The Complete Reference

J2EE

Chapter 3

J2EE Best Practices

With the onset of Internet technology, systems designers and programmers faced challenges similar to those faced by the men and women who worked towards making space flight a reality. The game had changed. Old technology and design methods were no longer adequate to build the new, highly efficient systems needed to maintain a competitive edge.

The old way of designing and building systems lacked the wherewithal to deliver highly efficient, enterprise-wide distributive systems to meet the 24 hours, 7 days a week instant demand expected by thousands of concurrent users. Technologists at Sun Microsystems, Inc. and other technology organizations threw out the book and rewrote the rules for system development.

The J2EE is one of their initial steps towards devising a better technology to create advanced distributive systems. But J2EE only provides the framework within which to build these systems. New design and programming techniques were needed to address the demands of corporations. Once again technologists had to think out of the box and devise new ways of building a system. And once again new concepts in design were born.

These concepts fall within two general categories: best practices and patterns. Best practices are proven design and programming techniques used to build advanced enterprise distributive systems. Patterns are routines that solve common programming problems that occur in such systems. You'll learn best practices in this chapter and then learn patterns in Chapter 4.

Enterprise Application Strategy

Not too long ago information technology departments of many corporations viewed an application in the same way some of us view an automobile. That is, an application and an automobile both have a useful life, after which they are replaced. This throwaway philosophy has its merits in that the corporation will have different needs five years hence, and therefore it makes economic sense to design an application to meet those needs rather than retrofitting the existing application.

The throwaway philosophy works well for applications that services a small group of users such as applications focused on departmental needs. However, this philosophy loses its economical and practical sense when applied to an enterprise application.

An enterprise application is a mission-critical system whose continual successful operation determines the corporation's success. This means that an enterprise application is complex in nature and meets the needs of a diverse, constantly changing division of a corporation. The bottom line is that building an enterprise application is time-consuming, and once the enterprise application is implemented successfully, few in the corporation want to tinker with a critical application that works fine.

Corporations face the realities of an enterprise application about three years after the application becomes operational. The enterprise application starts to age. Corporate needs change to meet new challenges in the marketplace and so the enterprise application must change. These changes occur gradually with the creation of a few new reports, and then maybe the addition of new fields and tables in key databases.

These are relatively minor changes, especially when compared to creating a new enterprise application. However, as the application reaches its five-year anniversary these minor changes begin taking their toll, mainly because of a philosophy that seems to be prevalent among programmers whose job it is to maintain legacy applications. These programmers have one and only one objective—make sure change to the legacy works without negatively affecting the application.

Rather than tinker with code that was written by another program and is working fine, the programmer typically writes a routine separate from existing code and then calls the routine using a hook in the legacy application. This results in minimum changes to the existing application and the new routine can easily be shut off should problems arise after the new routine is implemented.

While this is a clever and successful technique that has been used by programmers for decades, this technique inadvertently creates an application that looks like a bowl of spaghetti. The application isn’t maintainable, and it is for this reason that the corporation is practically forced to re-create the enterprise application. Simply said, it is less expensive to re-create the application than it is to pay a programmer to study and change spaghetti code.

Besides the maintenance nightmare of a legacy system, information technology departments also realize that departmental applications that once had a limited scope were useful to other departments throughout the company. These applications were cloned, which saved a corporation the cost of developing a similar application. However, cloning caused two additional problems. First, each clone was slightly modified to meet the needs of its users, and multiple programmers were maintaining the same basic application.

A New Strategy

Corporations have taken on a global strategy where business activities and applications that support those activities are dispersed to business units throughout the world. However, rather than working independently, these business units transfer knowledge among other business units within the corporation to give the whole corporation a competitive edge in the global market.

In addition, corporate management approaches business similar to how football team owners approach the game of football. Management sets the main objectives for the corporation and then assembles a team to reach the objectives. On the gridiron, players are hired, each having a specific role that helps the team reach its goal—win the game. In the corporate world, business units are the players. Business units are purchased, merged, and sold to come up with a cohesive organization to reach the corporate goal—earn profits.

All enterprise applications must interface with each other to assure that information can be shared amongst business units. That is, all applications and systems have to work together and have the flexibility built into the architecture so that an enterprise application is incrementally changed to meet new business demands without having to be reconstructed.

This eliminates the lag that occurs with the throwaway philosophy. No longer will business units need to suffer the pain that occurs when changes to the application are not made in a timely manner because the application is not maintainable. The new corporate information technology strategy is to employ an architecture within which enterprise applications can coexist and can be incrementally developed and implemented.

Information technology departments realize that many applications used by business units have the same functionality. This means that there is duplicate effort within the corporation to maintain those applications. The new strategy of making enterprise applications interoperable has led to a new concept used to build enterprise applications.

An enterprise application has become a collaborative effort that is divided into two roles—building functional components and assembling functional components into an enterprise application—which enables functional components to be shared amongst many enterprise applications.

The Enterprise Application

The term "enterprise application" is elusive since practically any application used by more than one person to conduct business could be considered an enterprise application. And yet looking at an application we can easily determine if it is an enterprise application or not by asking the question, "Does the application service the entire corporation or a small group of users within the corporation?"

It is important that there is a clear understanding of what is meant by the term "enterprise application" because techniques used to design and build an enterprise application may not be the best way to develop a smaller application. This is because an enterprise application must deal with performance, security, and other issues that are not found in other kinds of applications.

For the purpose of J2EE, consider an enterprise application to be one that

- Is concurrently used by more than a handful of users
- Uses distributive resources such as DBMS that are shared with other applications
- Delegates responsibility to perform functionality among distributive objects
- Uses web services architecture and J2EE technology to link together components (i.e., objects) that are dispersed throughout the corporate infrastructure

Unlike many smaller applications, an enterprise application is highly visible within a corporation whose success greatly depends on the application's successful operations. This results in corporate users having high expectations from an enterprise application. They want the enterprise application to

- Be available 24 hours a day, 7 days a week without any downtime
- Have an acceptable response time even in the face of increasing usage
- Have the flexibility to be modified quickly without requiring a redesign of the application
- Be vendor independent
- Be able to interact with existing systems
- Utilize existing system components (i.e., objects)

This means developers have to create an enterprise application that is available and scalable to adjust to increases and decreases in demand. The application must be extensible and maintainable so new business rules can be easily added to the application. In addition, the application must be portable so the company isn't locked into a specific vendor. Developers also must build in interoperability so the enterprise application interacts with other applications and is able to reuse existing code.

Clients

As discussed in the previous chapter, J2EE is organized into tiers, the first of which is the Client Tier. Software working on the Client Tier has several functions, many of which are easily developed by programmers. However, there are a few functions that are less intuitive to program and therefore pose a challenge to programmers. These functions are to

- Present the application's user interface to the person who is using the application
- Collect and validate information from the person using the application
- Control the client's access to resources
- Prevent clients from sending duplicate requests

Client Presentation

An enterprise application uses a thin client model where nearly all functionality associated with the application is contained on the server-side rather than with the client. Thin clients handle the user interface that presents information to the user and captures input from the user. There are two strategies for building presentation functionality into a client. These are to use a browser-based user interface or to create a rich client (i.e., applet or application) where a graphical user interface is programmed into the client. Each has its advantages and disadvantages.

A browser-based user interface is written in either the Hypertext Markup Language (HTML) or the Extensible Markup Language (XML), which is used by an XML-enabled browser to interact with the user. The browser runs on the client machine and interprets HTML or XML code into elements of a user interface.

The browser-based strategy enables easy implementation of the presentation layer of an enterprise application because details of presentation such as user interface controls and event handling are built into the browser. Furthermore, browsers provide a standardized user experience that incorporates elements that are intuitive to use. That is, little or no training is needed for a person to use a browser-based application.

However, the browser-based strategy has disadvantages too. First, the developer doesn't have exact control over presentation to the user. Instead, the developer suggests user interface elements to the browser, such as font, color, and position of text and images. The browser controls how these elements are displayed. Developers who use the browser-based strategy must test the enterprise application with various browsers and browser versions to be sure that the presentation is acceptable.

Another disadvantage of the browser-based strategy is the presentation is limited to interactions that can be implemented using a markup language or plug-ins, which limits the design of the user interface. That is, features that cannot be written in a markup language or provided by a plug-in cannot be implemented in the application.

