10 J2EE Patterns

CERTIFICATION OBJECTIVES

- Intercepting Filter Pattern
- Front Controller Pattern
- Model View Controller Pattern
- Business Delegate Pattern
- Service Locator Pattern
- Transfer Object Pattern
- ✓ Two-Minute Drill
- Q&A Self Test

In this final chapter of the book, we move from the "micro" level of individual servlet and tag APIs to the "macro" level of designing entire systems. You leave the low-level world of servlet containers and the tag life cycle API, and embrace some altogether bigger notions on how a J2EE system should be constructed.

This comes about because the SCWCD exam includes questions on six design patterns. A design pattern is like an architectural blueprint for solving some common problem, and J2EE applications exhibit a whole range of common problems that have been solved hundreds of times before. So when you encounter your own J2EE design issues, you may not have to think for yourself, but instead adopt an off-the-shelf design pattern.

The questions in this area are much more like those you find in the first stage of the Sun Certified Enterprise Architect (SCEA) examination. At first sight, this may seem like a soft topic. How difficult can big design ideas be compared to knowing low-level APIs? But proceed with caution: This is an area that often lets SCWCD exam candidates down, and it accounts for nearly 10 percent of the questions. Design patterns are not fluffy, easy things—you can't afford to be woolly in your grasp on them. So with that stern warning in mind, read on.

Core J2EE Patterns

You have probably encountered design patterns before in your work as a Java developer. Even if you haven't read the legendary design patterns book, you will have used at least one pattern.[1] You don't believe me? You can't have got this far in your Java career without using the java.util.Iterator interface. This is a living embodiment of the iterator design pattern documented in the Design Patterns book. As you'll recall, java.util.Iterator keeps the specifics of what kind of collection you are dealing with at arm's length, while allowing you to traverse the contents of the collection. If you compare this with the pattern in the GoF book, you'll find that this description of java.util.Iterator matches the goals for the iterator pattern.

Patterns are not confined to the lowest levels of a computing language. Good design ideas can be found at any level—indeed, some of them work at several levels (from fiddly components to enormous building blocks). J2EE boasts its own set of design patterns, targeted at solving problems working with J2EE-size components,

[1]See *Design Patterns—Elements of Reusable Object-Oriented Software,* by Gamma, Helm, Johnson, and Vlissides [Addison Wesley]. The authors are often referred to as the Gang of Four—abbreviated GoF—and their magnum opus referred to as the GoF book.

so the building blocks involved might be servlets, JavaServer Pages, and Enterprise JavaBeans as well as other support classes. The motivation for discovering and documenting J2EE patterns is the same as for patterns elsewhere. Your problem needs a solution. Chances are that somebody else has hit your problem (or one very like it) before. If they've taken the trouble to document that solution, you can adopt it: either as a ready-made solution or as one that requires a little adaptation.

Elements of a Pattern

In this chapter I've tried to cover everything you need to know to do well with the SCWCD design patterns questions, but I do have only 60 or so pages at my disposal. If you want to broaden and deepen your design exploration, you should know that the "bible" for J2EE design patterns is a book called *Core J2EE Patterns: Best Practices and Design Strategies* (by Alur et al., published by Sun Microsystems). Five out of the six design patterns required for the SCWCD exam were first defined in this book. You'll find the content of the core patterns book on the Sun web site:

```
http://java.sun.com/blueprints/corej2eepatterns/Patterns/
```

Just one unfortunate thing: At least at the time of writing, the free resources on the web site are aimed at J2EE 1.3, whereas the SCWCD is pitched at J2EE 1.4. Fortunately, the patterns world moves a little more slowly than the rate of API evolution, so most of what the web site covers is absolutely germane to your SCWCD studies. However, if you want the absolute latest information, you'll need to purchase the second edition of the book (revised for J2EE 1.4).

The features of design patterns are described according to a set of headings that act as a template for each pattern. This set of headings is not standard between champions of design patterns. So what the GoF book calls "implementation," the Core J2EE Patterns book calls "strategies," and so forth. The Core J2EE Patterns book talks in terms of

- **Context**—where in the J2EE environment you are likely to encounter a need for the pattern
- **Problem**—an account of the design problem that a developer needs to solve
- **Forces**—motivations and justifications for adopting the pattern
- **Solution**—how the design works, both in structural terms (depicted with UML diagrams) and implementation terms (often backed up with example code)
- **Consequences**—what are the pros and cons when the pattern is applied

My “template” for describing the patterns will be simpler than this. Here is what I will describe:

- How each pattern works, with reference to a concrete example
- The benefits (and, sometimes, drawbacks) of using the pattern
- Particular scenarios when it might be appropriate to adopt the pattern

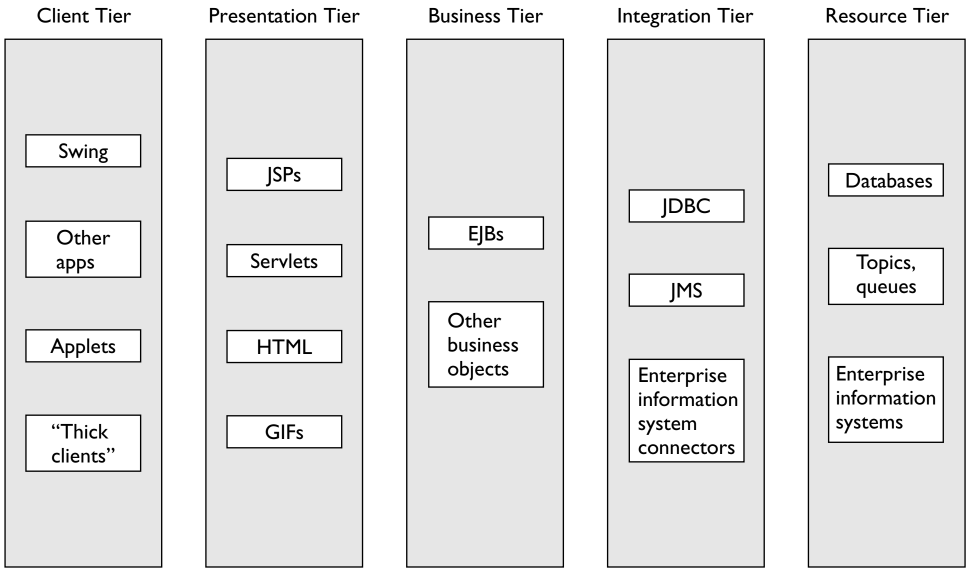
I’ve found this approach gets you to the heart of the exam objectives quicker than any other. Your task in the exam is not to be able to implement a design pattern (even though understanding the implementation clearly helps). You’ll see from the wording of the exam objectives that you have to match benefits and scenarios to specific patterns.

A Road Map for J2EE Patterns

First of all, Sun’s J2EE patterns are based on a tiered approach to application development. Of course, Sun wouldn’t claim to have invented this idea, but has sensibly adopted it as the best approach to separating out concerns and responsibilities: an aspect of object orientation in the large. The five tiers are depicted in Figure 10-1, together with the J2EE (and other) components you are likely to find in each.

FIGURE 10-1

Tiers for Application Development



You'll be familiar with many of the components depicted in the presentation tier. However, we haven't touched on components in the business tier: They haven't impinged at all on what you have needed to learn for the SCWCD—until now. The principal J2EE business component is the Enterprise JavaBean, or EJB. Fortunately, you don't need any great expertise in its use; that's the subject of a whole separate certification exam (the Sun Certified Business Component Developer Exam, or SCBCD). You do need a token idea, though, of what Enterprise JavaBeans are, and the issues they present.

To start off with, Enterprise JavaBeans are designed to be self-contained business components. That doesn't mean they constitute a single Java class: An Enterprise JavaBean is composed of a number of classes, some written by developers, some generated. However, a particular Enterprise JavaBean—maybe representing some business entity such as "Customer"—can be considered as a unit in its own right.

Just as servlets run in a servlet container, so EJBs run in an Enterprise JavaBean container. You can't make an EJB with the *new* keyword, any more than you'd consider doing the same for a servlet: The container supplies the life cycle. Whereas you fire an HTTP request to make use of a servlet in a servlet container, you execute JNDI (Java Naming and Directory Interface) code to get hold of the services an EJB can offer within its EJB container.

Originally, EJBs were designed to be independent, so independent that they did not even have to run in the same JVM as the client code using them. In the past, this meant that every use of an EJB involved a remote method call using Java's Remote Method Invocation (RMI) protocol. This doesn't have to be the case anymore when EJBs are co-located in the same JVM as the client using them. However, EJBs are to be treated with respect because they do a lot for you (not least of which is masking network concerns); they are heavyweight components. You might pay a price in performance for naïve use of EJBs.

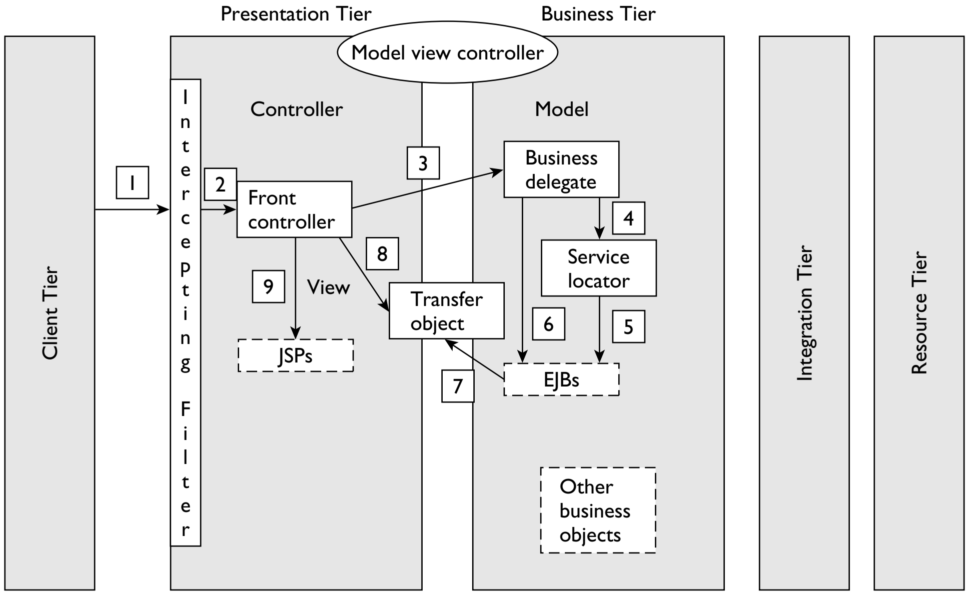
The preceding paragraphs made a very minimalist introduction to a very big topic, but it is important to understand that EJBs are a primary force in driving some of the patterns you encounter for the SCWCD (Business Delegate, Service Locator, and Transfer Object).

Having understood tiers and components, the next thing to consider is where the design patterns fit. Figure 10-2 shows the tiers again, but this time with the six design patterns superimposed. (You may also want to reference the diagram on the Sun web site that shows how all the J2EE core patterns fit together: That's at the Sun URL referenced a little earlier in the chapter.)

You can see from Figure 10-2 that the patterns you are concerned with live within or straddle different tiers, and often work in combination with one another. Here's a

FIGURE 10-2

How the J2EE Patterns Fit Together



brief explanation of each of the six patterns so that you have a road map for the rest of the chapter.

- Intercepting Filter: used to check or transform a request or a response.
- Front Controller: a "gateway" for all requests—can be used to check requests and control navigation centrally.
- Model View Controller: used to keep presentation and business concerns independent. The model holds the business data, the view presents the data, and the controller mediates between the two.
- Business Delegate: used to provide a friendly front end to business-level APIs (especially when those APIs involve Enterprise JavaBeans).
- Service Locator: used to locate services (!), which usually means executing highly technical code to get back references to resources such as Enterprise JavaBeans and JMS queues.
- Transfer Object: encapsulates the data returned from complex business objects (usually EJBs again).

There is a whole flow of control among these different patterns. Figure 10-2 shows how they might typically collaborate, tracing the flow of a request through the application. A request is trapped by an Intercepting Filter (1 in Figure 10-2)—if the filter allows it, the request passes on to a Front Controller (2). The Front Controller makes use of a Business Delegate to obtain business data (3). The Business Delegate uses a Service Locator (4) to find the business component it needs (5)— often an EJB. The Business Delegate requests data (6), which it gets back in the form of a Transfer Object (7), and returns this to the Front Controller. The Front Controller takes what it needs from the Transfer Object (8), then forwards to an appropriate JSP (9) (which—in all likelihood—uses data originating from the Transfer Object). Where does Model View Controller fit in? That's the collection of components represented by the JSP (for the view), the Front Controller (evidently the controller!), and the Business Delegate (fronting the business model).

A Working Example

This chapter is underpinned by a working example that uses all six patterns in its construction. There are no Enterprise JavaBeans within the example, for they require an Enterprise JavaBean container to support them (like the open source JBoss: See http://www.jboss.org). At this stage of your SCWCD exam preparation, coming to terms with an EJB container is a bridge too far. So instead, I have implemented the integration tier of the example using Java Remote Method Invocation (RMI). This introduces quite a few of the issues associated with EJBs but doesn't require additional software (beyond the J2SDK) to make it run.

Figure 10-3 shows how the application fits together and indicates which classes embody J2EE patterns. Some of the classes actually reflect the name of the pattern. There's a danger in this: You might think that one pattern = one class. Whereas that can be the case, it doesn't do justice to the full story. Even if only one class is involved, that class must cooperate in the right way with other classes and do a particular job to count as an implementation of a pattern.

