whether the right data is being returned by the business objects. There are a number of techniques a developer could use to do this. The developer could write a set of print() routines in their test program to dump out the contents of each type of value object. This would require a fair amount of coding with very little reuse. A second option would involve adding a toString() method to each type of value object, which would allow the developer to easily view the fields within the object. This is somewhat better, but it still requires the developer of each value object to implement a toString() method.

A more reusable approach is to introduce the notion of an abstract ValueObject that all other value objects would extend. At a minimum, this abstract class would require two methods: (1) an abstract method that returns a list of fields in the value object, and (2) a single toString() method that can generically create a String representation for any value object using the Reflection API. The following listing provides an illustration of this technique. The same approach can also be used to provide a generic toXML() method for each value object.

```
public class ValueObject extends AbstractValueObject {
    public int id;
    public String name;
    public String[] getFieldNames(){ return new String[] {"id","name"}; }
}

public abstract class AbstractValueObject {
    public abstract String[] getFieldNames();
    public String toString() {
        StringBuffer result = new StringBuffer();
        String[] fields = this.getFieldNames();
        Class c = this.getClass();
        for (int i = 0; i < fields.length;i++)
            result.append(fields[i] + ": " + c.getField(fields[i]).get(this) + "\n");
        return result.toString();
}
```

Listing 1. Creating a ValueObject toString() Method

4. Return collections of value objects instead of collections of remote objects.

Use of the ValueObject pattern does not completely eliminate unnecessary remote calls to the application server. In the example of a user interface that performs a wildcard search against a database, the results of that search might include a large number of records that meet the user's criteria. Using EJBs, this functionality is provided by the various finder() methods on each entity data object. In the EJB1.1 specification, all finder() methods are required to return a collection of remote interfaces[2]. If this collection were returned back to the presentation tier, the front-end would be required to make a separate remote call for each record found. Obviously, this could introduce significant performance overhead if a large number of objects have to be processed.

[1] Here, "model" refers to the Model-View-Controller (MVC) architecture, where the model typically represents the data, and the view represents the user interface.

[2] The EJB2.0 specification introduces the concept of business methods on entity beans. This will provide the capability to return data elements in any format.

The J2EE Patterns Catalog introduces the Session Façade pattern, which uses a session bean to "encapsulate the complexity of interactions between the business objects participating in a workflow" [ALUR]. Unlike the entity bean, which models a very fine-grained object, the session bean can be used to represent other fine-grained objects. Using this pattern, a developer could implement a service that transforms the collection of remote objects to a collection of value objects. This would allow the client to make a single request to retrieve all of the data necessary to populate the user interface table.

Furthermore, if the entity beans used in the J2EE application all implemented a similar interface, a development team can introduce a very simple and elegant way of performing this conversion. This interface could provide a single method called getValueObject() and each class that implements this interface would include the necessary code to construct and return their value object representation. With this in place, a converter utility can be written to automatically perform the conversion. The following listing demonstrates how this can be accomplished.

```
static public Collection convertEntityData(Collection old_list) {
    ArrayList new_list = new ArrayList();
    Iterator i = old_list.iterator();
    while (i.hasNext()) {
        Object o = i.next();
        if (o instanceof SomeInterface) {
             SomeInterface l = (SomeInterface) o;
             new_list.add(l.getValueObject());
        }
    }
    return new_list;
}
```

Listing 2. Converting a Collection of Remote Objects

5. **Use aggregate value objects to minimize the number of remote calls to the server.**
In addition to requiring multiple rows from a single entity, a J2EE application may also need to display data from multiply entity types on a single user interface screen. For example, an order fulfillment application might contain a single screen populated with customer data and a history of their orders. A J2EE application could be designed in such a way that the presentation layer makes a remote call to each entity bean required. However, this approach would enforce a tight coupling between the client and the business objects and could potentially increase network overhead. The J2EE Patterns Catalog [ALUR] has included the Value Object Assembler to help here:

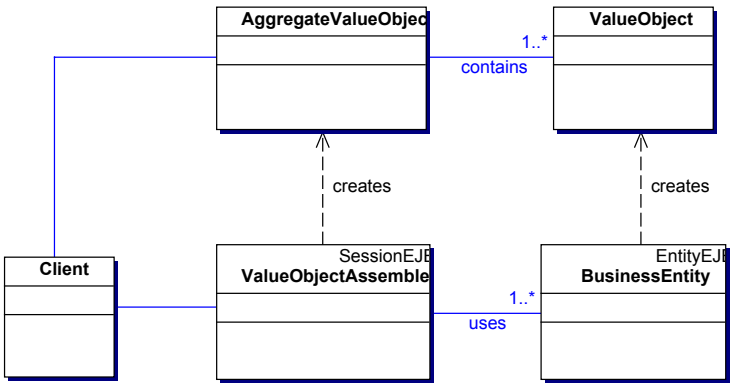

Figure 2. Value Object Assembler Pattern

Leveraging the Session Façade pattern, the Value Object Assembler pattern uses a Session EJB to act as an aggregator of the data. This assembler bean creates an aggregated value object that contains all of the required value objects. The higher-level aggregated object could be implemented as a simple object that contains other value objects, or possibly collections of value objects. The idea is that the

assembler session bean performs a number of lookups against the required entity beans, retrieves the value objects for each of the entity beans, and stores the resulting data into the aggregated object. This coarse-grained object can then be returned to the client application.

6. Carefully consider how your J2EE application will manage exceptions.

The design of a J2EE application must also consider how exceptions will be created and managed. Development organizations often overlook the value of a robust error handling mechanism. This becomes even more critical when designing an n-tier architecture where the presentation layer must process lower-level exceptions and present them to the user appropriately. The following figure illustrates a few techniques that are often used by development teams to manage exceptions:

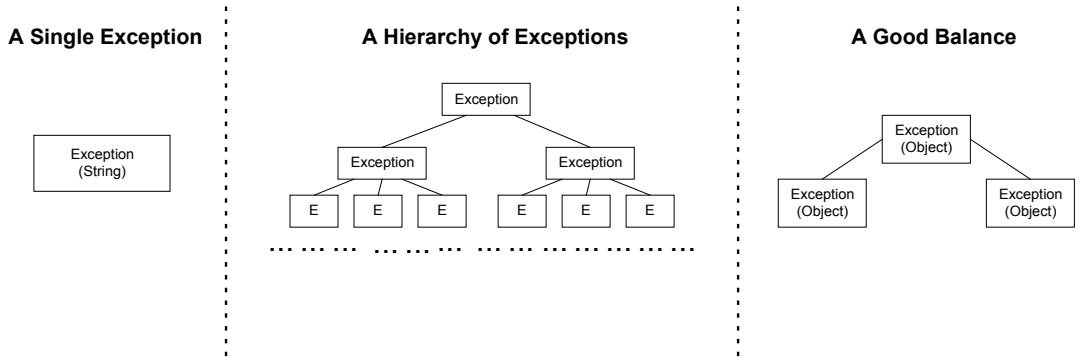

Figure 3. Strategies for Managing Exceptions

With the approach on the left, a single Exception class is used for handling all exceptions. The error message to describe the exception is placed in the String parameter. The advantage to this approach is that the presentation tier only has to catch one type of exception and display the message. This works fine until the message returned by the middle tier isn't appropriate for the user. In order to handle this case, the presentation tier would have to perform a string comparison on the error message and convert it to some other error message for the user. This would require the application to hardcode certain error messages, thus making the application very hard to maintain.

The middle option takes the other extreme. With this approach, Exception classes are created for every possible exception that can be thrown by the back-end. For example, for the exception, "Record Not Found", this architecture would require an Exception for every entity defined. While the presentation tier can more precisely handle each error, it also has to handle many more exceptions. What's needed is a balance between the two approaches. The option on the right demonstrates how this can be accomplished. The idea is to create enough Exception classes to define the major Exception categories within the J2EE application, and then further define these classes through the use of an Object parameter. This Object class might include an exception code, exception description, etc. Overall, this approach provides a better coupling with the presentation tier without sacrificing flexibility.

7. Use Policy objects to manage data validation at the business and presentation layers.

A J2EE application must also provide a robust mechanism for data validation. One approach is to introduce the concept of data policies for handling the validation and display of data elements within the application. These data policies are used to simplify validation both on the presentation and the business tier. For example, an AccountCodePolicy class might be developed to validate an account code entered for a customer order. This policy class might check that the code entered is exactly 3 characters and only contains alphanumeric characters.

The concept of data policies is actually based on the Strategy design pattern [GOF]. Essentially, this pattern allows a designer to separate an object (e.g., Account), from a particular behavior used by that object (e.g., AccountCodePolicy). In situations where an object is "complex" in nature, it might use many behaviors depending on the state of that object. With the Strategy pattern, a developer can easily add, remove, or change the default behavior associated with an object. This approach can also eliminate the need for conditional statements on the client side (e.g., if the account code validation is driven by the customer type code).