Still another disadvantage is the presentation layer is server-side dependent. This means the application accesses the server more than if a richer client strategy was used to create the user interface. Practically each time an event occurs in a browser-based presentation, the browser must access the server, which might decrease performance and response time for the user.

A browser-based strategy typically uses the HTTP protocol. HTTP is a stateless protocol, so the developer must have a strategy for maintaining session state on the server. This situation can become complex if the system requires failover support. Most "out-of-the-box" technologies for managing session state do not replicate the session.

In contrast, the richer client strategy gives a developer total control over elements of the presentation and event trapping. That is, a developer can use WYSIWYG to create the presentation and isn't limited to only events that are trapped by the browser.

In addition, a richer client accesses a server only as needed and not in response to nearly every event that occurs in the presentation. This might result in fewer server interactions and increase response time because there are fewer messages sent to the server.

Best Practice—Developing a Client for Enterprise Application *The best practice when developing a client for an enterprise application is to use the strategy that is most appropriate for each aspect of the presentation. That is, both strategies can be used for pieces of the presentation of an enterprise application. Use the browser-based strategy for simple presentations and the richer client strategy whenever more complex presentations are necessary that cannot be adequately handled directly by a browser.*

Client Input Validation

Information that is entered into a client by a user should be validated, depending on the nature of the information. Details of the validation process are application dependent; however, the developer has two places where validation can occur: with the client or on the server side.

A developer can implement three kinds of validation strategies (see Figure 3-1): syntactic, lexical, and semantic. Syntactic validation determines if information that consists of several related values is well formed, such as time that is composed of hours and minutes. Lexical validation looks at a single value to assure that the type of value corresponds to the type of data that is expected by the application. For example, an hour value must be an integer to pass a lexical validation. Semantic validation examines the meaning of the information to determine if the information is likely to be correct. Semantic validation determines if the value of the hour is less than 24 and greater than or equal to 0.

There are three fundamental factors that must be considered when designing validation procedures for an enterprise application. These are to avoid duplicating the validation procedure within the application, provide the user with immediate results from the validation process, and minimize the effect the validation process has on the server.

Duplication of the validation process can become a maintenance nightmare whenever rules for validation change. Ideally, validation rules should be applied by one object that is called whenever the validation process is needed. Duplication problems can occur if the validation process is built into client software because there might be many instances of the client software.

Users require nearly instantaneous feedback as to the validity of the data whenever they enter information into a client. Feedback should indicate that the data is valid or

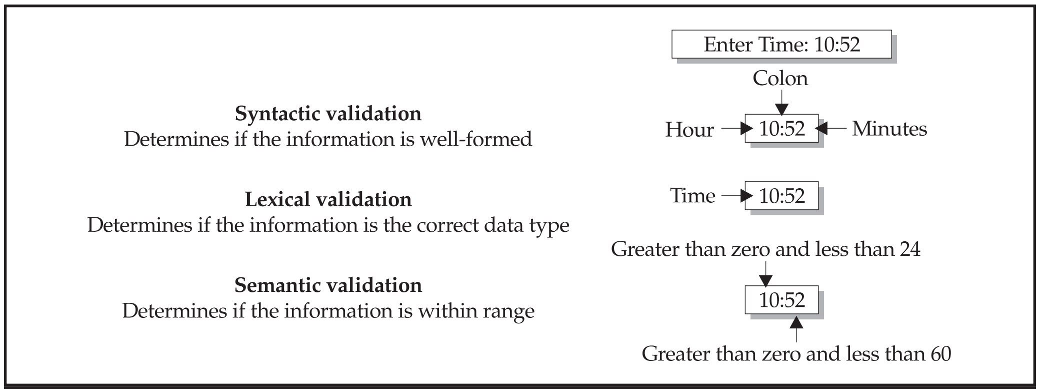
Enter Time: 10:52
Colon
Syntactic validation
Determines if the information is well-formed
Hour
10:52
Minutes
Lexical validation
Determines if the information is the correct data type
Time
10:52
Greater than zero and less than 24
Semantic validation
Determines if the information is within range
10:52
Greater than zero and less than 60

Figure 3-1. *Syntactic, lexical, and semantic are three validation strategies that can be used in an enterprise application.*

invalid. Any delay providing feedback to the user can lead to a poor user experience with the application.

A common source of delay is when a developer uses a server-side validation process. This is because a conflict might arise if multiple applications concurrently call the object that validates data. A request to validate data might be queued (see "The Power of Threads" later in this chapter) until another application's data is validated. There are a host of other reasons for delays on the server side. These include heavy network traffic and slow hardware components such as the server itself.

However, if designed properly, you shouldn't have any threading or concurrency issues on the server. Validation is typically performed by regular expressions. The Pattern class in the JDK1.4 is also thread safe. This means you compile the pattern once and use it with any number of processes.

In this case, Pattern is declared as a static member of the validation class to ensure only one copy is compiled. Matcher classes are created as required to match against the Pattern.

Best Practice—Developing a Validation Strategy *The best practice when developing a validation strategy is to perform as much (if not all) of the validation on the server side. Avoid validating on the client side because client-side validation routines frequently must be updated when a new version of the browser is released.*

Also, it is easy to bypass client-side scripts used in the validation process. Someone with minimal knowledge of protocols can save the page, delete the JavaScript, and then resubmit the form. Another concern is that validation scripts may never load because they are sometimes filtered by the company's firewall.

The second step of the validation process is to perform more sophisticated analysis of the information, such as verifying that a customer's account number is valid and that the customer is in good standing with the company. These validation rules are likely to change during the life of the enterprise application and therefore server-side software should perform this validation.

Best Practice—Reducing the Opportunity for User Error *Another best practice is to reduce the opportunity for a user to enter incorrect information by using presentation elements that limit choices. These elements include radio buttons, check boxes, list boxes, and drop-down combo boxes. Information that is collected using those elements doesn't have to be validated because the application presents the user with only valid choices. There isn't any opportunity for the user to enter invalid data.*

Client Control

In practically every enterprise application, clients are restricted to resources based on the client's needs. The scope of resources a client can access is commonly referred to as a *client view*. A client view might consist of specific databases, tables, or rows and columns of tables. There are two ways for a client view to be defined: through embedding logic to define the view into the application, or by using a controller component that is known as a resource guard.

The resource guard (see Figure 3-2) is a component that resides outside of the application that receives requests for resources from all applications. A request for a resource contains a client ID that is compared to the client's configuration. Access is either granted or rejected based upon access rights within the client's configuration.

Another method that is useful whenever only database access is required is to group together users who have similar needs into a group profile, then assign permissions to the group. In this way, the DBMS manages security directly without you having to write your own security routines.

Best Practice—Using Security Measures *The best practice is to use security measures that exist in DBMS or networks, where feasible. However, if this isn't the best alternative, use a resource guard rather than embed the definition of a client view into the application, unless only one client uses the resource. This is because multiple clients and applications can use the resource guard to access shared resources. In this way, the logic to define the client view isn't duplicated in multiple applications.*

As discussed previously in this chapter, requirements for an enterprise application are in constant change, even after the application is implemented. This means the application might be the sole user of a resource when the application is placed in production, but will need to share the resource at some point in the future.