The idea of the example application is to display information about the six patterns for SCWCD. The application has only two working screens: home.jspx and pattern.jspx. home.jspx displays a menu, from which you can select the pattern you want information about (Figure 10-4). On selecting a pattern, you see a detail screen with information about the pattern (Figure 10-5).

For such a simple application, the supporting architecture is very ornate, but the complexity is there to illustrate how to apply the six patterns. Let's consider how the application works, with reference to Figure 10-3. We'll start with the back end: the resource tier. The "database" of pattern information is represented by a simple properties file, called patternsDB.props. The PatternLoader class is arguably part of both the integration and business tiers. It contains a method to load the contents

FIGURE 10-3

How the Working Example Implements J2EE Patterns

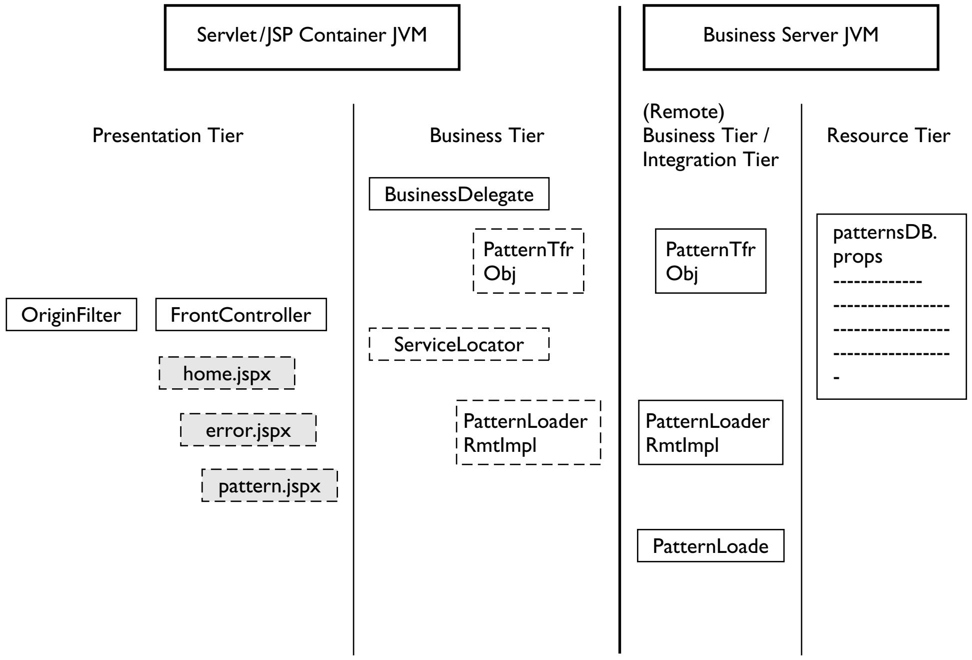
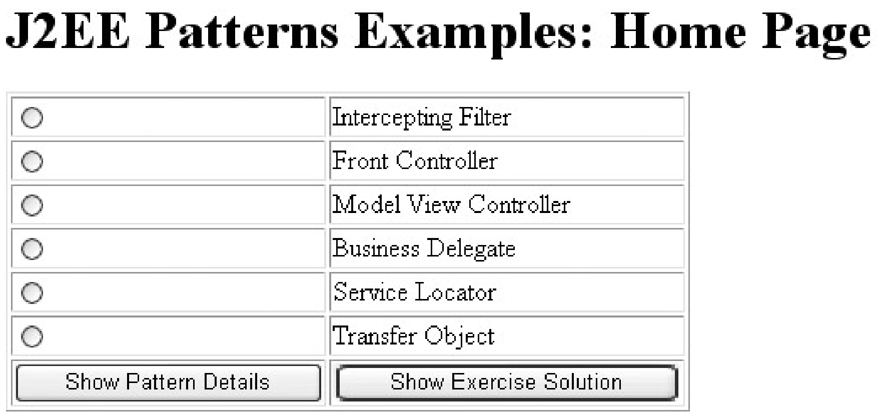
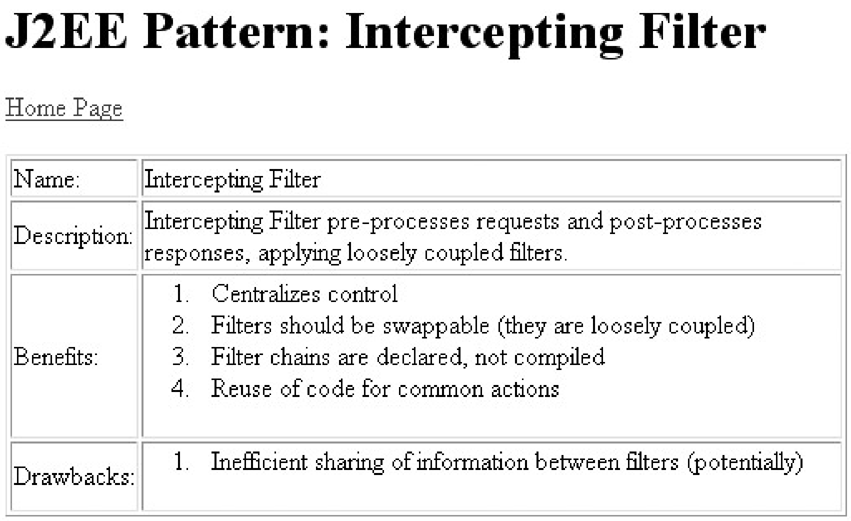


FIGURE 10-4

The Working Example Home Page



of the "database" into an internally held java.util.Properties object, using simple file input/output for connectivity to the data (in a real production system, you might expect to see JDBC code connecting to a relational database). PatternLoader also contains the first of our patterns: Transfer Object. It provides a "getData" method that returns a PatternTfrObj, which encapsulates all the data about one pattern. We'll see later why this is an advantage.



J2EE Pattern: Intercepting Filter

Home Page

Name:	Intercepting Filter
Description:	Intercepting Filter pre-processes requests and post-processes responses, applying loosely coupled filters.
Benefits:	1. Centralizes control 2. Filters should be swappable (they are loosely coupled) 3. Filter chains are declared, not compiled 4. Reuse of code for common actions
Drawbacks:	1. Inefficient sharing of information between filters (potentially)

FIGURE 10-5 The Working Example Detail Page

To make PatternLoader more EJB-like, it's wrapped up in a remote implementation class called PatternLoaderRmtImpl. The idea is that the web application will run within Tomcat and be forced to access this part of the business tier through remote calls. PatternLoader and its remote implementation run in an entirely separate JVM, launched with Java's `rmiregistry` command—we'll see more of that in the deployment instructions later.

Let's now come at the application from the other direction. First I make a request for the main screen of the application from my browser. The request is intercepted by the OriginFilter class. This is the key player in the Intercepting Filter J2EE pattern, and is a bona fide class implementing the Filter interface and declared as a filter in the deployment descriptor. Its purpose is to guard the borders of the application, turning away requests from any host of origin it doesn't like (in this case, it's set up to allow only requests from the localhost with IP address 127.0.0.1, but you can adjust this by changing the relevant initialization parameter within the web.xml filter declaration).

If the request gets past the border guard, it goes to a servlet called FrontController (named after the J2EE pattern it supports). Front Controller determines from parameters passed whether this is a request for a specific pattern or not. If not, it forwards to the home menu page, home.jspx. A selection from this page goes back to Front Controller—this time with a named pattern as parameter. This prompts FrontController to do a deal more work. It passed the pattern name requested to the BusinessDelegate class (our next J2EE pattern). BusinessDelegate checks to see if it has information about the pattern in its own data cache. If not, it sees if it has a reference to a remote PatternLoader object. If not, it calls on the ServiceLocator (next pattern) class to find the remote PatternLoader object. Once armed with a remote PatternLoader object, BusinessDelegate uses it to return a PatternTfrObj (the Transfer Object). Now the explanations have met in the middle. This is the point where our survey of the presentation and business tiers meets the earlier exploration of the resource, integration, and (remote) business tiers.

We'll revisit the application in detail when explaining the purpose of the different J2EE design patterns. It would be a good idea to deploy the application at this point. The WAR file is in the CD in /sourcode/ch10/lab10.war. Deploy this WAR file in the normal way; then access the home page with a URL such as the following:

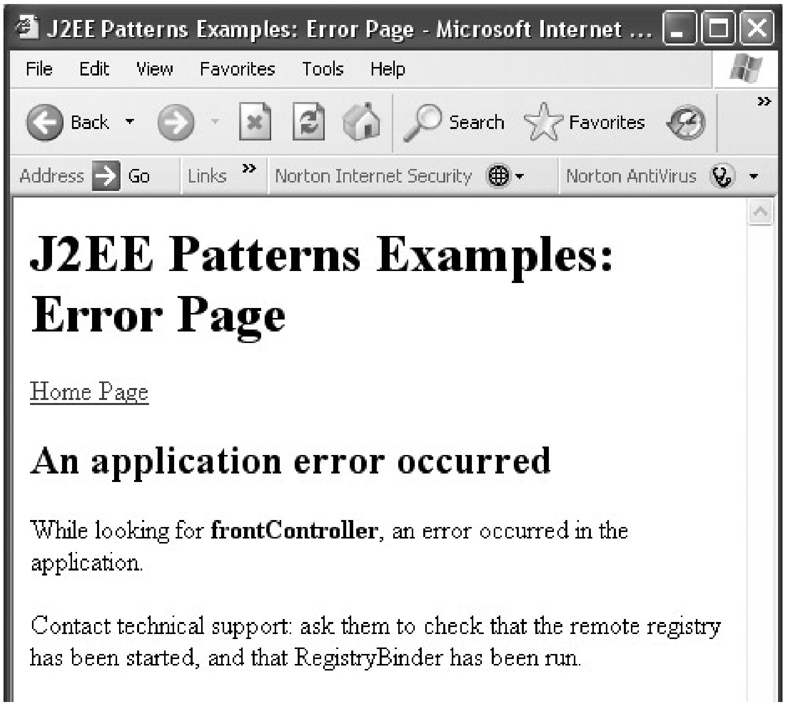
```
http://localhost:8080/lab10/controller
```

You should see the screen we saw earlier in Figure 10-4—the application home page. However, when you select one of the patterns and click the "Show Pattern Details" button, you'll see an application error screen (Figure 10-6). This is because the remote layers of the application haven't yet been started up. To bring these to life, take the following steps:

1. Start *two* command windows. In both command windows, change the current directory to /WEB-INF/classes within the deployed application directory (e.g., <Tomcat Installation Directory>/webapps/lab10/WEB-INF/classes).
2. In the first command window, execute the command `rmiregistry`. No parameters—just as printed. If the command is not recognized, make sure that you have the /bin directory of your J2SDK installation in your path (so if we're talking MS-DOS, adjust the PATH environment variable). This is the Java command that starts off your Remote Method Invocation Registry, which listens (by default) on port 1099. If successful, the command just hangs there, not giving much appearance of doing anything—don't worry about it!
3. In the second command window, execute the command `java webcert.ch10.lab10.RemoteBusinessServer`. The result should be a message saying "Pattern Loader bound in RMI Registry." Again, the command appears to hang in midair—leave it to do its stuff.
4. Now return to your browser, and try again to access a pattern through the main menu. This time, you should see a detail screen as in Figure 10-5.
5. When you have finished with the application, just abort the two command windows (CTRL-C and EXIT in MS-DOS terms).

FIGURE 10-6

The Working Example Error Page


J2EE Patterns Examples: Error Page - Microsoft Internet ...
File Edit View Favorites Tools Help
Back
Search
Favorites
Address Go
Links
Norton Internet Security
Norton AntiVirus

J2EE Patterns Examples: Error Page

Home Page

An application error occurred

While looking for **frontController**, an error occurred in the application.

Contact technical support: ask them to check that the remote registry has been started, and that RegistryBinder has been run.

With luck, RMI will be familiar to you from your previous programming experience (though it's not a mandatory part of the SCJP exam). Don't worry if not: It's certainly not central to the explanation of patterns—I've included it to give more of a sense of a "real" full-blown J2EE application. We'll revisit some of the technicalities when explaining the Transfer Object pattern at the end of the chapter.

CERTIFICATION OBJECTIVE

Intercepting Filter Pattern (Exam Objectives 11.1 and 11.2)

Given a scenario description with a list of issues, select a pattern that would solve the issues. The list of patterns you must know are: ***Intercepting Filter****, Model-View -Controller, Front Controller, Service Locator, Business Delegate, and Transfer Object.*

Match design patterns with statements describing potential benefits that accrue from the use of the pattern, for any of the following patterns: ***Intercepting Filter****, Model-View-Controller, Front Controller, Service Locator, Business Delegate, and Transfer Object.*

Let's start with some good news—you have learned all about the Intercepting Filter design pattern already. The kindly designers of J2EE have built the Intercepting Filter pattern directly into J2EE architecture. You learned all about Filters and FilterChains in Chapter 3, and how these could be used to "pre-process" a request for a web resource, or "post-process" the response. These interfaces and classes (together with the life cycle support for them in a servlet container) are the concrete embodiment of the Intercepting Filter pattern.

Intercepting Filter Pattern

The Intercepting Filter pattern works in the following way. A filter manager intercepts a request on its way to a target resource. From the content of the request (usually from its URL pattern), the filter manager calls on the help of a filter chain object, which creates the filter objects needful to the request (if they haven't been created already). The filter chain also dictates the correct sequence of filters to apply (hence the name: filter chain). The filter chain object infers the appropriate sequence from some information in the request: As we've seen with the "official" filter mechanism, the basis for this is the request's URL pattern.