One of the benefits of using this pattern is that it reduces the need for developers to validate data directly on the presentation tier using hardcoded algorithms, rules, or string constants. Data objects can be associated to the appropriate data policy classes, and developers only have to invoke the appropriate validate() routine to check whether the data entered is valid. Using this policy mechanism can maintain data consistency across the user interface and can also improve code maintainability. Changes on the policy can be made in one place without requiring code changes to all affected components.

deploying the application

8. Use JavaDoc to simplify the development of EJB components.

One of the difficulties in using J2EE is managing all of the components required for each EJB. For example, one entity component may require the entity bean class, the remote and home interfaces, the primary key class, the deployment descriptor, other environment properties, and possibly a value object [ROM]. The developer has a number of options to manage these dependencies. They can manually create and maintain each component. They could also invest in an IDE that simplifies the development of these components, like TogetherSoft ControlCenter[3] or WebGain Visual Café[4]. However, this approach can also introduce some code maintenance issues if the code base is ever modified outside the IDE.

Another approach that is gaining momentum is the use of JavaDoc tags to specify the EJB information within the bean class. With these tags, a developer can specify information about the EJB component classes, including value object creation rules, mappings from the EJB attributes to the underlying database tables, and finder queries for deployment. A tool that leverages the Doclet API could then be developed to automatically create the dependent EJB files from the JavaDoc tags [POLL].

Luckily, there are a number of open-source tools available that provide this code generation capability from JavaDoc. EJBGen was one of the first EJB code generation tools made available. A similar tool called EJBDoclet was later developed and made available on SourceForge [BRUE]. With EJBDoclet, a developer could embed JavaDoc in their bean class and then automatically generate remote and locate interfaces, primary key classes, home interfaces, value objects, CMP classes, and deployment descriptors. This tool was later superseded by XDoclet, a more generalized tool that supports the generation of any type of files from JavaDoc encoded files. The tool currently provides templates for EJB code generation and web application files (e.g., web.xml and taglib.tld).

The following illustrates how EJB component information can be specified for a bean class. This example specifies information on about the bean class, including the bean type, JNDI information,

[3] See www.togethersoft.com for more information on the TogetherSoft ControlCenter product.

[4] See www.webgain.com for more information on the WebGain VisualCafe product.

primary key fields, finder methods, and remote classes. EJB tags can also be specified at the method level. This might be used to specify whether a field is part of the primary key, whether it is persistent, etc.

```
/**
 * This is an account bean. It is an example of how to use the
 * EJBDoclet tags.
 *
 * @see Customer Accounts are owned by customers, and a customer can
 * have many accounts.
 *
 * @ejb:bean   name="bank/Account"
 *             type="CMP"
 *             jndi-name="ejb/bank/Account"
 *             primkey-field="id"
 * @ejb:finder signature="Collection findAll()"
 *             unchecked="true"
 * @ejb:interface remote-class="test.interfaces.Account"
 */
```

Listing 3. EJBDoclet Example

There are clearly many benefits to using this approach, including enhanced usability and code maintainability. The goal is that the developer can focus on the business logic, without having to worry about the "plumbing".

9. **Keep in mind the various third party products used in development.**
   The APIs, libraries, and frameworks provided internally by an organization are not the only tools used by developers. During the lifecycle, a developer may also be exposed to build tools, IDEs, application servers, multiple JDK versions, version control systems, code generation tools, and possibly other J2EE-based frameworks. In many cases, new developers may not be familiar with all of these tools, yet organizations often fail to provide sufficient information to the developer on how to use them. Appropriate documentation should be collected for all third party tools, including information on the directory structure used, configuration files, commands to start and stop each tool, a quick reference guide to get started, and a troubleshooting guide for debugging common problems.

   Another common problem is that the documentation provided on each tool is not always properly versioned. In addition to placing the application code under source control, an organization should consider versioning any documentation required for each tool, as well as any configuration files required for the developer to properly use the tool in the environment. For example, an application server like HP Bluestone typically requires a set of configuration files that define JNDI and JDBC connection information. Typically, these configuration files aren't versioned with the application itself, which can lead to confusion because developers may not know the proper database that they should connect to for testing purposes. The process used for managing source control should be enhanced to enable versioning of third party documentation and configuration information so that a developer can easily checkout, build, deploy, and test the J2EE application.