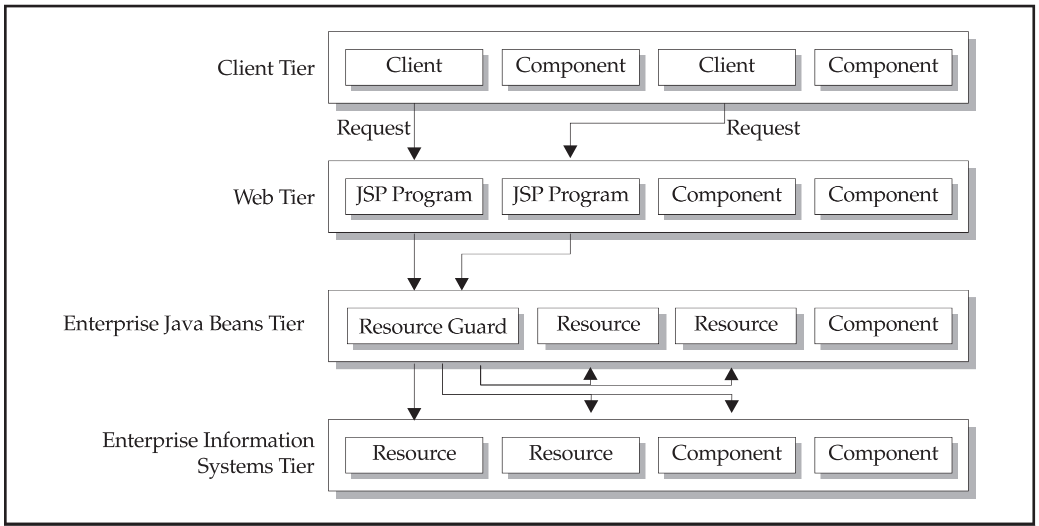

Figure 3-2. *A resource guard is a component outside the application that manages access to restricted resources.*

Best Practice—Working with a Resource that Becomes Sharable *The best practice to use when a resource becomes sharable is to remove the embedded logic that defines the client view from the application and place the logic into a resource guard. In this way, the logic is preserved while still providing flexibility to the application.*

A resource guard is a component that is shared with other applications. Therefore, the developer must construct a resource guard using one of the techniques used to share code within J2EE. Three of the more common ways are to build a resource guard using a JavaServer Page, a servlet, or Enterprise JavaBeans.

Best Practice—Using Resource Guards *The best practice is to use an Enterprise JavaBean as the resource guard and use security mechanisms that are already in the web container and resource, rather than build the logic for the resource guard from scratch.*

Once the client view is defined, the developer must determine a strategy for implementing the client view. There are two commonly used strategies: the all-or-nothing strategy or the selective strategy. The all-or-nothing strategy requires the developer to write logic that enables or prevents a client from accessing the complete resources. Simply said, the client can either access all features of a resource or is prohibited from accessing the resource entirely.

In contrast, the selective strategy grants a client access to the resource, but restricts access to selected features of the resource based on the client's needs. For example, a client may have read access to the table that contains orders, but doesn't have rights to insert a new order or modify an existing order.

Best Practice—Using the Selective Strategy *The best practice is to use the selective strategy because this strategy provides the flexibility to activate or deactivate features of a resource as required by each client. The all-or-nothing strategy doesn't provide this flexibility. This means the developer will need to replace the all-or-nothing strategy with the selective strategy after the application is in production, should a client's needs change. In addition, the selective strategy can also be used to provide the same features as provided by the all-or-nothing strategy. That is, the developer can grant a client total access to a resource or prohibit a client from accessing the resource.*

Duplicate Client Requests

A common problem with thin client applications is for the client to inadvertently submit a duplicate request for service, such as submitting a duplicate order. There are many ways a client can generate a duplicate request, but they all stem from the same source, which is the browser user interface.

A browser is the user interface used in a thin client application. However, the browser contains elements that can lead to a duplicate request being sent. Namely, the Back and Stop buttons. The Back button causes the browser to recall the previously displayed web page from the web server. The Stop button halts the implementation of a request. This means the browser processes some, but not all, of the requests.

Normally, the selection of these buttons has minimal consequence to the client because the browser either displays an unwanted page (previous page) or displays a partial page. In both cases, the user can easily correct the situation.

However, a problem might occur if the client is sending a web form such as an order form to the Web Tier. Let's say that the user submits the order form. The application generates a confirmation web page that the browser displays. The user then inadvertently selects the browser's Back button, which redisplays the order form. This might be confusing and cause the user to resubmit the order form, thinking that a snafu occurred. In reality, two orders are submitted.

Best Practice—Flushing the Session *The best practice is to flush the session explicitly and updating the session object without waiting for the page to complete. The session is still available and checks can be made on the server to see if the form was previously submitted.*

Let's say the beginning of the JSP updates the session state to indicate the form was submitted and processed, but the server terminates the JSP before it completes. This can happen when the user hits the Stop button or the ESC key, causing the process on the server to be terminated. The session state object may not be updated. Explicitly updating the session object without waiting for the page to complete alleviates potential problems (Figure 3-3).

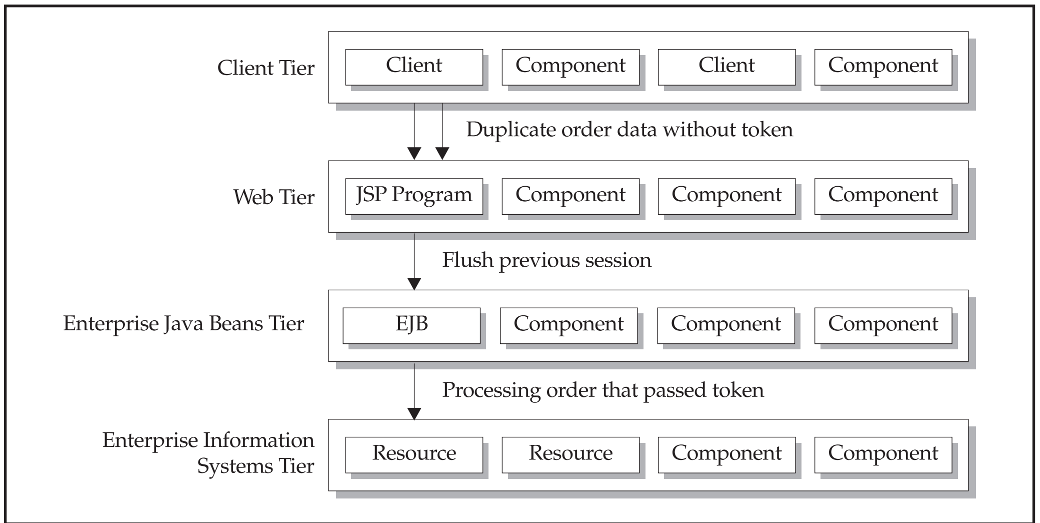

Figure 3-3. *Flush the previous session as the first action the program takes when a new form is submitted. This avoids duplicate sessions from occurring.*

Sessions Management

A web service-based enterprise application consists of distributive services (i.e., components) located on J2EE tiers that are shared amongst applications. A client accesses components by opening a session with the Web Tier. The session begins with an initial request for service and ends once the client no longer requests services.

During the session, a client and components exchange information, which is called session state. Practically any kind of information can be a session state, including a client's ID, a client's profile, or choices a client makes in a web form.

A component is an entity whose sole purpose is to receive information from a client, process that information, and return information to the client when necessary. Information used by a component is retained until the request from the client is fulfilled. Afterwards, information is destroyed. A component lacks persistence. Persistence is the inherent capability of retaining information (session state) between requests.

This means that it is up to the enterprise application to devise a way to maintain session state until the session is completed. There are two common ways to manage session state: on the client side or server side on the Enterprise JavaBeans Tier.

Client-Side Session State

Session state can be maintained by the client rather than on the server using one or a combination of three techniques. These are by using a hidden field in an HTML form, by rewriting URLs, and by using cookies.

An enterprise application typically uses an HTML form to collect information from a user. An HTML form can contain many elements. The more commonly used elements are text, fields, and buttons. Text consists of characters that appear on the form such as the title of the form and instructions for completing the form.

A field is the place on the form where the user enters data. Each field has a field name that uniquely identifies the field and a value (see Listing 3-1). There are several kinds of fields, including drop-down combo boxes that list valid entries, radio buttons, check boxes, and free-form text fields. A button is an image selected by the user to submit or clear the form.

Listing 3-1 HTML code that creates a form that contains a field called FIRSTNAME.

```
<INPUT TYPE="TEXT" NAME="FIRSTNAME" SIZE="40">
<INPUT TYPE="SUBMIT" VALUE="SUBMIT">
<INPUT TYPE="RESET" VALUE="CLEAR">
```

When the user selects the Submit button, the browser extracts the field names and field values from the form and assembles them into a query string (see Listing 3-2). The browser then calls a component on the Web Tier, which is usually a JSP program or servlet, and passes field names and field values as parameters to the component.

The component then processes this information. Once processing is completed, the component dynamically generates a web page that may contain another form, depending on the nature of the application.