So in Figure 10-7, we see that a request is made for the file /resources/resource1 .html. This request is grabbed by a FilterManager object, which builds a FilterChain object. The FilterChain object holds references to the matching filters, which the FilterManager determines by matching up the request URL to the URL mappings for each of the five available filters. From these, it selects one ofthe following choices:

FIGURE 10-7

A Request's Route Through Filters

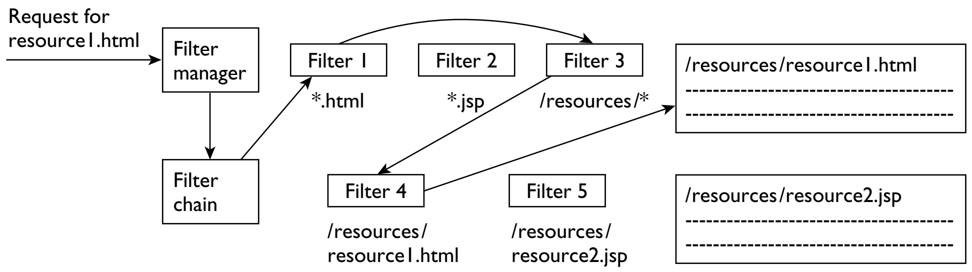
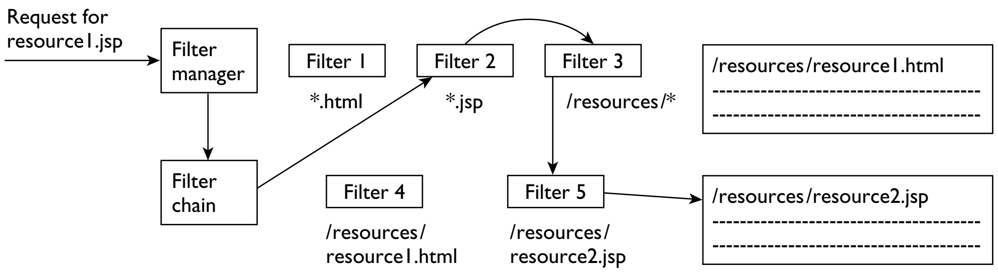


FIGURE 10-8

Another Request Routing Through Filters



- Filter 1: because of the extension match (*.html)
- Filter 3: because of the path match (/resources/*—anything whose path begins with /resources)
- Filter 4: because of the exact match (/resources/resource1.html)

See if you can work out the logic for the filter selection in Figure 10-8, when /resources/resource2.jsp is requested.

I am being very prescriptive here in saying that this J2EE pattern works by URL matching. If you write code for a customized Intercepting Filter, you might use some other criteria altogether for the selection of filters in the chain. It is just that in this case, the J2EE pattern has to be implemented in your application server because it's a mandatory part of the servlet specification—and the servlet specification goes with URL matching (and servlet naming) as the basis for filter selection.

How Is Intercepting Filter Used in the Example Application?

The example application uses a filter called OriginFilter. This is specified in the deployment descriptor as follows:

```
<filter>
  <filter-name>OriginFilter</filter-name>
  <filter-class>webcert.ch10.lab10.OriginFilter</filter-class>
  <init-param>
    <param-name>origin</param-name>
    <param-value>127.0.0.1</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>OriginFilter</filter-name>
  <servlet-name>FrontController</servlet-name>
</filter-mapping>
```

You can see that the filter has an initialization parameter called "origin" that holds an Internet address as a value—the one for the local host, 127.0.0.1. The implementing class has the following filter logic:

```
public void doFilter(ServletRequest request, ServletResponse
response,
          FilterChain chain) throws IOException,
ServletException {
  String origin = request.getRemoteHost();
  String mustMatch = config.getInitParameter("origin");
  if (origin.equals(mustMatch)) {
    chain.doFilter(request, response);
  } else {
    sendErrorPage(response, origin);
  }
}
```

The code performs a simple check, getting hold of the Internet address associated with the incoming request and comparing this with the initialization parameter setup on the filter in the deployment descriptor. If the two match, the filter chain is used to invoke the next item in the chain (`chain.doFilter()`), which is the FrontController servlet (there are only two things in the chain: the filter and the servlet).

Why Use Intercepting Filter?

What you see in the example code is a typical use of a filter: to perform an authentication check. In this case, the code is only allowing approaches from one particular server. When we first discussed filters in Chapter 3, you saw that you could harness them for a range of purposes—including but not limited to auditing, security, compressing, decompressing, and transforming request and response messages. I apply two tests when deciding whether some or other functional requirement should be placed in a filter:

1. Is the requirement discrete?
2. Is it common to more than one type of request (and/or response) from my web application?

For test 1, I define "discrete" as whether or not the filter can perform its work independently of other considerations. If other filters are placed before or after, will the current filter remain unaffected? For test 2, I may not want to apply my filter globally, but if it's executing logic that is specific to one particular resource, then perhaps the code belongs with that particular resource, not in a filter. Filters are meant to be more general purpose.

Possible reasons for using a filter are limited only by your imagination. Here are some scenarios where you might want to do some processing on the request before letting it loose on the targeted resource:

- You want to check something about the origin of the request. This might be the type of device making the request (laptop or iPod?).
- You might want to detect how the request is encoded, and maybe re-encode it before it reaches the core of your application.

These are "inbound" reasons for using a filter. What about "outbound" possibilities—things you might want to do to the completed response before returning it to the requester? You might

- Translate the response in some way—to make it more intelligible to another computer system or to a human.
- Tag your response in some general way—perhaps adding a copyright notice.

There's nothing stopping you from using a servlet to carry out any of the above tasks. The very worst case would have every servlet in your application embedding copied and pasted logic to perform common filtration actions (count yourself lucky if you have never witnessed this kind of bad practice!). A much better approach is to dedicate a servlet as a "gateway" to your application—this is like the Front Controller pattern we'll be considering next. This gateway (or controller) servlet could examine the request, perform whatever actions are required (using helper classes or other included servlets as necessary), then permit access to the actual requested resource. There are some drawbacks to this approach, however:

- You have to design the approach yourself. You might end up by reinventing the elegant wheels of the existing filter mechanism.
- It's easy to overburden your "gateway" servlet with too many responsibilities. At worst, your servlet will contain a burgeoning series of nested checks.
- Depending on your design, you'll need to change servlet code to add new filter logic or alter the sequence of the filter chain.

The official list of benefits (or at least, consequences) of using the Intercepting Filter pattern are as follows:

- Control is centralized for chosen activities (like encryption) but in a loosely coupled way (the encryption happens independently of anything else going on around it).
- Filters promote the reuse of code. It's easy to plug a filter in where you want it (as it presents a standard interface) so that there's no barrier to including it in your web application.
- Including a filter doesn't involve recompilation: Everything is controlled from the deployment descriptor, declaratively. You can juggle your filter order without recourse to javac (though dependent on your application server, you might have to remake and redeploy your WAR file).

The only real identified drawback for filters is this: Information sharing between filters is likely to be inefficient. You may well need to reprocess an entire request or entire response all over again in each filter in the chain. Imagine a multipart request—that's a POST with a load of file attachments. Suppose that these are composed of a random mixture of documents and spreadsheets and that you have one filter dedicated to automatic translation of the documents and another filter to verify the technical integrity of the formulae in the spreadsheets. Each filter would have to read through the entire request, with the document filter ignoring spreadsheet attachments and the spreadsheet filter needlessly churning through documents. However, to flip-flop between the two filters as each attachment type was recognized would constitute a violation of each filter's independence from the other.

EXERCISE 10-1

ON THE CD

Intercepting Filter Pattern

Consider how you would use the Intercepting Filter pattern to fulfill the following requirement: A company keeps information on the sales of goods from certain of its suppliers. These are accessible through the company's extranet and are currently produced through servlets whose response is delivered in plain-text, comma-delimited format. Some of their suppliers (readily identifiable through the host component of their requesting machines) are more sophisticated and would like the files delivering for a trial period ready-translated into an agreed XML format.

What are the benefits of the approach you use versus placing the same logic in a combination of servlets and JSPs?

If you run the working example that accompanies this chapter, select the Intercepting Filter pattern radio button option, and click the "Solution to Exercise" button, you'll find a discussion of the above scenario. The URL to launch the working example is likely to be

```
http://localhost:8080/lab10/controller
```

CERTIFICATION OBJECTIVE

Front Controller Pattern (Exam Objectives 11.1 and 11.2)

Given a scenario description with a list of issues, select a pattern that would solve the issues. The list of patterns you must know are: Intercepting Filter, Model-View-Controller, ***Front Controller****, Service Locator, Business Delegate, and Transfer Object.*

Match design patterns with statements describing potential benefits that accrue from the use of the pattern, for any of the following patterns: Intercepting Filter, Model-View -Controller, ***Front Controller****, Service Locator, Business Delegate, and Transfer Object.*

The essence of Front Controller is that it acts as a centralized point of access for requests. There are many responsibilities Front Controller may take on. As a gateway, it might control access through authentication and authorization rules. It may delegate to business processes. Almost invariably, it plays a crucial role in controlling (screen) flow through an application. In this next section, we'll consider how Front Controller works, the benefits it provides, and some scenarios for which it is well suited.

The Front Controller Pattern

Like Intercepting Filter, Front Controller is another presentation tier pattern. The main difficulty in answering questions about Front Controller is, in fact, distinguishing it from the Intercepting Filter pattern, for both have a lot in common. If you look back at Figure 10-1, though, you can see that Front Controller is embedded deeper within the presentation layer; Intercepting Filter is more at the fringes (as its name implies).

The pattern works by having an object (the Controller object) as the initial point of contact for requests to a web application (after those requests have made it past any en route filters!). The controller may validate the request according to any criteria it chooses. It may also extract information from the request that determines which page to navigate to next.

The prime example of a Front Controller that you are likely to encounter in your working life is the framework Struts. This uses what the Core J2EE Patterns book terms the "multiplexed resource mapping strategy." What this boils down to is this: The URL mapping *.do maps to a master servlet (acting as a Front Controller). Dependent on what is before the .do, the master servlet delegates to a class of type Action. So thisaction.do might map onto a class called ThisAction, and thataction.do to ThatAction.

How Is Front Controller Used in the Example Application?

Front Controller is best understood by example. In our application, the Front Controller pattern manifests itself in a single servlet called FrontController. This FrontController confines itself strictly to navigational issues. Here is the complete code for the servlet class (excluding package and import statements):

```
public class FrontController extends HttpServlet {
  public static final String[] patternNames = { "interceptingFilter",
    "frontController", "modelViewController", "businessDelegate",
    "serviceLocator", "transferObject" };
  private BusinessDelegate businessDelegate = new BusinessDelegate();
  protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {
    // Sort out link back to controller servlet
    String actionLink = request.getContextPath() + request.getServletPath();
    request.setAttribute("actionLink", actionLink);
    // Sort out the action required
    String patternName = request.getParameter("patternChoice");
    String page;
    if (isPatternNameRecognized(patternName)) {
      try {
        setPatternAttributes(patternName, request);
        page = "/pattern.jspx";
      } catch (PatternNotFoundException pnfe) {
        // Application error handling: masks remote error
        request.setAttribute("patternName", pnfe.getPatternName());
        page = "/error.jspx";
      }
    } else {
      page = "/home.jspx";
    }
    // Forward to required page
27     RequestDispatcher rd = getServletContext().getRequestDispatcher(page);
28     rd.forward(request, response);
29   }
30   protected boolean isPatternNameRecognized(String patternName) {
31     for (int i = 0; i < patternNames.length; i++) {
32       if (patternNames[i].equals(patternName)) {
33         return true;
34       }
35     }
36     return false;
37   }
38   protected void setPatternAttributes(String patternName,
39     ServletRequest request) throws PatternNotFoundException {
40     PatternTfrObj pto = businessDelegate.findPattern(patternName);
41     // translate value object properties to request attributes
42     request.setAttribute("patternName", pto.getName());
43     request.setAttribute("patternDescription", pto.getDescription());
44     request.setAttribute("patternBenefits", pto.getBenefits());
45     request.setAttribute("patternDrawbacks", pto.getDrawbacks());
46   }
47 }
```

The purpose of the code is to navigate to a page displaying the correct pattern information or, if no specific pattern is requested, to display the menu (home) page. Here's a breakdown of what the code does:

- In lines 02 to 04, a String array is declared as a constant. Called patternNames, it holds the names by which the six J2EE patterns in this chapter are known internally to the application.
- In line 05, FrontController declares an instance of a class called BusinessDelegate. We'll explore this class fully later in the chapter. It serves to front the business logic that FrontController needs—namely, getting hold of business objects that contain J2EE pattern information.
- The `doGet()` method begins at line 06. In lines 09 and 10, the FrontController servlet sorts out a URL to point back to itself. This is built dynamically from the context path and the servlet path, avoiding any literal hardcoding. Then the URL is made available as a request attribute (called `actionLink`) that can be accessed in the JSP "views"—avoiding JSPs having to hard-code any URL that may later change.
- In line 12, FrontController determines if there is a request parameter available called `patternChoice`, and stores the value as the local variable `patternName`.

