

Proven Solutions to Intractable Problems



J2EE™
Design
Patterns

O'REILLY®

William C. R. Crawford & Jonathan Kaplan

J2EE Antipatterns

The design patterns we have discussed so far are about learning from what others have done correctly. But often, studying others' mistakes is even more valuable. Skiers, watching the trail from the chairlift above, might point out someone doing a particularly good job getting down the slope. But they always discuss exactly who took a spectacular wipeout and what the hapless victim did to bring it upon themselves. Did he turn a little too fast or put his weight too far back? *Antipatterns* are to patterns what the falling skier is to the successful one: recurring, sometimes spectacular mistakes that developers make when faced with a common problem.

In this chapter, we present a few of the most common antipatterns in the J2EE world. The list is by no means complete. Just like the active community collecting design patterns, there is an equally active community cataloguing antipatterns and their solutions. For a general introduction, the text by William J. Brown, et al, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis* (Wiley & Sons), is *the* book on antipatterns; it has a decidedly process-based focus. The current seminal work on Java enterprise antipatterns, which we've relied on heavily, is Bruce A. Tate's excellent resource, *Bitter Java* (Manning).

We will look at the following antipatterns:

Excessive Layering and Leak Collection

Cover repeated architectural errors that affect performance and extensibility.

Magic Servlet, Monolithic JSP, Compound JSP, and Overstuffed Session

Cover breakdowns in Model-View-Controller separation.

Everything Is an EJB, Round-Tripping, and Stateful When Stateless Will Do

Help determine how and how not to use EJBs.

Some of these antipatterns are closely related to the regular patterns we've discussed elsewhere in this book. Since antipatterns are recurring mistakes, and design patterns are recurring solutions, it make sense that most antipatterns have a corresponding pattern or two.

Causes of Antipatterns

Every program, big or small, contains shortcuts, mistakes, and code that could have been thought out a little better. What distinguishes an antipattern from these errors is that, like a design pattern, it is repeated. When we explore the causes of antipatterns, we don't need to look at why they exist—we need to look at why they persist and propagate. The main reasons that antipatterns spread are:

- Inexperience
- Unreadable code
- Cut-and-paste development

Inexperienced developers are the major cause of antipatterns. Obviously, newer developers are less likely to have come across these common mistakes previously, and are less likely to recognize when they are making them. The danger of antipatterns is subtler when experienced developers use new technology. This danger is especially pronounced in J2EE technology, which evolves quickly, and in which many of the standard tutorials and code generation tools are full of antipatterns. To combat the problem, most organizations use training as well as code reviews with senior developers. But the best defense against antipatterns is to know your enemy: understanding and recognizing antipatterns is the key to avoiding them.

Unreadable code is another fertile breeding ground for antipatterns. Often, the existence of a well-known antipattern will be hidden because readers have to spend all their mental cycles trying to figure out what the code does, not why it does it. Developers sometimes favor conciseness and optimization over readability without taking into account the maintenance costs. Every shortcut that saves a few keystrokes or a few bytes of memory should be balanced against the costs in terms of developer hours and lost business when a hard-to-find bug is discovered. Unreadable code is best fought with consistency: publishing and enforcing code guidelines, relentless commenting, and aggressive review.

Cut-and-paste development refers to code that is taken from one place and pasted directly into another. Teams sometimes assume that cut-and-paste is a form of reuse: because the code has been used before, it is often considered more robust. Unfortunately, cut-and-paste actually makes code less reliable, since the pasted portion is used out of context. Even worse, bug changes don't propagate: changes to cut-and-paste code must be propagated manually. Cut-and-paste should only be necessary when entire objects cannot be reused. If you find yourself copying a particular piece of code, consider abstracting it and making it available to many classes through inheritance or as part of a separate utility class.

Architectural Antipatterns

The first class of antipatterns we will look at is architectural in nature. These antipatterns are not J2EE-specific: they affect many Java applications and the Java APIs themselves. They are included in a book on J2EE because they affect highly scalable, long-running applications—meaning they are of particular relevance to J2EE developers.

Excessive Layering

If you’ve read the preceding chapters, you’ve probably noticed that design patterns tend to suggest adding *layers*. Façades, caches, controllers, and commands all add flexibility and even improve performance, but also require more layers.

Figure 12-1 shows a rough sketch of the “typical” J2EE application. Even in this simplified view, each request requires processing by at least seven different layers. And this picture doesn’t even show the details of each individual layer, which may themselves contain multiple objects.

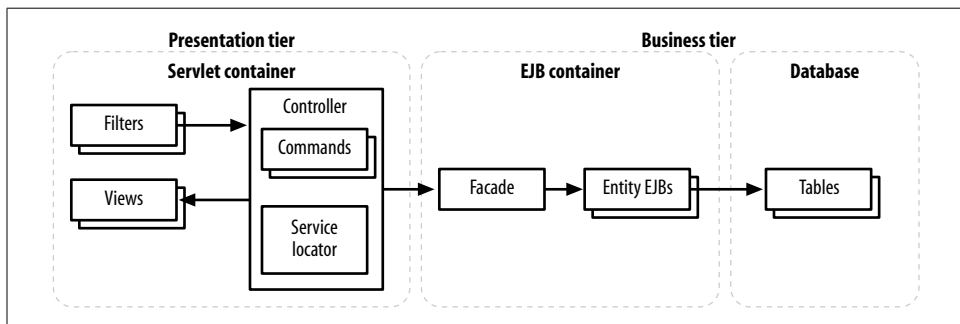


Figure 12-1. A standard J2EE application

Unfortunately, in a high-level, object-oriented environment like J2EE, the layers we add are not the only layers that exist. We’ve already talked about containers—the servlet and EJB containers are the main ones—and how they provide advanced services, generally through layers of their own. The underlying containers are themselves built on top of Java’s APIs; the APIs are still a few layers away from the JVM, which interact through layered system libraries with the operating system—which finally talks to the actual hardware. If you look at stack traces from a running JVM, it is not surprising to see a Java application with a call stack well over 100 methods deep!