Listing 3-2 Here is a query string created when the browser extracts the field name and field value from Listing 3-1 assuming the value of the field is Jim Keogh.

```
www.mysite.com/jsp/myjsp.jsp?FIRSTNAME=JimKeogh
```

Hidden Field

The component can include in the HTML form a field that isn't displayed on the form. This field is called a hidden field. A hidden field is similar to other fields on the form in that a hidden field can hold a value. However, this value is assigned to the hidden field by the component rather than by the user.

A hidden field is treated the same as other fields on the form when a user selects the Submit button (see Listing 3-3). That is, the name of the hidden field and the value of the hidden field are extracted from the form along with the other fields by the browser and sent as a parameter to the component. This means that session state can be retained by assigning the session state as a value to one or more hidden fields on each form that is used during the session.

Listing 3-3 Here is the same HTML form as shown in Listing 3-1; however, a hidden field has been placed into the form.

```
<INPUT TYPE="HIDDEN" NAME="AccountNumber" VALUE="1234">
<INPUT TYPE="TEXT" NAME="FIRSTNAME" SIZE="40">
<INPUT TYPE="SUBMIT" VALUE="SUBMIT">
<INPUT TYPE="RESET" VALUE="CLEAR">
```

Best Practice—Using Hidden Fields *The best practice for using a hidden field to maintain session state is to do so only when small amounts of string information need to be retained. Although using a hidden field is easy to implement, it can cause performance issues if large amounts session of state are required by the application. This is because the session state must be included with each page sent to the browser during the session regardless if the session state plays an active role on the page.*

In addition, hidden fields contain only string values and not numeric, dates, and other data types, which means the component might need to convert the string values into more appropriate values each time the page is sent to the component. Furthermore, session state is exposed during transmission. This means session state that contains sensitive information such as credit card numbers must be encrypted to secure the information during transmission.

URL Rewriting

URL rewriting is another strategy for managing session state. A URL is the element within the web page that uniquely identifies a component and is used by the browser to request a service. The browser can attach to the URL field names and field values that are passed to the component if the component requires this information to process the request. This is what happens when a user submits an HTML form, as discussed in the previous section.

URL rewriting simulates the routine the browser performs to extract field names and field values from an HTML form. That is, the developer places field names and field values into a query string as part of the URL statement (see Listing 3-4). When the user selects the hyperlink that is associated with the URL statement, the browser calls the component identified by the URL and passes the component field names and field values that are contained in the URL statement.

Listing 3-4 Sample of a URL rewrite that includes a customer ID placed in the hyperlink that calls a JSP program.

```
<P> <A HREF = "http://www.mysite.com/jsp/myjsp.jsp?AccountNumber=1234>
Click to place a new order</A></P>
```

Best Practice—Using URL Rewriting *The best practice is to use URL rewriting to retain session state whenever an HTML form is not used by the client. For example, the user might place an order using an HTML form. The order contains the user's account number along with other information.*

After the order is processed, a JSP program generates a confirmation page that is displayed by the browser. The confirmation page doesn't contain an HTML form, but does contain a hyperlink that is associated with a component that generates a new order form. The hyperlink asks the user if he or she wants to place another order. If the user selects the hyperlink, the browser calls the component and a new order form is displayed on the screen.

The developer might rewrite the URL that is associated with this hyperlink to include the name of the account field and the user's account number. In this way, the account number is automatically submitted to the component that generates a new order form when the user selects the hyperlink.

There are also several critical disadvantages of URL rewriting. The most important disadvantage is that maintaining session state depends on the client's machine. Problems with the client machine might cause the application to lose the session state. Also, including session state with every page requires the system to process and transmit more code than if cookies or server-side session state management were used.

Cookies

A cookie is a small amount of data (maximum of 4KB) that the enterprise application stores on the client's machine. A cookie can be used to store session state. The developer can create a JSP program that dynamically generates a page that writes and reads one or more cookies. In this way, session state persists between pages.

Best Practice—Using Cookies *The best practice is to use a cookie only to retain minimum data such as a client ID and use other techniques described in this section to retain large amounts of information. A developer must also implement a contingency routine should the user discard the cookie or deactivate the cookie feature.*

There are two major disadvantages of using cookies to retain session state. First, cookies can be disabled, thereby prohibiting the enterprise application from using a cookie to manage session state.

The other disadvantage is that cookies are shared amongst instances. This means a client might run two instances of a browser, creating two sessions. All instances access the same cookie and therefore share the same session state, which is likely to cause conflicts when processing information because the wrong session state might be processed. And of course, the user can always view the contents of the cookie using a basic text editor, making information stored in the cookie insecure.

Server-Side Session State

As discussed in the previous section, storing session state on the client side has serious drawbacks, most of which center on the dependency on the client's machine. That is, session state is lost if a client's machine fails.

An alternative to maintaining session state on the client side is to store session state on the server. Typically, the information technology department of a corporation goes to extremes to assure that the server and backup servers are available 24 hours a day, 7 days a week, which is not the treatment given to client machines.

Best Practice—Using the HttpSession Interface *The best practice is to maintain session state on the Enterprise JavaBeans Tier using an Enterprise JavaBean or on the Web Tier using the HttpSession interface. Each session state is assigned a session ID, which relates the session state with a particular client session. The session ID is used whenever the session state is written to or retrieved from the server.*

This provides the most reliable way to save and access session state and is scalable, thereby able to handle multiple sessions and various sizes of session state. It also decreases the vulnerability to someone inadvertently or covertly gaining unauthorized access to the session state.

Replication Servers

It is not uncommon for an enterprise application to use a cluster of replication servers where each server has the full complement of components. Whenever a request is received from a client, the request is routed to the next available server within the cluster. In this way, the infrastructure can maintain acceptable performance even if hundreds of requests are received simultaneously.

However, this can easily result in a problem if session state is maintained on the server side. Which server has the session state for the client? There are two strategies that are used to manage this problem. These are to replicate session state across all servers within the cluster or to route a client to the same server for the duration of the session.

However, keep in mind that clustering J2EE servers is vendor specific and is not part of the J2EE specifications. In addition, vendors typically replicate session on a primary and secondary server rather than on all the servers.

Best Practice—Maintaining a Sticky User Experience *The best practice is to maintain a sticky user experience, depending on your business needs. This means the client always uses the same server during the session and the session state is stored on one server within the cluster. This also means that the session is lost should the server go down.*

Valid Session State

Another issue that is common with an enterprise application that stores session state on a server is whether or not the session state is valid. Session state automatically becomes invalid and removed when the session ends. However, there might be occasions when the session ungracefully terminates without removing the session state, such as during a communication failure that occurs during the session.

Best Practice—Setting a Session Timeout *The best practice is to always set a session timeout, which automatically invalidates session state after time has passed and the session state has not been accessed. The actual length of time before the session automatically terminates will vary depending on the nature of the application. However, once time has expired, the session ends—and therefore the session state is removed.*

Web Tier and JavaServer Pages

The web tier contains components that directly communicate with clients. The Web Tier is also the location where JavaServer Pages (JSP) programs reside. JSPs are commonly proxy to an application server as implemented by Tomcat and Weblogic. Weblogic uses web server plug-ins in their implementation.

This is a particularly good technique to use when vertically scaling your application because it provides additional security. All executable processing resides below the web server and out of reach of an attack, which is usually focused on web servers. A JSP program is a component that provides service to a client. The nature of the service depends on the design of the application.

A JSP program is identified with a URL that is associated with a hyperlink built into a web page displayed on the client. When a user selects the hyperlink, the browser calls the JSP program, which executes JSP statements.

For example, a JSP program might receive order information (field names and field values) that a browser extracted from an HTML order form and passed to the JSP program. The JSP program uses information received from the browser to process the order by calling one or more Enterprise JavaBeans. Once processing is completed, the JSP program generates a dynamic web page that serves as a confirmation of the order.

Presentation and Processing

A JSP program can contain two components: the presentation component and the processing logic. The presentation component defines the content that is displayed by the client. Processing logic defines the business rules that are applied whenever the client calls the JSP program.

Although placing both the presentation and processing logic components in the same component seems to compartmentalize enterprise application, this technique can lead to nonmaintainable code. This is because of two reasons.