- The remaining logic in the `doGet()` method is concerned with navigating to the right page. The major determining factor is whether of not the `patternName` passed is recognized or not. At line 14, the `doGet()` method calls `isPatternRecognized()` (lines 30 to 37), which takes the value of the `patternName` local variable, and compares this with the valid values for pattern names in the String array `patternNames`. Even if `patternName` is **null** (intentionally not passed), `isPatternRecognized()` won't fail, but will simply return **false**.
- If `patternName` is not recognized (or **null**), the name of the page to navigate to is set to the home page (line 24).
- If `patternName` is recognized, `doGet()` calls the `setPatternAttributes()` method (line 16). This method uses the BusinessDelegate object to return an object representing pattern data (line 40). The object returned is of type PatternTfrObj — a manifestation of the TransferObject pattern that we discuss at the end of the chapter. The `setPatternAttributes()` method then transfers the attributes of the PatternTfrObj object to a set of request attributes (lines 42 to 45).
- Back at line 17—after a successful call to `setPatternAttributes()`—the name of the next page to navigate to is set to be the pattern detail page (pattern.jspx).
- Line 18 catches a possible exception from `setPatternAttributes()`, the business error PatternNotFoundException. This doesn't originate from `setPatternAttributes()`—it's passed on from the BusinessDelegate object if an error occurs. We'll see later that the BusinessDelegate class masks any "technical" errors and translates them to this "application" exception.
- If a PatternNotFoundException is thrown, the `doGet()` method sets up an error page name to forward to (line 21).
- Finally in the `doGet()` method, at lines 27 and 28, a RequestDispatcher object is used to forward to whichever of the three pages has been determined by the foregoing logic.

The net result is that the three JSP documents in the application (home.jspx, pattern.jspx, and error.jspx) have nothing in the way of hard-coded navigation information. Here's a short extract from home.jspx:

```
<form action="${actionLink}">
  <table border="1">
    <tr>
```

```
        <td><input type="radio"
          name="patternChoice" value="interceptingFilter" /></td>
        <td>Intercepting Filter</td>
      </tr>
  ...
```

You can see that the target action for the HTML form is derived from an EL variable actionLink—which ties back to the request attribute set up in the controller so that all requests link back to the controller. From this short extract, you can also see that the parameter value for the pattern name comes from a value on a radio button.

Why Use Front Controller?

The reasons you might want to use Front Controller are numerous:

- *Central control of navigation.* Instead of allowing one JSP to directly link to another, you plant logical links in your JSPs instead—sending you via the Front Controller, which makes the ultimate decision about forwarding to the next JSP.
- *Central control of requests.* You might want this control so that you can easily track or log requests.
- *Better management of security.* With centralized access, you can cut off illegal requests in one place. Authentication and authorization can also be centralized (but only worry about this if declarative security, as discussed in Chapter 5, isn't enough for you. Better to have security embedded in your deployment descriptor than in code—even when that code is centralized in a Front Controller!).
- *To avoid embedding control code within lots of separate resources.* This is an approach that easily leads to a copy-and-paste mentality.

And the drawbacks? The Core J2EE Patterns book points out that the pattern can lead to a single point of failure. Given, though, that your Front Controller is likely to be implemented as a servlet, you can always mitigate this by distributing your application, as long as your application server supports this. And as servlets are by their nature multithreaded, it is not as if your Front Controller should be a bottleneck allowing only one request through at a time.

exam watch

If you are asked to match the pattern for a scenario that contains the words "managing the workflow," then the answer is very likely to be Front Controller.

on the job

It is very easy to overload a servlet acting as a Front Controller with too many responsibilities and too much code. The answer is to ensure that Front Controller, wherever necessary, delegates to appropriate helper classes. If you want to go further with this (beyond what the SCWCD exam requires), take a look at the View Helper pattern.

When Do You Choose a Front Controller Instead of an Intercepting Filter?

It's often hard to tell which pattern to use—Front Controller or Intercepting Filter. The Core J2EE Patterns book identifies these two patterns as complementary: "Both Intercepting Filter and Front Controller describe ways to centralize control of certain types of request processing, suggesting different approaches to this issue." You can't really argue with that statement, but it does leave you wondering which approach is suitable for which requirement. It's often a matter of experience and taste—which is less clear-cut than you would like in face of the SCWCD exam! The following Scenario and Solution table may help you to get a feel for which pattern is appropriate and when.

SCENARIO & SOLUTION	
Controlling the flow of navigation from one page to another	Front Controller (as we've seen)
Converting responses from one form of XML to another	Intercepting Filter (as long as there is a general way of translating the XML for many different responses)
Zipping up the response	Intercepting Filter
Executing different blocks of business logic dependent on a parameter in the request	Front Controller (dispatching to other classes that contain the business logic)
Stopping a request dead in its tracks if your application dislikes its encoding	Intercepting Filter
Enforcing J2EE authentication and authorization (as per Chapter 5 on security)	Neither Intercepting Filter nor Front Controller. You want to protect individual components declaratively in the deployment descriptor as far as possible.
Enforcing custom authentication or authorization rules	Depends: could be Intercepting Filter or Front Controller. Use Intercepting Filter if these are "blanket" rules, applying to all (or most) resources. If authentication and authorization needs to be closer to business logic, use Front Controller.

EXERCISE 10-2

ON THE CD

Front Controller Pattern

Would the Front Controller pattern help with the following requirement? List the pros and cons of adopting the approach. A company has a medium-sized web application that is undergoing extensive enhancement. The company realizes that the JavaServer Pages in the application should be reorganized into different directories relating to subsystems within the application, but that this will upset the many embedded links within the current pages. Because the company is working on all aspects of the pages, changing the links is not a big deal—but the company would like to protect itself from URL volatility in JSPs in the future.

If you run the working example that accompanies this chapter, select the Front Controller pattern radio button option, and click the "Solution to Exercise" button, you'll find a discussion of the above scenario. The URL to launch the working example is likely to be

```
http://localhost:8080/lab10/controller
```

CERTIFICATION OBJECTIVE

Model View Controller Pattern (Exam Objectives 11.1 and 11.2)

Given a scenario description with a list of issues, select a pattern that would solve the issues. The list of patterns you must know are: Intercepting Filter, ***Model-View-Controller,*** *Front Controller, Service Locator, Business Delegate, and Transfer Object.*

Match design patterns with statements describing potential benefits that accrue from the use of the pattern, for any of the following patterns: Intercepting Filter, ***Model-View -Controller,*** *Front Controller, Service Locator, Business Delegate, and Transfer Object.*

Model View Controller doesn't quite fit with the other five patterns on the list for the SCWCD exam. It sits at a higher level than the others. You combine several J2EE patterns to make up a Model View Controller architecture, some of which are on our exam hit list (such as Front Controller and Business Delegate) and some of which are not (View Helper and Dispatcher View).

Model View Controller isn't just a J2EE pattern—it has a far longer pedigree going back to the heydays of the object-oriented language SmallTalk. If you explore web sites dedicated to discussions of patterns (they exist!), you will find more disagreement and controversy about the definition of Model View Controller (MVC) than almost any other pattern. But don't be daunted—this section of the chapter will pick out the incontrovertible bits that you need for the SCWCD.

Model View Controller

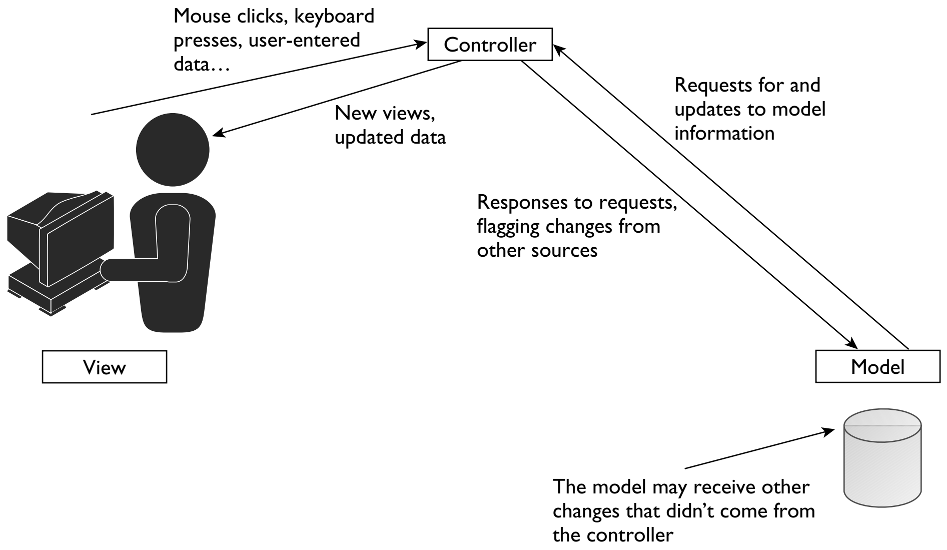
The main motivation for using MVC is to separate your presentation logic from your business logic. The idea is to have a "controller" component that mediates between the two—something we have already explored in the preceding section on the Front Controller pattern. You can see the Model View Controller pattern depicted in Figure 10-9.

The view is far from unintelligent—your view code may be the most complex part of your application (all the more reason not to tangle it up with business logic). Figure 10-9 indicates two principle responsibilities for the view:

- To detect each user action (like a mouse click or an ENTER keypress) and transmit it to the controller component

FIGURE 10-9

The Model View Controller Pattern


Mouse clicks, keyboard presses, user-entered data…
Controller
New views, updated data
Requests for and updates to model information
Responses to requests, flagging changes from other sources
View
Model
The model may receive other changes that didn't come from the controller

- To receive information back from the controller and organize this appropriately for the recipient

The view can range in complexity from some text on a screen and a gray button or two to a fully interactive display resplendent with color. Indeed, different circumstances may demand entirely different views.

From our previous discussion of the Front Controller pattern, you can predict the likely responsibilities of the controller:

- To supply the view with the information it needs—usually by obtaining information from the underlying model
- To interpret user actions and take appropriate actions—usually updating the model and switching view (or both)
- To receive changes from the model that come from other sources, and update views as appropriate

That third and last responsibility causes some problems in the web browser environment—see the nearby On the Job comments.

Finally, what are the model's responsibilities? Easy: to look after the underlying data. It must respond to requests for information and update from the controller. It must (or should) inform the controller of external changes to the data that didn't arrive through the controller.

What I've described here is the scale at which Model View Controller works in J2EE terms—at application level. You should be aware, though, that MVC can be found at the micro as well as the macro level. Swing components have the same idea: a visual component, the view (like a JTable); an underlying representation of the data, the model (JTableModel); and the Swing framework acting as controller mediating between the two.

But because our focus is J2EE web applications, we need to consider where we are likely to find the constituent parts of MVC. To start with, where is the view? Typically, the view is represented by a JavaServer Page. At run time, though, it's really the web browser that acts as the view component. It's not literally the JSP you interact with—that's been translated to pure HTML and delivered to your browser. When you type in text and click buttons, you are using the browser, which sends the results back to the controller component in your design. This is something of an academic point, for it is ultimately the JSP that dictates the availability of those text fields and buttons. So in design and development terms, the JSP is the view.

exam watch

There is nothing stopping you from having other components for views—you don't have to have a JavaServer Page. A pattern is a "logical" design, with several possible physical implementations. A servlet could act as a view component (though has the disadvantage of embedding HTML in code, which we discussed much earlier in the book). Maybe your aim is to present a graphical chart that allows client-side interaction (reordering columns, changing line colors). In these circumstances, an applet may be a good choice. When you meet questions on the exam, don't be too narrow in your interpretation of what a view can be. A view may not even be graphic—just because your client is another business application running silently in batch mode, it can still have a view on your system.

Look out for any of these scenarios—they probably indicate use of the Model View Controller pattern:

- ***When you see the words "presenting data in different ways"***
- ***When the presentation changes dependent on user feedback***
- ***When the data has to be presented to different types of users***

We've already seen (in J2EE terms) that an appropriate choice for a controller component is often a servlet (with assistant classes where necessary). The model is composed—more often than not— of Enterprise JavaBeans (though there are good reasons why we might want to place some classes in between our controller and the Enterprise JavaBeans, as we'll see with the Business Delegate and Service Locator patterns).

on the job

In an ideal world, changes to the model from other sources would result in updates to the appropriate views. Suppose you have a web page where you are viewing volatile stock quotes. Your J2EE application has a real-time feed from another application, which updates quote figures every 30 seconds, so the model data is always up to date. Ideally, you would like the model to be able to tell the controller "I've changed." Then the controller could update the view.

If your view client was a Swing application with a permanent connection to the controller, this could be achieved easily. The trouble is that the web is a connection-less world. Unless you press the refresh button in your browser, you won't see the latest figures. Consequently, you are more or less reliant on a browser client "pulling" fresh changes from the controller. In the cruel

real world, you may have to fudge the illusion of the controller "pushing" to the client by inserting some tawdry bit of script in your web page to cause an auto-refresh at given intervals.

How Is Model View Controller Used in the Example Application?

The example application for this chapter separates out model, view, and controller in ways that you have probably already guessed. Here they are anyway:

- The view consists of three JSP documents: home.jspx, pattern.jspx, and error.jspx. These views are more or less dumb consumers of request attributes—they aren't very clever (so all the better for later maintenance).
- The controller is the FrontController class, which we have already examined in some detail. We have seen how it uses logic to direct to the different JSP documents, and how it populates those request attributes required by the views.
- The model is the BusinessDelegate class, and everything behind it. We'll explore this in the later patterns.

Why Use Model View Controller?

Imagining this question asked out loud at a design patterns conference, I can almost hear the collective sharp intake of breath. MVC is such a sacred pattern that you are practically obligated to use it in every circumstance—whether you are adding a Swing widget to a screen design or putting together a complex J2EE architecture for an investment bank.

Perversely, though, I'll start this section by exploring some reasons for not adopting MVC. By its nature, MVC increases the number of moving parts in your application. There is more overall complexity. Very few of the examples in this book (present chapter notably excepted!) use a Model View Controller design. This could be down to bad design choices on my part, but I prefer to claim that MVC would have just got in the way of explaining other concepts. The undoubted benefits of MVC are outweighed by the complexity it brings.