10. **Use Ant to manage the J2EE development and build processes.**
    A well-defined build process can ensure that the software is built in a consistent manner each time. This becomes even more important as the build process becomes more complex with J2EE technologies. Ant is one tool that has been introduced to aid in managing the build process [CYM]. The tool provides the following advantages over "make": (1) because it is written in Java, it eliminates the need for specific shell scripts or BAT files for specific platforms, (2) it is easily extensible with many extensions already available through open-source, and (3) it is highly configurable and easy to use. Essentially, Ant is driven from an XML-based build file that is composed of a set of targets. A single target might range from initializing the build, compiling the code, deploying it, or possibly

testing it. Each target is then composed of a set of tasks, where an individual task might be making a directory, deleting files, running JAVAC, or executing the JAR command.

Ant can then be extended and customized through the definition of new tasks. Developers are free to create their own tasks based on their needs. For example, a developer may want to customize the build process to automatically restart the application server after each build. A number of useful extensions have already been built, including tasks for generating JavaDoc, for automating the process of checking Java code against a set of standards, and for automatically generating EJB and deployment descriptors with XDoclet.

Here are some important best practices as it relates to using Ant for J2EE development:

- Use a single build process for both command-line builds and IDE builds. This might require using an extension that integrates the IDE with Ant. However, in doing this integration, try not to place any IDE-specific tasks within the build scripts. Doing this will make it difficult to perform the build on non-development machines.
- Specify properties for commonly used directories, files, or class libraries, and place these properties in a global build file for all applications to reference. For example, references to the ubs.jar file required by HP Bluestone can be placed in a global build.xml file.
- Provide a deploy target to simplify the deployment of your J2EE application to the targeted application server. This can minimize developer impacts if the application server vendor changes.
- If multiple targets are specified in the build file, provide sufficient documentation on the dependencies and relationships between each target.
- Don't place any platform-specific information in the build scripts. Developers often hardcode source or target locations with drive letters (e.g., C:\projects), which make it difficult to perform the build on other platforms.
- Use Ant to unit test your J2EE application [DAV].For example, a target can be specified that automatically runs the Junit TestRunner tool. This approach can help catch bugs sooner in the development cycle.

conclusion

This paper has presented ten important best practices as it relates to the design, development, and deployment of J2EE applications. While there are many design decisions that must be considered, these guidelines can help simplify the architecture while still maintaining flexibility and robustness. The key points that the reader should take away from this paper are:

1. Provide the right level of design documentation for the J2EE developer, with the understanding that there are different roles within the developer community.
2. Leverage value objects to minimize the communication between client and server, thereby improving overall scalability and performance of the system.
3. Design the architecture with a robust exception handling mechanism. This can influence the ease of integration between the application tiers.
4. Consider the use of third party and open-source tools to simplify the development and deployment of J2EE-based components.
5. Introduce a well-defined build process to improve reliability and consistency of the deployment process.

REFERENCES

[ALUR] Alur, Deepak, J. Crupi, and D. Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Sun Microsystems Press, 2001.

[BRUE] Bruell, Ingo. *How to Generate Enterprise Java Beans with EJBDoclet (XDoclet)*. http://www.jboss.org/documentation/HTML/ch10s29.html.

[CYM] Cymerman, Michael. *Automate your build process using Java and Ant*. JavaWorld (www.javaworld.com), October 2000.

[DAV] Davis, Malcolm. *Incremental development with Ant and JUnit: Using unit test to improve your code in small steps*. IBM developerWorks, November 2000.

[GOF] Gamma, Erich et. al. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[JDOC] Sun Microsystems. *How to Write Doc Comments for the Javadoc Tool.* http://java.sun.com/j2se/javadoc/writingdoccomments.

[PIC] Picon, Joaquin et. al. Design and Implement Servlets, JSPs, and EJBs for IBM WebSphere Application Server. IBM Redbooks, 2000.

[POLL] Pollack, Mark. *Code Generation Using JavaDoc*. JavaWorld (www.javaworld.com), August 2000.

[ROM] Roman, Ed. *Mastering Enterprise JavaBeans and the Java 2 Platform, Enterprise Edition*. John Wiley & Sons, Inc., 1999.

[SIDE] Roman, Ed. *EJB Design Patterns*. www.theserverside.com (to be published).