First, presentation and processing logic components tend to become complex and difficult to comprehend, especially when the code consists of a mixture of HTML code and Java scriptlets. That is, there is too much information in the program for the programmer to digest.

The other reason is that programmers with different skill sets typically write each component. A programmer who is proficient in HTML writes the presentation component. A Java programmer writes the processing logic component. This means that two programmers must work on the same JSP program, which can be inefficient.

Best Practice—Separating Code *The best practice for writing a JSP program is to separate the presentation code and the processing code (see Figure 3-4). Place the presentation code in the JSP program and place the processing code in Enterprise JavaBeans. Have the JSP program call the Enterprise JavaBeans whenever the JSP program is required to process information. An alternative best practice is to simply include files that contain code to hide the code from the graphic artist.*

There are several benefits to using this strategy. First, the JSP program is easier for a programmer to comprehend since the processing logic no longer resides in the JSP program. That is, there are fewer lines of code for the programmer to digest, making the program maintainable.

The task is also divided along discipline lines. The HTML programmer can focus on creating the JSP program while the Java programmer builds the Enterprise JavaBeans, and both tasks can occur in parallel.

Another benefit is that the processing logic is sharable with other enterprise applications. This is because the Enterprise JavaBean that contains the processing logic can be called by other applications.

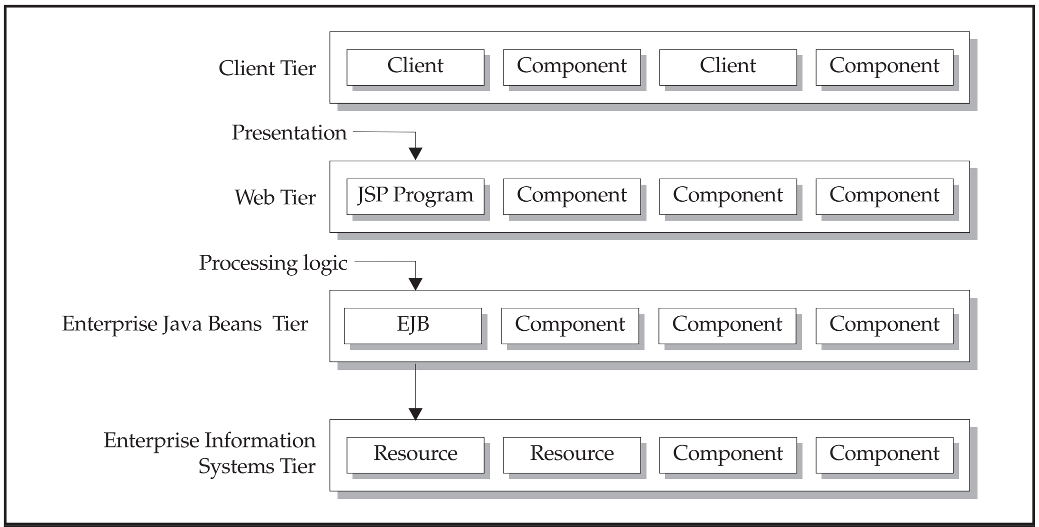

Figure 3-4. *A JSP program is used to create the presentation for an enterprise application while Enterprise JavaBeans are used to process information.*

The Inclusion Strategy

The designer of an enterprise application typically uses the same elements for all web pages of the user interface to provide continuity throughout the application. This means that the same code can appear in more than one web page, which is inefficient and a maintenance nightmare.

A further complication arises when multiple JSP programs generate web pages, because the developer must identify and change every JSP program that generates a web page each time a change is made to an element that is common to all pages.

Best Practice—Using the Inclusion Strategy *The best practice to avoid redundant code is to use the inclusion strategy, which uses either the include directive or include action to insert commonly used code into a JSP program. The JSP program include directive is used within the JSP program to specify the file whose content must be included in the JSP program. This file should contain shared code such as HTML statements that define common elements of web pages. The include action calls a JSP program within another JSP program. Commonly used code is contained in the called JSP program.*

The critical difference between the include action and the include directive is that the include action places the results generated by the called JSP program into the calling JSP program. In contrast, the include directive places source code into the JSP program rather than the results of running that source code.

Best Practice—Using the Include Directive *The best practice is to use the include directive whenever variables are used in the JSP program. The include directive places the variable name in the calling JSP program and lets the calling JSP program resolve the variable. In contrast, the include action places the value of the variable in the calling JSP program.*

Style Sheets

It is important that an enterprise application's user interface is consistent throughout the application. Consistency helps the user become familiar with how to use the application. This means that the developer must write JSP programs that generate web pages having the same general appearance (i.e., same font style, font size, and color combination).

Consistency therefore requires that each web page contain redundant code that defines the web page style. And, as mentioned in the previous section, redundant code leads to maintenance headaches.

Best Practice—Enforcing Continuity with CSS *The best practice to enforce continuity among web pages that comprise an application's user interface is to use Cascading Style Sheet (CSS). CSS is a file that describes the style of elements that appear on the web page.*

The developer tells the client (i.e., browser) to reference the CSS file by using a CSS directive in the web page. The browser then automatically applies the style defined in the CSS file to the web page that contains the CSS directive. The programmer can easily change the style of the user interface by modifying the CSS file. Those changes are automatically implemented as the client displays each web page.

Besides making the user interface maintainable, CSS also reduces the amount of storage space required for web pages. This is because code that is used to define the style is placed in a shared file rather than being replicated in all web pages that use the style.

Simplify Error Handling

An enterprise application should trap errors that can occur at runtime using techniques that are available in Java (see *Java 2: The Complete Reference*). However, it is important that the application translate raw error messages into text that is understood by the user of the application. Otherwise, the error message is likely to confuse and frustrate the user.

Although errors can occur throughout the application, a client should present to the user errors generated by a JSP program or by a servlet. This is true even if an Enterprise JavaBean called by the JSP program catches the error. Once the error is trapped, the Enterprise JavaBeans forwards the error message to the JSP program. The JSP program translates the error message into a message easily understood by a user and then displays the translated error message in a dynamically generated a web page.

Best Practice—Handling Errors *A best practice for handling errors is to generate a user-friendly error message that reflects the processing that was executing when the error occurred, session state (where applicable), and the nature of the error.*

Let's say that a user submitted an order form. The browser called a JSP program and passed order form information to the program. The JSP program called an Enterprise JavaBean that connects to a database and processes the order.

Suppose the Enterprise JavaBeans is unable to connect to the database. An error is thrown and caught by the Enterprise JavaBeans who forwards the error message to the JSP program. The error message "unable to connect to the database" is meaningless to the person who submitted the order, so it doesn't make sense to display this error message.

However, the JSP program evaluates the session state, which indicates the web page displayed an order form and that the order was being processed at the time the error occurred. The JSP program knows from the error message received from the Enterprise JavaB eans that connection to the order database failed. This means the JSP program could format an error message that says, "We experienced technical difficulties when processing your order. Please wait ten minutes and resubmit your order."

Best Practice—Saving Error Messages *Another best practice when handling errors is to have either the JSP program or the Enterprise JavaBeans save all error messages and related information (i.e. session state) to an error file, then notify technical support that an error was detected.*

Enterprise JavaBeans Tier

The Enterprise JavaBeans Tier contains Enterprise JavaBeans that provide processing logic to other tiers. Processing logic includes all code and data that is necessary to implement one or more business rules.

As discussed in the previous chapter, the purpose of creating an Enterprise JavaBean is to encapsulate code that performs one task very well and to make that code available to any application that needs that functionality.

Although the concept of using Enterprise JavaBeans is easily understood, there can be confusion when designing Enterprise JavaBeans into an application's specification. Simply stated, the developer must determine what functionality should be built into an Enterprise JavaBean.

Let's say that an online order entry application must determine if a customer is in good standing before the customer order is processed. The business logic requires the application to verify the customer's status is "good" before placing the order in the orders table. This means two database tasks must be performed. Should each task be in its own Enterprise JavaBeans? If so, then how are both Enterprise JavaBeans called? Should both tasks be included in one Enterprise JavaBean?

These questions reoccur many times during the development of an enterprise application, and answers to these questions can have either a negative or positive effect on the performance of the application.