Teaching examples are a special case, though. What about real J2EE applications that avoid MVC? One of the JSTL libraries contains custom actions dedicated to SQL (database access functionality). This library has a host of useful facilities for embedding database access directly in your JavaServer Page. But hang on—because the JavaServer Page is normally the view, and the database is ultimately a model, where is the controller? You could perhaps use some JSPs with embedded SQL actions separate from JSPs representing the view, and have controlling logic separating them. But this seems perverse: You lose the convenience of embedding an SQL tag right in an HTML table where you want it, and you still end up putting the model code inside a JavaServer Page. It doesn't feel like the purpose for which the JSTL SQL library was intended.

Therefore, many J2EE architects would avoid using the JSTL SQL components altogether, just because they fundamentally violate MVC rules. I would say that it depends on the job at hand. If you have a one-developer project, where the developer has good web design and Java skills, then the SQL components might be ideal for getting a quick-and-dirty version of your application into existence. Later you might choose to re-architect the system along cleaner lines.

So after that exploration of when not to use Model View Controller, let's enumerate the reasons for using it:

- You want to reduce dependencies between the view code and the business model code. That way,
  - You can graft on additional views much more easily (so you might have a web client and a Swing client showing alternative views of the same model data).
  - You can change the view without affecting the business model, and vice versa.
  - The controller is the only volatile component that might be affected by changes at either end.
- You are using a framework that has MVC built into it. Struts is a very popular framework. If you are in a development team that uses it, you are forced into a set of programming standards that abide by MVC rules.
- You want the maintenance benefit that comes from separating the layers. Expert Java programmers can concentrate on the controller and model ends of the application. Web designers (potentially, nondevelopers) can concentrate on the JSP view side of things. Alternatively, if you are using a fancy graphical view, expert Swing Java programmers can concentrate on the visual aspects and be freed up from most concerns about model access.

EXERCISE 10-3

ON THE CD

Model View Controller Pattern

Consider again the company featured in the first exercise, 10-1. It had an extranet application executing business logic to manufacture comma-delimited files or XML for various suppliers. A further requirement arises. Some consumers of the same data don't want file downloads: They need a view of the data presented in a simple web page with an HTML table. How could Model View Controller be integrated into the application to fulfill this requirement, without disturbing the way that existing users of the system work?

If you run the working example that accompanies this chapter, select the Model View Controller pattern radio button option, and click the "Solution to Exercise" button, you'll find a discussion of the above scenario. The URL to launch the working example is likely to be

```
http://localhost:8080/lab10/controller
```

CERTIFICATION OBJECTIVES

Business Delegate Pattern (Exam Objectives 11.1 and 11.2)

Given a scenario description with a list of issues, select a pattern that would solve the issues. The list of patterns you must know are: Intercepting Filter, Model-View-Controller, Front Controller, Service Locator, ***Business Delegate****, and Transfer Object.*

Match design patterns with statements describing potential benefits that accrue from the use of the pattern, for any of the following patterns: Intercepting Filter, Model-View -Controller, Front Controller, Service Locator, ***Business Delegate****, and Transfer Object.*

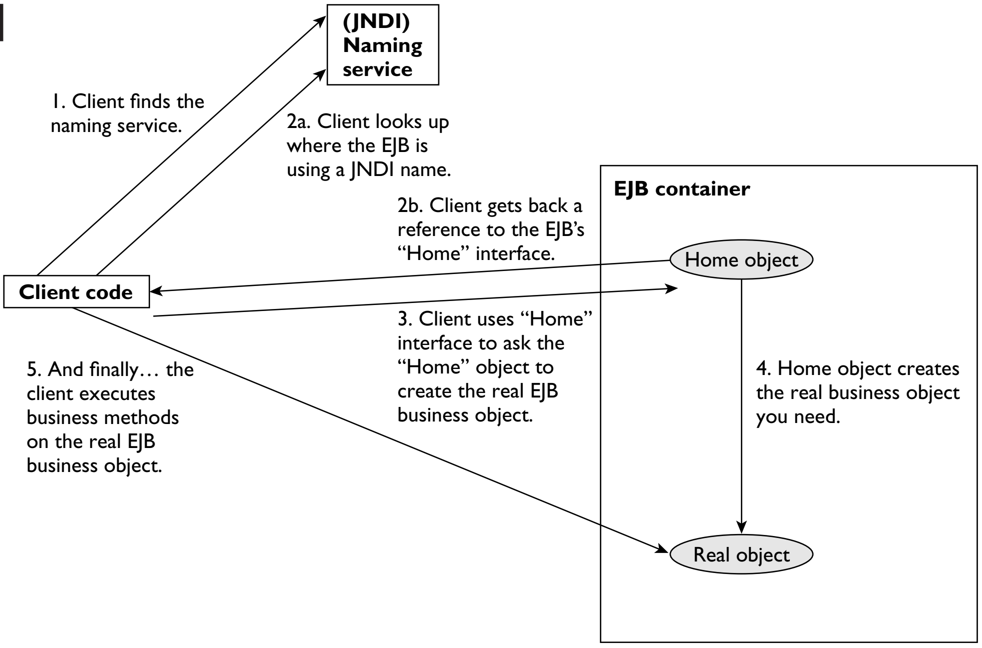
Model View Controller damps down the destructive aspects of change that ripple through the presentation tier. What, though, if you want to insulate your presentation tier more fully from the effects of volatility at the model end? Enter the Business Delegate Pattern, which creates a buffer zone for business APIs.

Business Delegate Pattern

The Business Delegate pattern works by placing a layer in between a client (possibly a Front Controller object) and a business object. That way, even when business object interfaces change, it may well be possible to absorb the change within the business delegate object—without changing the interface that this presents to its clients.

FIGURE 10-10

Using an Enterprise JavaBean


(JNDI) Naming service
1. Client finds the naming service.
2a. Client looks up where the EJB is using a JNDI name.
EJB container
2b. Client gets back a reference to the EJB's "Home" interface.
Home object
Client code
3. Client uses "Home" interface to ask the "Home" object to create the real EJB business object.
4. Home object creates the real business object you need.
5. And finally… the client executes business methods on the real EJB business object.
Real object

It's not just about change. The other service a Business Delegate typically provides is encapsulation of all the ugly details that might be involved in getting hold of the business object in the first place, and coping with any errors thrown up. Without turning this chapter into an EJB tutorial, consider the process shown in Figure 10-10. This shows (in slightly simplified form!) a typical process involved in executing a business method on an Enterprise JavaBean.

There are three independent processes running. First is our client-side code. Second is the "Naming Service"—like a phone directory, which translates names to references to objects. Third is the EJB container. These three processes may run as threads inside the same JVM, within separate JVMs on the same physical computer, or on three separate computers. The actual implementation doesn't matter, as long as you get the idea that what's going on here is a big deal. There is plenty of potential for breakdown in the message passing between the three participants.

Figure 10-10 describes the process involved in obtaining the bean. The code to achieve this (not shown here) is not so *very* frightening—it distills down to a half a dozen lines of code, doubled or tripled by whatever mechanics you put in place to catch the many exceptions that might result—RemoteExceptions (even where the network is all local to one machine), CreateExceptions (because something goes wrong in the EJB creation mechanics), and so on. However, the code involved is not beginner material: You need to know plenty about EJBs to handle them successfully. Given that there is a parallel certification exam dedicated to EJBs alone, you can believe that gaining the knowledge is a long and painful process. Already you have a motive for burying this code in a layer of its own and giving your Java web application developers a simpler API for accessing business data (after all, they have enough to think about—witness the thickness of this book!).

That's not all of it, though. We mentioned exceptions above. The process of looking up the name of your EJB involves JNDI (Java Naming and Directory Interface) code. This can kick up some pretty intimidating exceptions just by itself—InterruptedNamingExceptions and NameNotFoundExceptions, to name but two. These may be important for support developers trying to diagnose a problem with your application, but you may want some friendlier business-type exception to pass through the presentation tier and ultimately drive whatever nonthreatening message shows to the user of the application. And we haven't even started on EJB exceptions.

Still that's not all. There are vendor-specific issues. Surely, J2EE (including the JNDI and EJB) specifications involve standard interface calls, such that your code is portable from one application server to another? Well, yes they do—but there may still be vendor-specific details. Let's pick on JNDI again to furnish an example. As Figure 10-10 shows, the long road to an EJB business method call begins with finding your JNDI Naming Service. This involves getting hold of a JNDI object called an InitialContext. The parameters for getting hold of the InitialContext (which host to look up, which port, etc.) do vary from one platform to another. How to feed the parameters to the calling code (directly or via properties files) varies as well.

With luck, you are getting the idea that it might be a good idea to confine some of this code to easily accessible regions of your web application. And that is, of course, where the Business Delegate pattern comes in. It is used to

- Encapsulate EJB (and other complex business object access) code
- Encapsulate JNDI code (but often defers to the Service Locator pattern for that—see the next section of the chapter)
- Present simple business interfaces to presentation-tier clients
- Cache (sometimes) the results of expensive calls on business objects

Finally in this description, where does the Business Delegate belong? Because it has the name "Business" within it, you would assume the business tier. In physical terms, though, the classes involved are likely to remain close to your presentation code (the EJB server itself is potentially remote). Perhaps for this reason, it's described as a *client-side* business abstraction and often regarded as belonging to the (physical) presentation tier.

exam watch

The presence of the acronym "EJB" in an exam question scenario is very likely to imply the use of Business Delegate. However, EJBs are not the only kind of business object that can be fronted by the Business Delegate pattern. Be sensitive to other possibilities and phrases that describe systems for managing information: "business rules" and "stored procedures," for example.

How Is Business Delegate Used in the Example Application?

The example application has a class called BusinessDelegate that exhibits most of the Business Delegate pattern features. Here is the code for the BusinessDelegate class (minus package statement and imports):

```
public class BusinessDelegate {
  Map patternCache = new HashMap();
  ServiceLocator serviceLocator = new ServiceLocator();
  PatternLoaderRmtI patternLoader;
  public PatternTfrObj findPattern(String patternName)
    throws PatternNotFoundException {
    // Is the pattern already in the cache?
    PatternTfrObj pattern = findCachedPattern(patternName);
    // Not in cache: use remote object
    if (pattern == null) {
      if (patternLoader == null) {
        patternLoader = serviceLocator.findPatternLoader();
      }
      try {
        pattern = findRemotePattern(patternName);
      } catch (RemoteException re) {
       // Print the stack trace for internal diagnosis
        re.printStackTrace();
        // Re-throw an "application" exception
        throw new PatternNotFoundException(patternName);
      }
      patternCache.put(patternName, pattern);
    }
    return pattern;
  }
  protected PatternTfrObj findCachedPattern(String patternName) {
    return (PatternTfrObj) patternCache.get(patternName);
28   }
29   protected PatternTfrObj findRemotePattern(String patternName)
30     throws RemoteException, PatternNotFoundException {
31     System.out.println("Making expensive call to remote API...");
32     delay(3000);
33     if (patternLoader == null) {
34       throw new PatternNotFoundException(patternName);
35     }
36     return patternLoader.getData(patternName);
37   }
38  protected void delay(int millis) {
39     try {
40       Thread.sleep(millis);
41     } catch (InterruptedException ie) {
42     }
43   }
44 }
```

Here are the Business Delegate pattern features that this class implements:

- Its main business method begins at line 05: `findPattern()`. This returns a business object — of type PatternTfrObj — given a named pattern passed as a parameter. So the first point is that this class exposes a business method for use by classes in the presentation tier (in this case, the FrontController class).
- In line 02, BusinessDelegate declares a local cache (a HashMap)—for pattern business objects. Because pattern information doesn't change very much (at all!) in the example application, having a cache makes a great deal of sense. So at line 08 in the `findPattern()` method, the code first looks for the business object in the cache — only if it doesn't find it there will it execute calls to real remote business services. Note that at line 22, after doing a remote call to find a pattern, there is code to place that pattern in the cache.
- At line 15, BusinessDelegate calls its own method—`findRemotePattern()`—beginning at line 29. This calls the real business method on the real remote object. The real remote object (of type PatternLoaderRemoteImpl) is very much like an Enterprise JavaBean: You have to go through RMI to get hold of its methods. Consequently, `findRemotePattern()` might throw a RemoteException. Just to emphasize the expense and remoteness of making the call, the method throws in an arbitrary 3-second delay — this is so you can tell the difference when information comes out of the cache, which has no such built-in artificial delay.

- If the `findRemotePattern()` method fails to find a pattern, it throws a business-type exception: PatternNotFoundException (at line 34).
- There's still the possibility that RemoteException will be thrown—so back in the calling code, the call to `findRemotePattern()` is couched in a try-catch block. You can see how—in lines 17 to 20—the contents of the RemoteException are printed in the stack trace for later diagnosis. Then—at line 20—the application-friendly PatternNotFoundException is thrown for the benefit of the presentation code.
- One thing the BusinessDelegate doesn't do is contain code to find the remote business object in the first place (which involves JNDI code and other nastiness). Instead, it defers to a helper class called ServiceLocator, declared at line 03 and used at line 12. We will explore that in a subsequent part of the chapter devoted to the Service Locator pattern.

So you can see this BusinessDelegate class does most of what you would expect from the Business Delegate pattern: It encapsulates difficult-to-handle business objects, has its own cache for performance gains, and translates technical errors to application errors.

Why Use Business Delegate?

By now, most of the reasons for using Business Delegate have been exposed. Here's a summary:

- To reduce coupling between clients in the presentation tier and actual business services, by hiding the way the underlying business service is implemented.
- To cache the results from business services.
- To reduce network traffic between a client and a remote business service.
- To minimize error handling code (particularly network error handling code) in the presentation tier.
- Substituting application-level (user-friendly) errors for highly technical ones.
- If at first the business service doesn't succeed, the Business Delegate class might choose to retry or to implement some alternative API call to recover a situation. A business service failure doesn't immediately have to be passed on to a client.
- Make naming and lookup activities happen within the Business Delegate—or code that the Business Delegate uses (see Service Locator).

