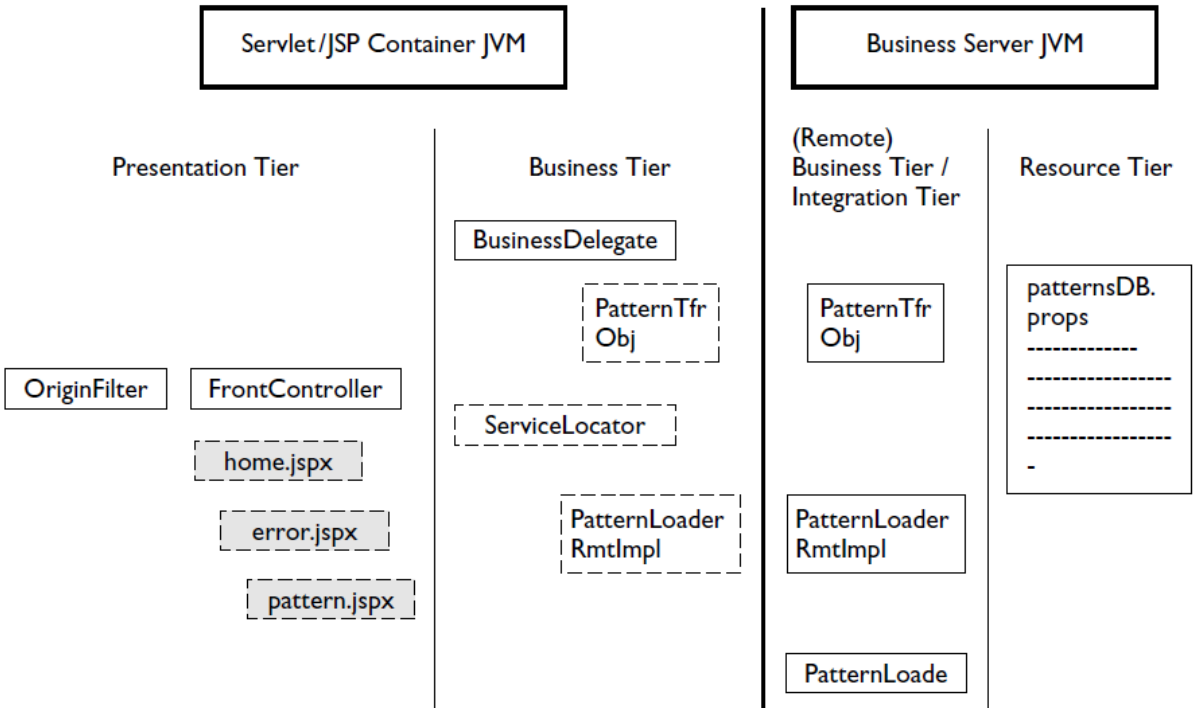
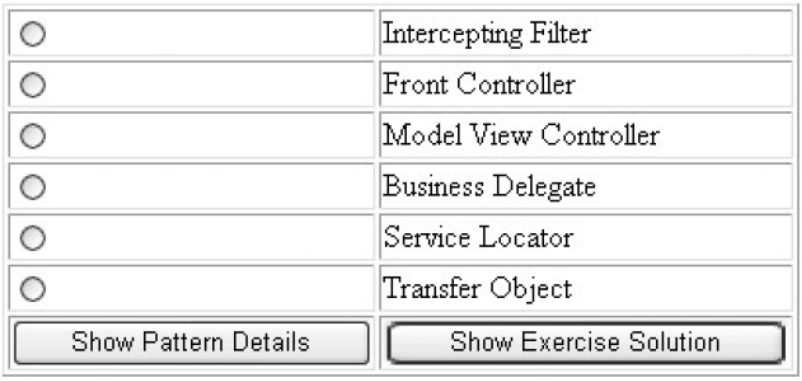
A Working Example for Design Patterns for SCWCD

The idea of the example application is to display information about the six patterns for SCWCD. There are no Enterprise JavaBeans within the example. So instead we implemented the integration tier of the example using Java Remote Method Invocation (RMI). Figure below shows how the application fits together and indicates which classes embody J2EE patterns.



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.



On selecting a pattern, you see a detail screen with information about the pattern:

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)

Let's consider how the application works. 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 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.

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.

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 fi de class implementing the Filter interface and declared as a fi lter 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 specifi c 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 fi nd 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.

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. Deploy and access the home page with a URL such as the following:

http://localhost:8080/lab10/controller

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. 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.

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.

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.

5. When you have finished with the application, just abort the two command windows.

Intercepting Filter Pattern

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).

Front Controller Pattern

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):

```
01 public class FrontController extends HttpServlet {
02 public static final String[] patternNames = { "interceptingFilter",
03 "frontController", "modelViewController", "businessDelegate",
04 "serviceLocator", "transferObject" };
05 private BusinessDelegate businessDelegate = new BusinessDelegate();
06 protected void doGet(HttpServletRequest request,
```

```
07 HttpServletResponse response) throws ServletException, IOException {
08 // Sort out link back to controller servlet
09 String actionLink = request.getContextPath() + request.getServletPath();
10 request.setAttribute("actionLink", actionLink);
11 // Sort out the action required
12 String patternName = request.getParameter("patternChoice");
13 String page;
14 if (isPatternNameRecognized(patternName)) {
15 try {
16 setPatternAttributes(patternName, request);
17 page = "/pattern.jspx";
18 } catch (PatternNotFoundException pnfe) {
19 // Application error handling: masks remote error
20 request.setAttribute("patternName", pnfe.getPatternName());
21 page = "/error.jspx";
22 }
23 } else {
24 page = "/home.jspx";
25 }
26 // 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 are known internally to the application.

In line 05, FrontController declares an instance of a class called **BusinessDelegate**. 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 or 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. 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. 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>

...
```

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. The parameter value for the pattern name comes from a value on a radio button.

Model View Controller

The example application separates out model, view, and controller in the following way:

- 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. It uses logic to direct to the different JSP documents, and it populates those request attributes required by the views.
- The model is the BusinessDelegate class, and everything behind it.

Business Delegate Pattern

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):

```
01 public class BusinessDelegate {

02 Map patternCache = new HashMap();

03 ServiceLocator serviceLocator = new ServiceLocator();

04 PatternLoaderRmtI patternLoader;

05 public PatternTfrObj findPattern(String patternName)

06 throws PatternNotFoundException {

07 // Is the pattern already in the cache?

08 PatternTfrObj pattern = findCachedPattern(patternName);

09 // Not in cache: use remote object

10 if (pattern == null) {

11 if (patternLoader == null) {
```

```
12 patternLoader = serviceLocator.findPatternLoader();
13 }
14 try {
15 pattern = findRemotePattern(patternName);
16 } catch (RemoteException re) {
17 // Print the stack trace for internal diagnosis
18 re.printStackTrace();
19 // Re-throw an "application" exception
20 throw new PatternNotFoundException(patternName);
21 }
22 patternCache.put(patternName, pattern);
23 }
24 return pattern;
25 }
26 protected PatternTfrObj findCachedPattern(String patternName) {
27 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 fi nd 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 later.

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.

Service Locator Pattern

Here's the class (minus package and import statements):

```
01 public class ServiceLocator {

02 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. It makes the point that the reference-finding process can be a lengthy one. 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. When you come to writing classes that deal not with simple RMI objects but with bigger business components—such as EJBs—the Service Locator pattern becomes great help.

Transfer Object Pattern

The example application doesn't use actual EJBs, but you still get the full Transfer Object pattern rendered. 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.

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.