Best Practice—Making JavaBeans Self-Contained *The best practice is make each Enterprise JavaBeans self-contained and minimize the interdependence of Enterprise JavaBeans where possible. That is, avoid having a trail of Enterprise JavaBeans calling each other. Instead, design an individual Enterprise JavaBean to complete a specific task.*

In the previous example, a good design is to use three Enterprise JavaBeans. One verifies the customer's status. Another processes the order. And the remaining Enterprise JavaBeans takes on the role of a controller.

A controller is an Enterprise JavaBeans that is called by the JSP program. The controller calls the Enterprise JavaBeans that verifies the customer's status. Based on information returned to the controller by this Enterprise JavaBeans, the controller either calls the second Enterprise JavaBeans to process the order or returns to the JSP program an order rejection notice, which the JSP program sends to the client.

Entity to Enterprise JavaBeans Relationship

There is a tendency for developers to create a one-to-one relationship between entities defined in an application's entity relationship diagram and with Enterprise JavaBeans. That is, each entity has its own entity Enterprise JavaBeans that contains all the processing logic required by the entity.

While the one-to-relationship seems a logical implementation of the entity relationship diagram, there are drawbacks that might affect the efficient running of the application. Each time an Enterprise JavaBean is created, there is increased overhead for the network and for the Enterprise JavaBeans container. Creating a one-to-one relationship, as such, tends to generate many Enterprise JavaBeans and therefore is likely to increase overhead, which results in a performance impact. In addition, this also limits the scalability of the application.

Best Practice—Translating Entity Relationship Diagrams *The best practice when translating an entity relationship diagram into Enterprise JavaBeans is to consolidate related processes that are associated with several entities into one session Enterprise JavaBeans. This results in the creation of fewer Enterprise JavaBeans while still maintaining the functionality required by the application.*

Efficient Data Exchange

A JSP program and Enterprise JavaBeans frequently exchange information while the enterprise application is executing. The JSP program might pass information received from the client to the Enterprise JavaBeans. Likewise, the Enterprise JavaBeans might return values to the JSP program once processing is completed.

There are two common ways in which information is exchanged between a JSP program and Enterprise JavaBeans. These are by individually sending and receiving each data element or by using a value object to transfer data in bulk.

Some developers intuitively use a value object only when bulk data needs to be transferred and send data individually when single values are exchanged. While this seems logical, there is a serious performance penalty—especially with enterprise applications that have many simultaneous clients. Transmitting individual data increases the network overhead.

Best Practice—Exchanging Information Between JSP and Enterprise *The best practice when exchanging information between a JSP program and Enterprise JavaBeans is to use a value object. In this way, there is less stress on the network than sending individual data and the value objects retain the association among data elements.*

Enterprise JavaBeans Performance

While Enterprise JavaBeans provide an efficient way to process business rules in a distributed system, Enterprise JavaBeans remains vulnerable to bottlenecks that occur when Enterprise JavaBeans communicate with other components. Bottlenecks effectively decrease the efficiency of implementing Enterprise JavaBeans.

The primary cause of bottlenecks is remote communication—that is, communication that occurs between components over the network. This is sometimes referred to as Enterprise JavaBeans chatter. As mentioned previously in this chapter, by its nature Enterprise JavaBeans uses appreciable amounts of resources to communicate with remote components.

Therefore, the more communication that occurs between an Enterprise JavaBeans and other components, the higher the likelihood that bottlenecks will occur. This is especially prevalent when clients directly access Enterprise JavaBeans, although the Enterprise JavaBeans specification prohibits such direct interaction.

Best Practice—Placing Components in Communication *The best practice is to keep remote communication to the minimum needed to exchange information and to minimize the duration and any communication. A common way to accomplish this objective is by placing components that frequently communicate with each other on the same server, where possible.*

Consider Purchasing Enterprise JavaBeans

There are entities and workflow common to many businesses. Information and processes used for both of these are encapsulated into entities Enterprise JavaBeans and session Enterprise JavaBeans, respectively. An entity Enterprise JavaBeans is modeled after an entity in the enterprise application's entity relationship diagram. A session Enterprise JavaBeans is modeled after processes common to multiple entities.

Many corporations have entities that use the same or very similar functionality. For example, many corporations use the same credit card approval process. Therefore, the same basic Enterprise JavaBeans is re-created in each corporation. This means corporations waste dollars by building something that is already available in the marketplace.

Best Practice—Surveying the Marketplace *The best practice is to survey the marketplace for third-party Enterprise JavaBeans that meet some or all of the functionality that is required by an entity Enterprise JavaBeans or session Enterprise JavaBeans for an enterprise application. The Sun Microsystems, Inc. web site offers third-party Enterprise JavaBeans in their Solutions Marketplace.*

The Model-View-Controller (MVC)

Developing an enterprise application is a complex undertaking because the application must be capable of serving many diverse clients simultaneously over a distributed infrastructure. Furthermore, the application must be scalable so the application can continue to provide acceptable performance regardless of the increase in the number of clients who use the application.

Throughout this section you learned the best practices for making an application scalable and maintainable by incorporating processing logic into Enterprise JavaBeans. However, deciding which features of an application should be built into an Enterprise JavaBeans can be confusing.

Best Practice—Using the Model-View-Controller *The best practice for simplifying the distribution of an application's functionality is to use the Model-View-Controller (MVC) strategy that is endorsed by Sun Microsystems, Inc. and which has its roots in the decades-old technology of Smalltalk.*

The MVC strategy basically divides applications into three broad components (see Figure 3-5). These are the model class, the view class, and the controller class. The model class consists of components that control data used by the application. The view class is composed of components that present data to the client. And the controller class is responsible for event handling and coordinating activities between the model class and the view class.

Enterprise JavaBeans are used to build components of the model class. Likewise, JSP programs and servlets are used to create view class components. And session Enterprise JavaBeans are used for controller class components.

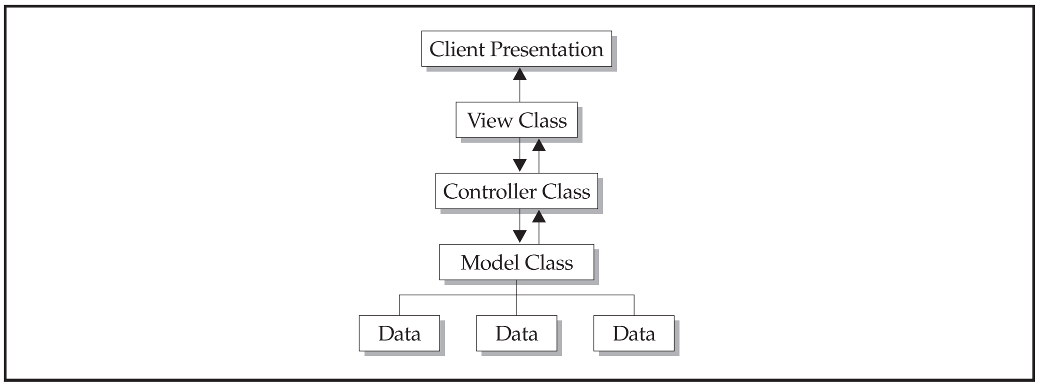

Figure 3-5. *The MVC strategy divides an application into a model class, a view class, and a controller class, which coordinates activities between the two other classes.*

The Myth of Using Inheritance

Inheritance is a cornerstone of the Java programming language, which itself is a feature inherited from the C++ programming language. There are three reasons for using inheritance in an enterprise application. First, inheritance enables functionality and data to be reused without having to rewrite the functionality and data several times in an application. Inheritance is also used to embellish both a functionality and data. That is, the class that inherits a functionality can modify the functionality without affecting the original functionality. Inheritance also provides a common interface based on functionality that is used by similar classes.

As you remember from when you learned of the Java programming language (see *Java 2: The Complete Reference*), there are two kinds of classes used in inheritance: the base class and the derived class. The base class contains methods and data some or all of which are inherited by a derived class. An object of the derived class has access to some or all of the data and methods of the base class and all the data and methods of the derived class.

A motor vehicle is a good example of inheritance. A motor vehicle is a base class and has an engine, drivetrain, wheels, and body, among other components. An automobile and a truck are two kinds of motor vehicles. They are derived classes and inherit an engine, drivetrain, wheels, body, and other components from the motor vehicle class.