- To act as an adapter between two systems (B2B-type communication—where a visible GUI isn't involved). The delegate might interpret incoming XML as a business API call.

EXERCISE 10-4

ON THE CD

Business Delegate Pattern

This time, you are managing a project to build a new web application to extend the functionality of an existing legacy COBOL system. The COBOL system has a number of programs that offer stable, well-tested business services and whose code you don't want to duplicate or change. There are Java APIs to handle calls to COBOL, but you know that they are tricky to write and use and that they throw a multitude of obscure exceptions, and you have at your disposal only a single developer skilled in their use—and he spends most of his days supporting existing systems. How could the Business Delegate pattern help the project and the future support of your new web application?

If you run the working example that accompanies this chapter, select the Business Delegate pattern radio button option, and click the "Solution to Exercise" button, you'll find a discussion of the above scenario. The URL to launch the working example is likely to be

```
http://localhost:8080/lab10/controller
```

CERTIFICATION OBJECTIVE

Service Locator Pattern (Exam Objectives 11.1 and 11.2)

*Given a scenario description with a list of issues, select a pattern that would solve the issues. The list of patterns you must know are: Intercepting Filter, Model-View-Controller, Front Controller, **Service Locator**, Business Delegate, and Transfer Object.*

*Match design patterns with statements describing potential benefits that accrue from the use of the pattern, for any of the following patterns: Intercepting Filter, Model-View -Controller, Front Controller, **Service Locator**, Business Delegate, and Transfer Object.*

The Service Locator pattern often acts in conjunction with the Business Delegate. We already discussed Business Delegate as a way of offloading the "technical" aspects of calling business APIs—you use a Business Delegate to do this instead of letting presentation code perform this task directly. What Service Locator does is to allow a Business Delegate class to further offload the most gruesome aspects of Business Service access code, which is normally the sleight of hand involving looking up the business service in the first place and getting some kind of reference to it. Thereafter, the Business Delegate can happily call APIs on the business object. So Service Locator encapsulates the mysteries of finding business objects (services).

The Service Locator Pattern

Service Locator is like a subset of Business Delegate—you might even include "service locator" code in methods within your "business delegate" class. It's a valid implementation of the pattern. However, I prefer the obvious separation of different classes—even in the small-scale example application that accompanies this chapter.

The main reason for using the Service Locator pattern is to avoid duplication of code that gets references to business resources (services) you need to use. To start the ball rolling, you most often use JNDI (Java Naming and Directory Interface) code, and this can easily proliferate across all kinds of clients. Furthermore, the operation of looking up some resource in a JNDI-compliant naming and directory service is expensive, so if the results of retrieving the resource can be cached rather than repeated, there is a performance saving to be had.

If you look at the J2EE Core Patterns book (or web site on line), you'll see that there are lots of specific "strategies" for the Service Locator pattern that have mainly to do with the particular kind of J2EE resource Service Locator is after. You might be after Enterprise JavaBeans. Or JMS (Java Messaging Service) objects. Or other things not mentioned: legacy programs in Enterprise Information Systems, or Java Mail, or URL factories. I boldly suggest that the specifics of these don't matter at all. The general principle holds true in all cases. Your Java code needs to get hold of a reference to a business service. That's code you will want to isolate.

How Is Service Locator Used in the Example Application?

Let's see how the ServiceLocator class in the example application contains the pattern features. At first sight, it doesn't look like much—here's the class (minus package and import statements):

```
public class ServiceLocator {
  PatternLoaderRmtI patternLoader;
03   public PatternLoaderRmtI findPatternLoader() {
04     if (patternLoader == null) {
05       initializePatternLoader();
06     }
07      return patternLoader;
08   }
09   protected void initializePatternLoader() {
10     try {
11       // Needless 6-second delay to make the point that looking up
12       // a remote registry might be expensive.
13       System.out.println("Making expensive call to naming registry...");
14       delay(6000);
15       patternLoader = (PatternLoaderRmtI) Naming
        .lookup("rmi://localhost/patternLoader");
16     } catch (MalformedURLException e) {
17       e.printStackTrace();
18     } catch (RemoteException e) {
19       e.printStackTrace();
20     } catch (NotBoundException e) {
21       e.printStackTrace();
22     }
23   }
24   protected void delay(int millis) {
25     try {
26       Thread.sleep(millis);
27     } catch (InterruptedException ie) {
28     }
29   }
30 }
```

The first and only public method in the class is called `findPatternLoader()`, and it returns a business object of type PatternLoaderRemoteI. The real work to get hold of this object occurs in the protected method `initializePatternLoader()`, but notice that if the class has gone to the trouble of executing this method already, it keeps hold of the reference (lines 04 to 07).

Within `initializePatternLoader()`,, you first of all find an artificial delay built in (lines 13 and 14). This is not part of the Service Locator pattern—your real application architecture is unlikely to need any artificial delays! It makes the point that the reference-finding process can be a lengthy one (if the delay wasn't there, you wouldn't notice the difference between cached calls and remote calls when running all the example components on your desktop PC). The real work is done at line 15. This uses a JNDI class called Naming to look up the remote business object in the RMI Registry. You can see from the stack of catch clauses that follow (lines 16 to 22) that there is plenty of potential for disaster.

If the JNDI call works, the code casts the object returned to the type of business object expected: a PatternLoaderRemoteI reference. This is held as an instance variable in the class, so there is no need to repeat the JNDI call.

At last the business of finding is done: The result (for clients—the BusinessDelegate class in our case) is an already found reference to a business object. Hiding RMI code like this may seem trivial. However, it makes the point that you can easily trap this code in one place. Inasmuch as this is a simulation of EJB use (without the need for an EJB container), take a look back at Figure 10-10. The Service Locator encapsulates the steps numbered 1 to 3 on the figure: everything from finding a naming service to getting hold of the remote object. This leaves the Business Delegate to concentrate solely on step 5: executing business methods on the remote object once found.

So when you come to writing classes that deal not with simple RMI objects but with bigger business components—such as EJBs—you are likely to be grateful to the Service Locator pattern.

Why Use Service Locator?

The reasons for using the Service Locator pattern are these:

- To hide JNDI (or similar) code that gets hold of a reference to a service. Even though JNDI code is pure, portable Java, the parameters fed into JNDI code can be vendor-specific, so locating all these details in one place (rather than leaking them all over your business objects) is desirable if you want to port your application later with minimum hassle.
- By locating JNDI (or similar) code in one place—or a few places—you avoid copying and pasting technical calls across many business objects.
- To minimize network calls—the Service Locator can decide which JNDI calls need to be made and when it has appropriate references to remote objects already. This can, of course, improve performance.

EXERCISE 10-5

ON THE CD

Service Locator Pattern

Consider the scenario from the previous exercise, where you used the Business Delegate pattern to front calls to legacy COBOL programs. How could the Service Locator pattern be used to further ease the maintenance of your web application?

If you run the working example that accompanies this chapter, select the Service Locator pattern radio button option, and click the "Solution to Exercise" button, you'll find a discussion of the above scenario. The URL to launch the working example is likely to be

```
http://localhost:8080/lab10/controller
```

CERTIFICATION OBJECTIVES

Transfer Object Pattern (Exam Objective 11.1 and 11.2)

Given a scenario description with a list of issues, select a pattern that would solve the issues. The list of patterns you must know are: Intercepting Filter, Model-View-Controller, Front Controller, Service Locator, Business Delegate, and ***Transfer Object****.*

Match design patterns with statements describing potential benefits that accrue from the use of the pattern, for any of the following patterns: Intercepting Filter, Model-View -Controller, Front Controller, Service Locator, Business Delegate, and ***Transfer Object****.*

The idea of the Transfer Object pattern is to encapsulate data—usually the data that would be available inside a complex (and possibly remote) object, such as an Enterprise JavaBean (EJB). The class representing a Transfer Object is a very simple thing, but what goes on around it is moderately profound and complex, and that's what we'll explore in this final section on J2EE design patterns.

The Transfer Object Pattern

Structurally, the object representing a Transfer Object is very simple. It's a Java Bean—not an Enterprise JavaBean—just a bean. And a bean only in the sense that is has a bunch of properties that can be accessed with `set` and `get` methods.

The idea of a Transfer Object is that it holds the data returned from a more complex business object, such as an Enterprise JavaBean. You may well ask a question: Wouldn't it be more straightforward to deal with the EJB business object directly? Why bother with additional objects? To understand that, let's consider client code that deals with a type of EJB called an Entity EJB. It's a big simplification, but more often than not, an Entity EJB represents a single row on a table (and has attributes to match). Let's consider a fashion retailing example. Suppose that we have a database table to record information about dresses — one row per dress. And suppose we tie an Entity EJB definition to this table, as shown in Figure 10-11.

You see how the attributes of the EJB (each with their own getters and setters) map on to equivalent table fields—fashionSeason to FSHN_SEASON, and so on. Let's consider what might happen if you had hold of the EJB and executed the `getSizeRange()` method. This might well be a remote call across the network. Because EJBs are often designed to disguise the JDBC database access details from you, your EJB might (under the covers) generate some JDBC SQL SELECT statement to get hold of the individual piece of information you are after: the size range for the dress. Now suppose that you want to get hold of the dress color. You make a call to `getColor()`, which goes remotely across the network and instigates a new and separate JDBC SQL SELECT statement to get hold of the color attribute from the same table row.

It should be said at this point that no sane EJB container vendor manages SQL access in such a naïve way. However, you should be getting the message that access

FIGURE 10-11

An EJB Mapping to a Database Table

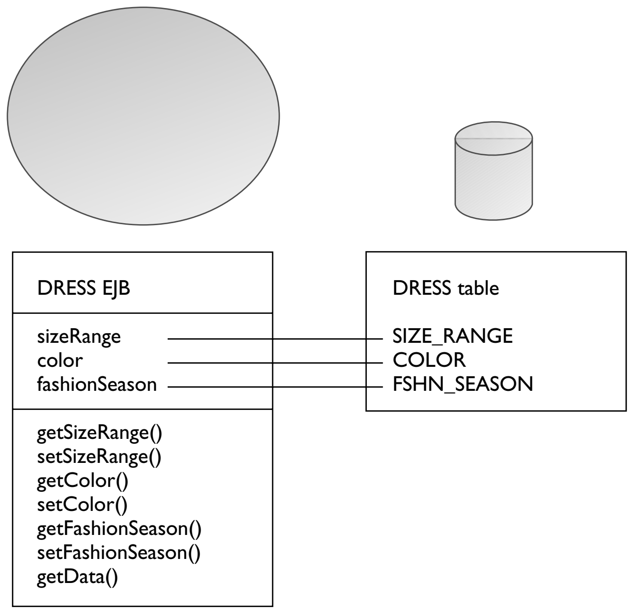
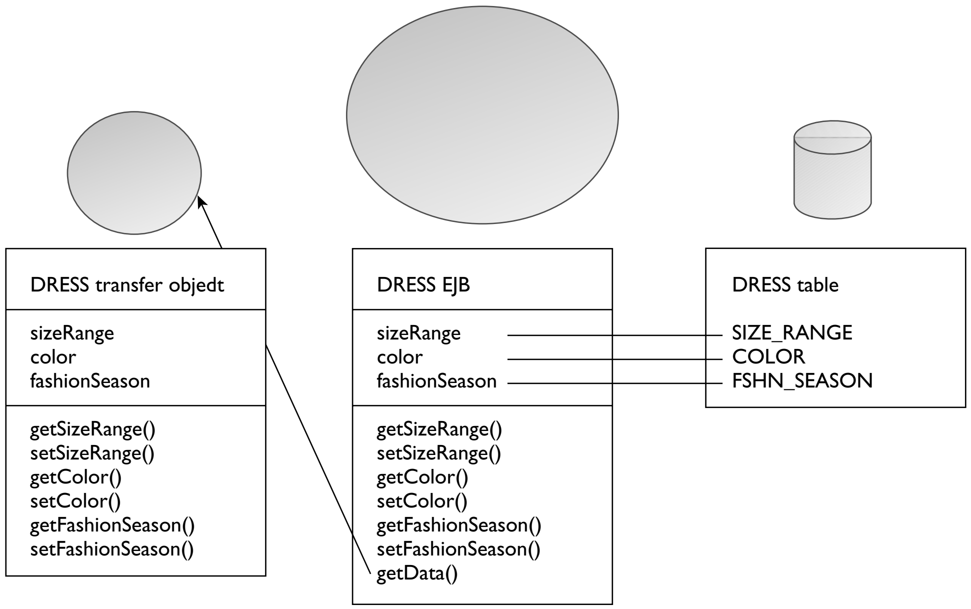


FIGURE 10-12

EJB `getData()` Method Returning A Transfer Object



to separate attributes in this piecemeal fashion can be expensive in networking and processing terms. The answer? To get hold of all the attributes you need together. Place in the EJB a `getData()` method that returns all three attributes at once—color, fashion season, and size range. `getData()` returns an object—a transfer object—that has all attributes of the EJB but none of the complexity. You might not want to know abut the fashion season for your particular call to `getData()`, but it's far cheaper to collect all the data you're likely to need in one hit and pass it back in one package—as shown in Figure 10-12.

on the job

Classes for Transfer Objects are always implement Serializable so that they can be returned from remote method calls.