However, the automobile class embellishes the engine, the drivetrain, wheels, and body to meet the needs of an automobile. Likewise, a truck class embellishes the same components to meet the needs of a truck.

The relationship between a base class and derived class is referred to as *coupling*. That is, there is a derived class bonded to a base class. A base class must exist for the derived class to exist. Both the automobile class and the truck class are coupled to the motor vehicle class.

Developers of enterprise applications are concerned with how to efficiently translate an entity relationship diagram into an application's class model. A common error is for the developer to rely heavily on inheritance, which results in an application built on coupled classes.

Best Practice—Using an Interface *The best practice when translating an entity relationship diagram to an application's class model is to use an interface rather than use coupling, where possible. An interface is a class that adds functionality to a real-world object.*

Interfaces and Inheritance

An interface contains functionality that is used across unlike real-world objects. This is different from a base class in that a base class provides functionality that is fundamental to like real-world objects.

For example, an acceleration interface provides acceleration functionality to any real-world object regardless if the real-world object is a motor vehicle, aircraft, or a baseball that is hit into the outfield. In this way, unrelated real-world objects that have the same functionality can share the same data and methods of an interface.

A common mistake is to use an interface as a base class because intuitively this seems sensible, but an interface is too narrow in scope which means an interface consists of one of many functionalities that are used by real-world objects.

Best Practice—Identifying Functionality *The best practice is to identify functionality that is common among real-world objects that are used by an application. Place these features into a base class. The remaining features that are unique to each of these objects are placed into a derived class. And where there is a function that is common to unlike real-world objects, place that function into an interface class.*

An interface (see "The Power of Interfaces" section) contains method signatures without specifying how those methods are implemented. This means an interface definition specifies the name of a method and the data type and position of the method's parameter list. The class that uses the interface defines code within the method. In this way, the application hides the implementation of the interface, which is an example of polymorphism.

Composition and Inheritance

A misnomer when designing an application that uses an interface is that designers cannot reuse the implementation of the interface—that is, the method that contains the interface signature can only be used with one object. This is untrue.

Best Practice—Creating a Delegate Class *The best practice is to create a delegate class. A delegate class contains implementations of interfaces that are commonly used by objects in an application. Let's say that the acceleration interface discussed previously in this section uses a standard formula to calculate the acceleration of an object. Therefore, the implementation of the acceleration interface is the same across unlike objects. A delegate class should be created that defines this implementation so the implementation can be used by other objects by calling the acceleration method of the delegate class.*

Using a delegate class is a keystone to the composition strategy, which is a way of extending the responsibilities of an object without inheritance. The composition strategy delegates responsibility to another object without the object being a derived class. This is a subtle but critical factor that differentiates composition from inheritance.

Inheritance extends the responsibilities of a base class by enhancing attributes and functionality of its derived class. Let's say the motor vehicle class should contain a method of installing a child seat in the vehicle.

Not all motor vehicles require this functionality, which means it doesn't make sense to include the functionality in the base class if the functionality isn't common to all motor vehicles. However, including the functionality in the automobile derived class can expand the functionality of the motor vehicle.

Best Practice—Using Composition *The best practice is to use composition whenever functionality needs to expand the responsibilities of unlike objects. Instead of incorporating the function in each object or in each of the object's base classes, the developer should delegate the responsibility to a delegate class that performs the functionality as required by the object.*

Potential Problems with Inheritance

While inheritance fosters strong coupling between similar objects and provides a mechanism for objects to share attributes and functionality, there are inherent problems that might occur that could have a negative impact on an enterprise application. One of these problems is called the *ripple effect*.

The ripple effect occurs whenever a change is made to the base class. Changes to a base class ripple down to all the derived classes and might negatively impact the implementation of attributes and functionality of the derived class.

This means that the developer who is responsible for maintaining a base class must examine the impact any change in the base class has on derived classes before making the change. Failure to assess the potential impact of a change could lead to errors in the derived class.

Another problem with inheritance happens as an application matures and requires frequent changes to both base and derived classes. These changes result in object transmute where data and methods of an existing class are moved to another class after the existing class is deleted.

This can lead to a number of issues such as lost data and methods and copying data and methods that are no longer necessary. And to further complicate the transition, the developer needs to assess the impact the object has on the derived class if the base class is replaced.

Best Practice—Minimizing the Use of Inheritance *The best practice is to minimize the use of inheritance in an enterprise application. Only use inheritance when there is commonality among objects that is not functionality. Otherwise, use composition. That is, an object that is a "type of" should inherit commonality from a base class. An object that performs functionality "like" other classes should use composition to delegate responsibility.*

You should use the refactoring strategy whenever changes are made to a base class. The refactoring strategy requires that the developer divide changes into small modifications and then make each small modification, followed by thorough testing. At the successful conclusion of the test, make the next small modification. Repeat these steps until all the modifications are completed.

Maintainable Classes

Classes are a focal point of every enterprise application since classes form the nucleus of many components in the application. It is therefore critical that an application is developed so that its many classes are easily maintained; otherwise, any modification of the application might be difficult to accomplish and introduce processing errors.

There are two factors that determine if classes are maintainable. These are coupling and cohesion. Coupling occurs when there is a class dependent on another class. This is illustrated in the previous section where a derived class is dependent on a base class. Coupling also occurs when a class delegates responsibility to another class.

Before a change can be made to either the derived class or the base class, the developer must assess the impact on the coupled class. As previously discussed, changes to a base class might negatively impact a derived class. Changes to a derived class won't affect the base class, but could inadvertently modify functionality inherited from the base class. In either scenario, additional precautions must be taken by the developer to assure that the modification doesn't cause a negative impact.

Cohesion describes how well a class' functionality is focused. That is, a class that has broad functionality isn't as cohesive as a class that has a single functionality. For example, a class that validates account status and processes orders is less cohesive than a class that simply validates an account status.

Another design consideration is to avoid packages with cross dependencies. If package A has a class that depends on a class in package B, then the developer needs to ensure he or she doesn't introduce a class in B that depends on a class in A.

Best Practice—Designing an Enterprise Application *The best practice is to design an enterprise application with highly cohesive classes and minimum coupling. This strategy ensures that classes are optimally designed for maintenance. This is because a class that has few functions and isn't dependent on another class is less complicated to modify than a class with broad functionality that inherits from a base class.*

In reality, an enterprise application is built as a collaborative effort and by its nature must have a blend of cohesion and coupling. Therefore, let the application requirements dictate the mixture of cohesiveness and coupling that is used in the design of an enterprise application.

Performance Enhancements

Some developers have concerns over the performance of an enterprise Java application because of the nature in which the application is compiled into bytecode rather than native code. Although bytecode is optimized, as you learned in the previous chapter, bytecode still needs to be interpreted by the Java Virtual Machine (JVM). It is this overhead that detracts from the application's performance.

There are two ways to reduce or practically eliminate the amount of bytecode that is interpreted at runtime: by using HotSpot, new in the Just In Time compiler from Sun Microsystems, Inc., or by using a native compiler.

HotSpot tunes the performance of an application at runtime so the application runs at optimal performance. HotSpot analyzes both client- and server-side programs and applies an optimization algorithm that has the greatest performance impact for the application. An application that uses HotSpot typically has a quick startup.

The drawback of using HotSpot is that machine time and other resources are used to analyze and optimize bytecode while the application runs, rather than simply dedicating these resources purely to running the application.

Another concern about using HotSpot is the complexity of debugging runtime errors. Runtime errors occur in the optimized code and not necessarily in the bytecode. Therefore, it can be difficult to re-create the error.

Best Practice—Testing the Code *The best practice for achieving top performance is to test both the native compiled code with the bytecode of the program. Code compiled with a native compiler such as that offered by Tower Technology optimizes translated bytecode into an executable similar to how source code written in C++ and other programming languages is compiled into an executable.*

However, the executable code is larger than the original bytecode because the compiler reduces the number of memory references, assuming you optimized it for speed. Executable code provides an economical alternative to the acquisition of hardware to boost performance of a Java enterprise application only if the overhead of the executable doesn't cause its own performance bottleneck.

It goes without saying that the major drawback of compiling a Java enterprise application is the application becomes machine dependent and therefore lacks the flexibility that is inherent in noncompiled Java applications.

A new hybrid strategy is developing to increase performance of a Java application where an application is divided into static and dynamic modules. Static modules such as an Enterprise JavaBeans container are compiled into native libraries that are executables, and dynamic modules such as Enterprise JavaBeans are compiled into bytecode. Only dynamic modules are optimized at runtime.

The Power of Interfaces

Designing an enterprise application using interfaces provides built-in flexibility, known as *pluggability*, because an interface enables the developer to easily replace components. As mentioned previously in this chapter, an interface is a collection of method signatures. A method signature consists of the name and the number and type of parameters for a method.

A developer implements the interface by defining a method of a class that has the same method signature as the interface. The developer must also define the method by itself. This means that the developer controls the behavior of the method whose method signature is defined in the interface.

Let's say a developer is building an enterprise application that processes orders. However, there are two different processes. One processes for bulk sales and another for retail sales. Two classes are defined, called retailOrder and bulkOrder.

Also, an interface is created to make uniform the way in which components send orders. We'll keep the interface simple for this example by requiring an order to have an account number, order number, product number, and quantity. The interface defines a method signature used to send an order. The method signature is shown in Listing 3-5.

Listing 3-5 Here is an example of a method signature.

```
sendOrder(int, int, int, int);
```

The developer must define a method in the retailOrder and bulkOrder classes that have the same signature as the sendOrder method signature. Likewise, the developer must place code within these methods to receive and process the order.

The programmer who wants to write a routine that sends an order needs only to know the class name to use and that the class implements the interface. The programmer already knows the method to call to send the order because the method signature is defined in the interface.

If the programmer wants to send a retail order, the programmer creates an instance of the retailOrder class and then calls the sendOrder() method which passes it the order information. A similar process is followed to send a bulkOrder, except the programmer creates an instance of the bulkOrder class. The call is made to the same method.

Simply said, an interface defines a standard way to interact with classes that implement the interface. This enables a developer to replace a class with another class that implements the same interface, and only the statement that creates the instance of the class needs to be changed in the program that calls the method. The routine calls the same method and passes the method the same argument.

Best Practice—Handling Differing Behaviors *The best practice is to create an interface whenever an application contains common behaviors where algorithms used to process the behaviors differ. Basically, the developer calls the method using the interface and passes the method information it needs and the method does everything else.*

Likewise, the developer whose method is called isn't concerned about the routine that called the method. Instead, the developer is only concerned about receiving and processing information.

The Power of Threads

Proper use of threads can increase the efficiency of running an enterprise application because multiple operations can appear to perform simultaneously. A thread is a stream of executing program statements. More than one thread can be executed within an enterprise application. This means that multiple statements can run parallel.

The thread class (see *Java 2: The Complete Reference*) is used to create a thread. Once the thread is created, the developer can specify the behavior of the thread (i.e., start, stop) and the point within the program where the thread begins execution.

A major benefit of using threads in an enterprise application is to be able to share processing time between multiple processes. Only one thread is processed at a time, although using multiple threads in an application gives the appearance of concurrent processing. Actually, the number of application threads being processed at one time is equal to or less than the number of CPUs on the machine.

The developer can assign a priority to each thread. A thread with a higher priority is processed before threads with lower priority. In this way, the developer is able to increase the response time of critical processes.

A simple example of using threads is when a client is printing a document while inputting data into the application. Typically, data entry has a higher priority than printing. Therefore, the data input thread priority is set as "high" and the printing thread is set as "low." Printing occurs while the application is waiting for the client to enter data. Once data is received from the client, the printing thread is temporarily suspended until the data input thread finishes processing—at which time the print thread resumes processing.

The use of threads in an enterprise application can be risky if methods executed by threads are not synchronized, because more than one thread might run within a method. Multiple threads that execute the same method share values, which can cause unexpected results.

However, synchronizing a method enables one thread at a time to run within the method. This practically locks the method and assures that values aren't shared. Other threads wanting to run within the method are queued until the executing thread finishes processing. However, the executing thread might be suspended if another thread executing elsewhere in the application has a higher priority than the executing thread.

Best Practice—Using Threads in an Enterprise Application *There are several best practices to employ when using threads in an enterprise application. Avoid using threads unless multiple processes require access to the same resources concurrently. This is because using multiple threads incurs processing overhead that can actually decrease response time.*

Keep the number of threads to a minimum; otherwise, you'll notice a gradual performance degradation. If your application experiences a decrease in performance, prioritize threads. Assign a higher priority to critical processes.

Another method to increase performance when using multiple threads in an application is to limit the size of methods that are synchronized. The objective is to minimize the amount of time that a synchronized method locks out other threads.

Careful analysis of a synchronized method might reveal that only a block of code within the method affects values that must not be accessed by another thread until the process is completed. Other code in the method could be executed without conflicting with other threads.

If this is the case, place the block of code that affects value into a separate synchronized method. Typically, this reduces the amount of code that is locked when the thread executes and therefore shortens the execution time of the method.

Be on the alert for possible deadlocks when using too many synchronized methods in an application. A deadlock can occur when a synchronized method calls other methods that are also synchronized. The call to the method is a thread that might be queued because another thread is executing in the called method. And the called method might also call another synchronized method, causing a similar situation to occur.

Safe processing must be the top concern when implementing threads in an application. Synchronizing methods is one way to assure a thread processes safely. Another way is to have the object that is threaded determine when to suspend and stop the thread, instead of having other objects control the process.

Threading issues typically occur on multi-CPU machines and rarely on single-CPU machines. Therefore, it is critical that applications that use threads be tested on a multi-CPU machine to be assured that all threading issues are addressed.

Best Practice—Suspending or Stopping Threaded Objects *The best practice is to design objects so that a request is made that a threaded object be either suspended or stopped, rather than directing the threaded object to suspend and stop. There is a subtle but important difference between a request and a directive. A directive causes an action to occur immediately. A request causes an action to occur at an appropriate time.*

The threaded object should have built-in logic that determines the point in the process when it is safe to suspend or stop the thread. This practice is safer than if another object issues the suspend command or stop command.

The Power of Notification

An enterprise application typically has many events that occur randomly. Each event might affect multiple objects within the application and therefore it is critical that a notification process be implemented within the application, so that changes experienced by an object can be transmitted to other objects that are affected by the change.

Let's say a developer built a stock-trading application that accepts real-time stock feeds from outside vendors. An object within the application is responsible for comparing an incoming stock price to the previous price for the same issue. If the incoming price deviates by 5 percent from the previous price, a notification of the change is made to another object that flashes the stock price on the display. This object also changes the color of the display to red, indicating a drop in price, or green, indicating an increase in price.

There are three ways in which objects are notified of changes: passive notification, timed notification, and active notification. Passive notification is the process whereby objects poll relative objects to determine the current state of the object. The current state is typically the value of one or more data members of the object.

Although passive notification is the easiest notification method to implement, this is also the notification method that has the highest processing overhead. This is because there could be many objects polling many other objects, and each poll consumes processing time.

An alternative to the passive notification method is timed notification. Timed notification suspends the thread that polls objects for a specific time period. Once time expires, the thread comes alive once more and polls relative objects to determine the current state of the object.

The drawback of both passive notification and timed notification is that each of these notification methods polls relative objects. This means that polling occurs even if there are no changes in status. This wastes processing time.

Best Practice—Using Active Notification *The best practice is to use active notification. Active notification requires the object whose status changes to notify other relative objects that a change occurred. Objects that are interested in the status of the object must first register with the object. This registration process basically identifies the objects that need to be notified when a change occurs.*

The original active notification method was called the observable-observer method, which is also known as the publisher-scriber model. The publisher is the object that changes and the subscriber is the object that is interested in learning about these changes.

There are also other variations on the same theme. One such variation is the observable-repeater method, which is nearly identical to the observable-observer method except the observer forwards changes received from the observable to objects that have registered with the observer.

Best Practice—Creating Different Threads *The best practice is to create different threads for each notification process and assign a priority to each thread based on the importance of the notification. Assign a high priority to those notifications that are critical to running the application, and assign a low priority to those less critical to the success of the application.*