You can even apply the same principle in reverse. A client may want to update this row on the Dress table through the system. A client can make changes to its own copy of the transfer object, then pass this back to the Enterprise JavaBean via a `setData()` method. That gives the EJB code the opportunity to execute a single SQL update statement for all the changed attributes in one hit, instead of updating each attribute in turn.

exam watch

Transfer Object is the design pattern formerly known as Value Object (if you were around for the previous version of J2EE, or are upgrading from the previous version of the SCWCD). Almost none of the details have changed—it just seemed a more appropriate name to the J2EE pattern gurus (because it's to do with the transfer of information).

How Is Transfer Object Used in the Example Application?

Bearing in mind that the example application doesn't use actual EJBs, you still get the full Transfer Object pattern rendered for your money. The simulation EJB is the class called PatternLoader. Rather than connecting to a database, this class maps on to a properties file in the file system, which loads into a Properties object. PatternLoader has its own `getData()` method, which returns a PatternTfrObj (Pattern Transfer Object). This takes data from the Properties object and populates attributes on the PatternTfrObj by calling appropriate getter methods. For brevity, the code isn't shown here—the source is, of course, available in the solution code.

You can follow the PatternTfrObj back through the system. BusinessDelegate makes the call to the `getData()` method. It passes the transfer object back to FrontController. As we've seen, FrontController strips the values into request attributes, which are then used by the JavaServer Pages. Alternatively, the PatternTfrObj could have been made directly available with the JSP as a bean (using the `<jsp:useBean>` custom action, for example).

The example application doesn't go as far as providing a `setData()` method on PatternLoader, whereby an updated version of the PatternTfrObj could be passed back in order to update the underlying properties file.

Why Use Transfer Object?

Having seen how the Transfer Object pattern works and where it fits, we just need to give some thought to why you would employ the pattern. These are the benefits:

- Transfer Objects simplify your life on the client side. You avoid direct dependence on potentially complex business objects and deal instead with relatively simple bean-like objects.

- You avoid the expense of repeated interaction with remote business objects. This occurs when you accumulate dribs and drabs of data from those business objects by a series of expensive remote calls. Better to pass all the data you are likely to need in one remote "hit." This may seem profligate if you never use some of these attributes, but this is likely to be a small overhead compared even to one unnecessary network call.

There are some real drawbacks to using the Transfer Object pattern, however. A Transfer Object is likely to duplicate code—notably in the attributes and getter and setter methods that shadow those of its associated Enterprise JavaBean (or other business component). That said, there are ways to mitigate this. You could consider coupling the Transfer Object and business component by having the business component extend the Transfer Object and so inheriting its attributes and getter and setter methods. Alternatively, you might have a tool available to generate Transfer Object code directly from your business component. These strategies are not that palatable, though, just because they tie together an object deep in the model with an object that is used—potentially—on the fringe of the presentation layer. You could imagine adding a fresh attribute to the EJB, regenerating the code for the associated Transfer Object, and forgetting to revisit a particular client using the Transfer Object (because it had no particular reason to use the new attribute).

Transfer Objects can also go stale. Vendors of EJB containers go to great lengths to ensure that EJBs keep up to date with changes from multiple users of the EJB. As soon as you copy the data from the EJB as you do with the Transfer Object, you have an object bouncing around your system with data that will inevitably get out of date. This may not matter very much, especially when you are only reading data. However, you may not be able to remain complacent over multiple user update issues. You may happily update values in the Transfer Object you are using. Yet in the meantime, someone may have sneaked up and updated the underlying business component. When you press the update button, what should happen? Should the old values in your Transfer Object (as well as your changes) overwrite the changes from the other user—which you were never aware of? Or should your changes be rejected? To protect against this, you may introduce complexity into your code—by adding some form of version control or locking to your system. When you deal directly with EJBs, you could end up attempting to solve problems that EJB container designers have already solved.

EXERCISE 10-6

ON THE CD

Transfer Object Pattern

For this exercise, we'll stick with the web application tied to legacy COBOL programs that has dominated the previous two exercises. The COBOL program update routines will disallow your update to a row on a table, if this row has been updated by another user since you read the record for update. How could you use Transfer Objects to integrate with this mechanism?

If you run the working example that accompanies this chapter, select the Transfer Object pattern radio button option, and click the "Solution to Exercise" button, you'll find a discussion of the above scenario. The URL to launch the working example is likely to be

```
http://localhost:8080/lab10/controller
```

CERTIFICATION SUMMARY

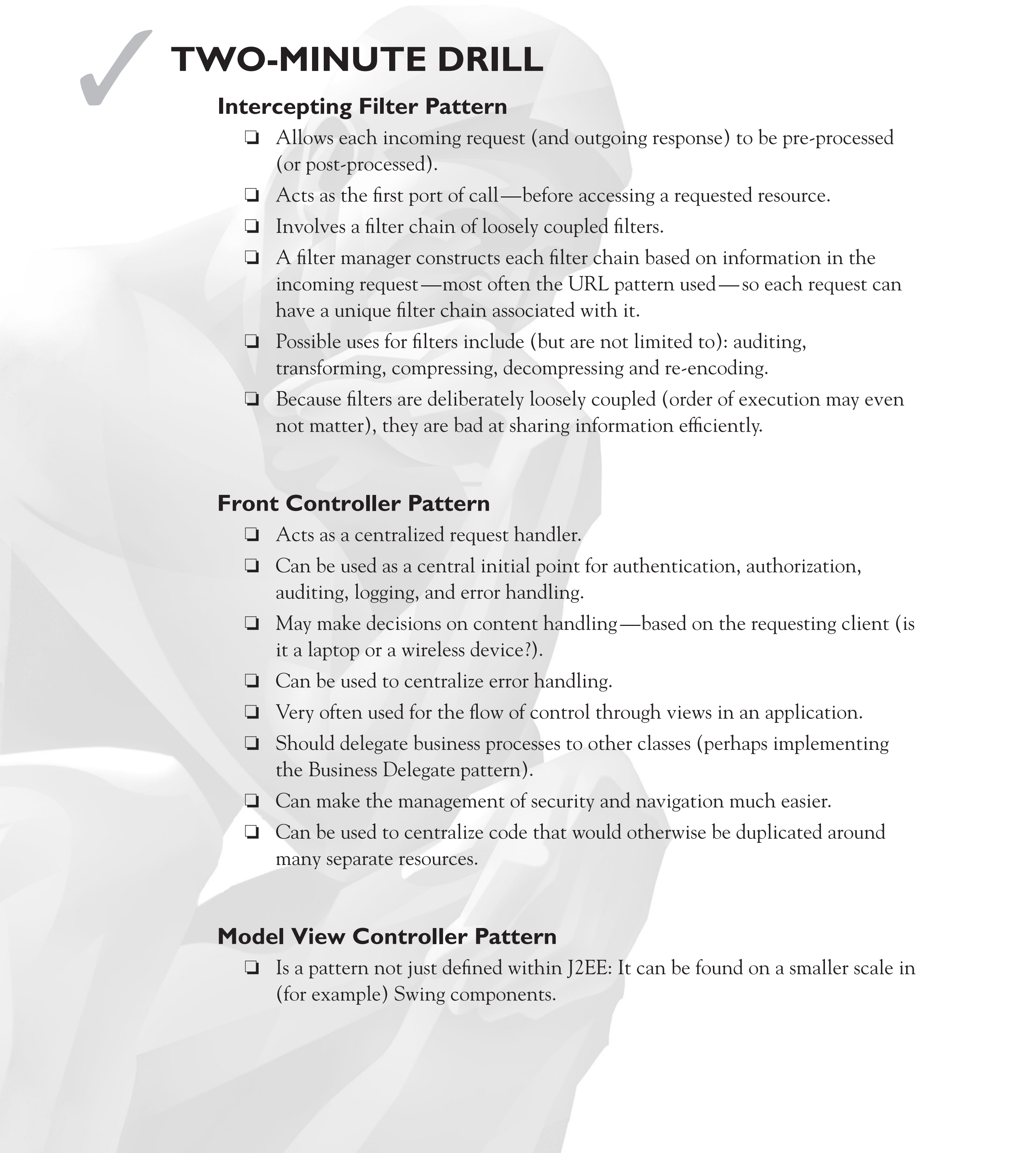
In this chapter you surveyed around a quarter of the J2EE core patterns catalogue, concentrating on the six J2EE patterns you need to know for the SCWCD exam.

You started by thinking about patterns in general—and how they supply a logical design solution to a particular design problem. You examined

- Intercepting Filter—and saw that this is a logical description of the Filter mechanisms you met in Chapter 3. You learned that Intercepting Filter can be used to manipulate or validate requests and responses on their way into or out of a web application. You saw that filters can be independent and loosely coupled, with a chain controlling the passage from one to another. You learned that a principal benefit of a filter chain like this is the avoidance of conditional code to fulfill the same function copied and pasted around several points of entry to your application.
- Front Controller—you learned that this is used as a gateway into a web application and can be used as a point where a request is authenticated before passing on to real resources within the application. You also saw that navigation

can be greatly simplified by adopting Front Controller. If all links lead to the Front Controller, you appreciated how navigation might be controlled by logical links.

- Model View Controller—you saw that this is a bit less specific than the other patterns; it describes how to separate out presentation logic and business logic, then interpose a controller to mediate between them. You saw that in J2EE terms, the View might be represented by JSPs, the controller by a class or classes implementing the Front Controller pattern, and the model by a Business Delegate (or at least, fronted by classes representing the Business Delegate pattern). You learned how the view sends user actions to the controller, and how the controller translates these into actions against the model. You learned that this separation of concerns can help in practical project terms—allowing developers with different skills to concentrate on components matched to their area of expertise.
- Business Delegate—you learned that this can be used as a translation layer between a client wanting to use business routines (like a Front Controller) and the business routines themselves. You further learned that the motive for this is to hide some of the complexity that might come with business routines—for example, where these involve calling methods on Enterprise JavaBeans. You saw also that changes to business routines need not necessarily result in changes to the interface that a Business Delegate exposes to its clients—so in this way, the presentation layer has an element of protection from volatility in the model layer.
- Service Locator—you saw that this is effectively a helper to the Business Delegate pattern. You learned that it is often the case in J2EE applications that just finding the resources you want to use can be an expensive business, involving calls to a naming service through JNDI. You saw how a Service Locator can encapsulate this often complex code, and also cache references to business objects once found.
- Transfer Object—the last pattern you met, this encapsulates data in a simple bean-like way. You learned that the idea of a Transfer Object is to return many items of data at once, instead of making repeated calls to (potentially) remote methods to get attributes one at a time. You saw how this can decrease network traffic, and—by giving the client a simple object to deal with—reduce the complexity of client code.



TWO-MINUTE DRILL

Intercepting Filter Pattern

- ❑ Allows each incoming request (and outgoing response) to be pre-processed (or post-processed).
- ❑ Acts as the first port of call—before accessing a requested resource.
- ❑ Involves a filter chain of loosely coupled filters.
- ❑ A filter manager constructs each filter chain based on information in the incoming request—most often the URL pattern used—so each request can have a unique filter chain associated with it.
- ❑ Possible uses for filters include (but are not limited to): auditing, transforming, compressing, decompressing and re-encoding.
- ❑ Because filters are deliberately loosely coupled (order of execution may even not matter), they are bad at sharing information efficiently.

Front Controller Pattern

- ❑ Acts as a centralized request handler.
- ❑ Can be used as a central initial point for authentication, authorization, auditing, logging, and error handling.
- ❑ May make decisions on content handling—based on the requesting client (is it a laptop or a wireless device?).
- ❑ Can be used to centralize error handling.
- ❑ Very often used for the flow of control through views in an application.
- ❑ Should delegate business processes to other classes (perhaps implementing the Business Delegate pattern).
- ❑ Can make the management of security and navigation much easier.
- ❑ Can be used to centralize code that would otherwise be duplicated around many separate resources.

Model View Controller Pattern

- ❑ Is a pattern not just defined within J2EE: It can be found on a smaller scale in (for example) Swing components.

- ❑ Separates out data (the model) from the presentation of data (the view), by having a controller object interposed between the model and the view.
- ❑ In J2EE terms, the view is often represented by JSPs and the model by business components (like EJBs—fronted by Business Delegates), and the controller is often a servlet (conforming to the Front Controller pattern).
- ❑ The controller selects which view to display.
- ❑ The controller can embed security functionality—maybe disallowing some requests from the view.
- ❑ This pattern can lead to a cleaner approach to application design, for concerns are separated.
- ❑ This pattern can even suit projects where different developers have different expertise—some developers may concentrate on business logic in the model and others on presentation in the view JSPs.

Business Delegate Pattern

- ❑ An abstraction of the business layer, this pattern hides business services.
- ❑ This pattern presents
  - ❑ a route into business services that stays constant even if the underlying business services change in minor ways
  - ❑ simpler access to business services
- ❑ Most often, this pattern is used to stop presentation code from accessing EJBs directly.
- ❑ The complexity of EJB method calls is encapsulated in the Business Delegate.
- ❑ May also cache the results of calling underlying business APIs.
- ❑ By reducing business API calls—which are often networked—a Business Delegate can reduce network traffic.
- ❑ May also cope with failing business services through retry and recovery code, passing on failures to the presentation layer only when there is no other option.
- ❑ May simplify the errors arising from business API calls, translating these to more application-oriented messages.

Service Locator Pattern

- ❑ Is used to find references to business services in the first place.
- ❑ May take some of the load off the Business Delegate pattern.
- ❑ Typically encapsulates calls to naming service (JNDI) routines.
- ❑ May cache references to business objects, avoiding unnecessary repeated JNDI calls.
- ❑ Reduces the need to have JNDI code at many points in the application—wherever a business object must be found.
- ❑ Can help with application server migration; because some details of JNDI are inevitably vendor-specific, it reduces the places where code needs to change.

Transfer Object Pattern

- ❑ Is used to encapsulate data returned from a business service (usually from an EJB).
- ❑ All (or a meaningful subset) of the attributes on a business object are collected together in a transfer object.
- ❑ Takes the form of a simple bean, with get and set methods.
- ❑ Is usually the return type for a `getData()` method on an EJB.
- ❑ The `getData()` method transfers the EJB attributes to the Transfer Object.
- ❑ Is usually the input parameter to a `setData()` method on an EJB.
- ❑ The `setData()` method takes attribute values on a Transfer Object, and updates these on the EJB.
- ❑ Reduces the need for separate API calls to get individual attributes from a remote business object.
- ❑ May contain stale data at variance with the originating business object (because the business object has been updated by another user).
- ❑ May duplicate attribute definition and getter/setter methods from the EJB (or other business object) with which it is associated.

SELF TEST

The following questions will help you measure your understanding of the material presented in this chapter. Read all the choices carefully because there might be more than one correct answer. The number of correct choices to make is stated in the question, as in the real SCWCD exam.

1. Which of the following patterns can reduce network overhead? (Choose one.)

   A. Model View Controller
   B. Business Delegate
   C. Service Locator
   D. Transfer Object
   E. All of the above
   F. B, C, and D above

2. Which pattern insulates the presentation client code from volatility in business APIs? (Choose one.)

   A. Business Delegate
   B. Service Layer
   C. Service Delegate
   D. Service Locator
   E. Model View Controller
   F. Business Service Locator

3. A company has a message queuing system, accessible with complex Java APIs. The company wants a new web application but also wants to minimize the specialized knowledge required to write business code that accesses the queuing system. Which J2EE patterns might best help the company with this problem? (Choose two.)

   A. Front Controller
   B. Front Director
   C. Business Controller
   D. Service Locator
   E. View Controller
   F. Business Delegate

4. From the following list, choose benefits that are usually conferred by the Transfer Object pattern. (Choose two.)

   A. Reduces network traffic
   B. Packages data into an accessible form
   C. Decreases client memory use
   D. Reduces code dependencies
   E. Simplifies EJB development

5. A company wants to structure its development department on specialist lines so that web designers can concentrate on page layout and Java developers can concentrate on business API development. Which pattern should the company use in application development to best support its organizational aims? (Choose one.)

   A. Service Locator
   B. View Helper
   C. Intercepting Filter
   D. Model View Controller
   E. Front Controller
   F. View Controller

6. From the following list, choose benefits that are usually conferred by the Service Locator pattern. (Choose two.)

   A. Acts as a wrapper around business API calls
   B. Reduces strain on the Business Delegate pattern
   C. Encapsulates naming service code
   D. Bridges gaps to non-Java (web services aware) platforms
   E. Caches business data

7. Which of the following patterns might best be used to reject requests from a host machine with “uk” in its domain name? (Choose one.)

   A. Front Controller
   B. Business Delegate
   C. Authentication Guard
   D. Request Analyzer
   E. Intercepting Filter
   F. Service Locator

8. From the following list, choose benefits that are usually conferred by the Model View Controller pattern. (Choose two.)
   - A. Insulation from volatility in business APIs
   - B. Reduced EJB code
   - C. Better graphical representation of information
   - D. Decreased code complexity
   - E. Separation of concerns
   - F. Better project management

9. Which of the following patterns might be used to dispatch a view containing an application-friendly error message? (Choose one.)
   - A. Business Delegate
   - B. Front Controller
   - C. Model View Controller
   - D. View Controller
   - E. Business Handler

10. From the following list, choose benefits that are usually conferred by the Business Delegate pattern. (Choose two.)
   - A. Presentation code stability
   - B. Less JNDI code
   - C. Less layering
   - D. Better business models
   - E. Simplified EJB code
   - F. Better network performance

11. Which scenario below could best use Transfer Object as a solution? (Choose one.)
   - A. An application needs to have its navigation rationalized—at the moment, JSPs have URLs pointing to other JSPs, and the naming is nonstandard throughout.
   - B. Some developers inexperienced with EJBs are having trouble calling business routines throughout a large application.
   - C. An application appears to have hit a bottleneck when using some EJBs. On analysis, it appears that there are numerous method calls to retrieve individual attributes.
   - D. An application appears to have hit a bottleneck through excessive calls to the JNDI naming service fronting an RMI registry.

12. From the following list, choose benefits that are usually conferred by the Front Controller pattern. (Choose three.)

    A. Imposes JSPs as views
    B. Reduces network traffic
    C. Adds a single point of failure
    D. Acts a gateway for requests to the system
    E. Centralizes control of navigation
    F. Reduces complexity of links in JSPs

13. Which of the patterns below will definitely reduce the lines of code in an application? (Choose one.)

    A. Front Controller
    B. Front View
    C. Model View Controller
    D. Intercepting Filter
    E. None of the above

14. From the following list, choose reasonable applications for the Intercepting Filter pattern. (Choose one.)

    A. Re-encoding request or response data
    B. Unzipping uploaded files
    C. Authenticating requests
    D. Invalidating sessions
    E. All of the above
    F. A, B, and C above
    G. A and B above

15. Which of the patterns below might reduce the lines of code in an application? (Choose one.)

    A. Business Delegate
    B. Service Locator
    C. Intercepting Filter
    D. All of the above

LAB QUESTION

Write your own application that uses all the patterns in this chapter. Of course, you have been working with my solution to this throughout the chapter.

SELF TEST ANSWERS

1. ☑ **F** is the correct answer. Business Delegate can reduce network overhead by caching results of executing business methods (where appropriate). Service Locator can cache references to remote objects to avoid making repeated networked calls to (mostly) JNDI code. Transfer Object collects together lots of data that might otherwise be garnered by repeated network calls.
   ☒ **A** is incorrect—Model View Controller is about the separation of concerns and doesn't directly address any network issues. **B**, **C**, and **D** are all patterns that contribute to reducing network overhead, but no one of these answers is correct on its own. **E** is incorrect because MVC is included in the list.

2. ☑ **A** is the correct answer—Business Delegate wraps business APIs. If the business APIs change, it may be unnecessary to change the interfaces that Business Delegate offers to clients in the presentation tier.
   ☒ **B**, **C**, and **F** are incorrect are they are made-up pattern names. **D** is incorrect—Service Locator is used by Business Delegate, so it shouldn't be directly exposed to presentation client code. **E** is incorrect because the Model View Controller doesn't specify how the Controller (the presentation client code in the question) should be kept at arm's length from the model.

3. ☑ **D** and **F** are the correct answers**.** The Service Locator pattern can be used to isolate the code that finds references to the objects that give access to the queuing system. The Business Delegate pattern can be used to translate simple method calls (used by the business code) into the complex method calls required by the queuing system.
   ☒ **A** is incorrect: Front Controller doesn't address complexity at the model end. **B**, **C**, and **E** are incorrect because they have made-up or incomplete pattern names.

4. ☑ **A** and **B** are the correct answers. Transfer Object can reduce network traffic by encouraging a call to one coarse-grained method instead of repeated calls to lots of fine-grained remote methods. Transfer Object also packages data in a convenient and simple bean-like form.
   ☒ **C** is incorrect—potentially, bulky Transfer Objects could increase client memory usage rather than reducing it. **D** is incorrect—Transfer Object doesn't reduce code dependencies; it permeates several layers, so it increases them instead. **E** is incorrect—the Transfer Object pattern implies increased work at the EJB development end: to define Transfer Objects and supply `getData()` and `setData()` methods, for example.

5. ☑ **D** is the correct answer. Model View Controller helps separate presentation and business concerns in a design sense, and this can support the organization of projects in the way described.

☒ **A** is incorrect—Service Locator is a refinement of the Java development side. **B** is incorrect—there is a pattern called View Helper, but it's a refinement of Model View Controller, so it's not central to the issue at hand. The same goes for answer **E**, Front Controller. **C** is incorrect—Intercepting Filter is independent of separating view and model development management. Finally, **F** is wrong—there is no such thing as the View Controller pattern.

6. ☑ **B** and **C**. The Service Locator pattern is often used in conjunction with Business Delegate, to take from it the responsibility of encapsulating naming service code.
   ☒ **A** is incorrect—wrapping business API calls is actually the responsibility of the Business Delegate pattern. **D** is incorrect, for there is nothing particular about Service Locator regarding web services. There is potential for using a Service Locator to front the finding of a web service, but it's not a benefit that can be described as being "usually conferred" by the pattern. Finally, **E** is incorrect—again, it's Business Delegate that may cache business data. Service Locator can cache things—usually references to remote objects that don't then have to be looked up again through the naming service.

7. ☑ **E** is correct. An Intercepting Filter could be used to check the remote host and block the request before it gets any farther into the application.
   ☒ **A** is incorrect, though certainly possible—a Front Controller could perform this task. However, for this particular sort of blanket authentication, I would judge that Intercepting Filter is a better choice—and the question wanted the best pattern. **C** and **D** are incorrect, for the names are made up. **B** and **F** are correct pattern names, but not right at all for this task.

8. ☑ **E** and **F**. Model View Controller is all about separation of concerns: different components dedicated to different aspects. Though a more contentious claim, Model View Controller can also lead to better project management, for different specialists (graphics programmers, business API experts) can concentrate on their own areas.
   ☒ **A** is incorrect—Business Delegate leads to insulation from business API volatility. **B** is incorrect—MVC has nothing to say about the amount of EJB code. **C** is incorrect—MVC doesn't dictate how information is represented graphically. It might improve as a result of separating concerns, but that's a by-product—not a stated benefit of the pattern. **D** is incorrect, for MVC might actually increase code complexity because it is a more complex framework to work within than simple JSPs accessing data with JSTL SQL custom actions, for example.

9. ☑ **B** is correct. Front Controller is used to dispatch to views in general—so it is the best choice for redirecting to an error page.
   ☒ **A** is incorrect—although Business Delegate might return an application-friendly exception from one of its methods, it shouldn't be used to dispatch to a view. **C** is incorrect—Model View Controller is not specific enough (it's the controller part we are interested in only). **D** and **E** are incorrect, for they are not valid pattern names.

10. ☑ **A** and **F** are the correct answers. The primary reason for using the Business Delegate pattern is to bring about presentation code stability. Presentation code is insulated from direct use of business APIs. Also, Business Delegate often brings about better network performance by caching data and avoiding calls to remote business objects if they don't need to be repeated.
☒ **B** is incorrect, for JNDI code is typically reduced by adopting the Service Locator pattern. **C** is incorrect, for Business Delegate introduces another layer into your code, so it increases layering. **D** is incorrect—nothing can improve business models except better analysis! **E** is incorrect, for Business Delegates—although they are liable to use EJBs—don't have any necessary effect on the way EJBs are coded.

11. ☑ **C** is the correct answer. Transfer Object could be used to collate the individual attributes on the EJB, so only one networked method call would be required to return all the attributes in one go.
☒ **A** is incorrect: Front Controller is used to rationalize navigation. **B** is incorrect—Business Delegate is used to encapsulate (and minimize) EJB method calls. **D** is incorrect—Service Locator is used to encapsulate (and minimize) JNDI calls.

12. ☑ **D**, **E**, and **F** are the correct answers. The Front Controller pattern does act as a gateway for requests, and it should be used to centralize control of navigation. It is also likely to reduce the complexity of links in JSPs; those links should all point to the Front Controller (preferably through an attribute set flexibly by the Front Controller itself).
☒ **A** is incorrect because the Front Controller pattern doesn't insist that you have to have JSPs for the view element: applets, Swing clients, or even other applications are valid view clients for the Front Controller. **B** is incorrect, for Front Controller makes no claim to reduce network traffic. **C** is incorrect—although Front Controller may become a single point of failure, this is hardly a benefit conferred by the pattern; it's more of a drawback.

13. ☑ **E** is the correct answer. None of the patterns listed will unequivocally decrease the number of lines of code in an application. Some may actually increase the lines of code (even though the code is likely to be in a much better organized state).
☒ **A** is incorrect—Front Controller may initially increase the number of components and lines of code. It could reduce navigation code where this is embedded in many different places, but not necessarily. **B** is incorrect—Front View doesn't exist. **C** is incorrect—again, Model View Controller may increase code, especially when first introduced. **D** is incorrect, for Intercepting Filter may just fulfill an additional requirement in the application. It may reduce code where it takes over from code that has been copied and pasted into many separate points of entry into an application.

14. ☑ **E** is the correct answer. All the applications listed in answers **A** through to **D** are reasonable applications for the Intercepting Filter pattern. Re-encoding and unzipping tasks

are the classic province of filters. It is reasonable to use a filter to perform some form of authentication. Equally, you might use a filter to invalidate a session—perhaps as the result of an authentication or authorization check.

☒ **A**, **B**, **C**, **D**, **F**, and **G** are incorrect answers based on the reasoning in the correct answer.

15. ☑ **D** is the correct answer—all of the above. Because the question (rephrased from question 13) is which patterns *might* reduce code, then all the patterns listed could do this. Business Delegate can ultimately reduce code by locating EJB method calls in one place. Because an EJB method call will have many lines of code just for exception handling, these multiply when distributed across all the presentation code that might make EJB method calls. Service Locator can reduce code in the same way, mainly for JNDI method calls. Finally, Intercepting Filter (as noted in question 13 as well) may reduce code by locating some functionality in one place that might otherwise be copied into (for example) all the servlets in an application.

☒ **A**, **B**, and **C** are incorrect answers according to the reasoning in the correct answer.

LAB ANSWER

The lab answer is in the CD at sourcecode/ch10/lab10.war. This is the example application that was used throughout the chapter—detailed instructions on its deployment can be found at the beginning of the chapter.