It’s easy to see how a CPU that can execute billions of instructions a second can get bogged down running the J2EE environment. It’s also easy to think that with all these layers, adding a few of our own can’t possibly make any difference. That’s not the case, however. Think of the structure as a pyramid: every call to a container method requires two or four calls to the underlying Java API, which requires eight

Java instructions, and so forth and so on. The layers we add are far more expensive than many of the preexisting layers.

An example of the *Excessive Layering antipattern* is a common scenario that we call the “Persistence Layer of Doom.” While abstracting database access away from business logic has so many benefits that we hesitate to say anything negative about the process, hiding SQL from other components has one serious problem: expensive activities (such as accessing a network or filesystem for a database query) start to look like cheap activities (such as reading a field from a JavaBean). Developers working on the presentation tier will inevitably call the expensive functions frequently, and end up assuming the entire business tier is horribly slow. We’ll talk more about this problem later in this chapter when we discuss the Round-Tripping antipattern.

Reducing layers

Because it’s easy to add layers to a J2EE application, it’s important to understand which ones are necessary and which are excessive. Unfortunately, there’s no generic answer, since the correct number of layers depends on the type and expected use of an application.

When deciding whether to add layers, we have to balance the costs with the benefits they provide. The cost of layers can be expressed in terms of design time, code complexity, and speed. The benefits are twofold. Layers that provide abstract interfaces to more specific code allow cleaner, more extensible code. Layers such as caches, which optimize data access, can often benefit an application’s scalability.

While we can’t provide a generic solution to layering problems, we can offer a few hints as to what level of layering is appropriate for generic types of J2EE applications:

All-in-one application

For small applications where the entire model, view, and controller always live on the same server (like a single-function intranet application), reduce layers as much as possible. Generally, this will mean condensing the business tier, often using DAO and business delegate objects that interact directly with an underlying database through JDBC. In the presentation tier, you should stick to a simple servlet/JSP model.

Front end

Often, medium-size applications provide a simple web or similar frontend to a shared data model, usually a legacy application. Examples include airline ticketing systems and online inventory tools. In these applications, the presentation tier is generally the focus of development. The presentation tier should scale across multiple servers, and should be made efficient and extensible with liberal use of layering. Most functions of the business tier will probably be supported by the underlying legacy application and should not be duplicated in a large, deeply layered business tier.

Internet scale application

The last type of application is the ultimate in J2EE: a large application spread over numerous servers meant to handle thousands of users and millions of requests per day. In these large environments, communication overhead between the many servers, as well as the cost of maintaining a large code base, dwarf the cost of layering. Using all the standard layers and multiple layers of caching in both the presentation and business tier can help optimize network transactions, while layers of abstraction keep the code manageable and extensible.

Communication and documentation are our best weapons. When layers are added for performance, document which calls are expensive, and, where possible, provide alternative methods that batch requests together or provide timeouts. When layers are added for abstraction, document what the layer abstracts and why, especially when using vendor-specific methods (see the “Vendor Lock-In” sidebar). These steps assure that our layers enhance extensibility or improve performance instead of becoming an expensive black hole.

Leak Collection

Automated memory management is one of Java’s most important features. It is also somewhat of an Achilles’s heel. While a developer is free to create objects at will, she does not control when or how the garbage collector reclaims them. In some situations, objects that are no longer being used may be kept in memory for much longer than necessary. In a large application, using excess memory in this way is a serious scalability bottleneck.

Fortunately, by taking into account how the garbage collector actually works, we can recognize common mistakes that cause extra objects. The Java Virtual Machine uses the concept of *reachability* to determine when an object can be garbage-collected. Each time an object stores a reference to another object, with code like `this.intValue = new Integer(7)`, the referent (the Integer) is said to be reachable from the object referring to it (this). We can manually break the reference, for example by assigning `this.intValue = null`.

To determine which objects can be garbage-collected, the JVM periodically builds a graph of all the objects in the application. It does this by recursively walking from a root node to all the reachable objects, marking each one. When the walk is done, all unmarked objects can be cleaned up. This two-phase process is called *mark and sweep*. If you think of references as strings that attach two objects together, the mark and sweep process is roughly analogous to picking up and shaking the main object in an application. Since every object that is in use will be attached somehow, they will all be lifted up together in a giant, messy ball. The objects ready for garbage collection will fall to the floor, where they can be swept away.

Vendor Lock-In

Many J2EE vendors offer enhancements to the core J2EE functionality, such as optimized database access methods or APIs for fine-grained control of clustering capabilities. Using these functions, however, ties your application to that vendor's implementation, whether it's a database, MOM, or application server. The more your application depends on a particular vendor's APIs, the harder it is for you to change vendors, effectively locking you into the vendor you started with.

J2EE purists will tell you why vendor lock-in is a bad thing. If your vendor decides to raise their prices, you are generally stuck paying what they ask or rebuilding your application. If you sell software to a customer, they too must buy your vendor's product, regardless of their own preference. And if the vendor goes out of business, you could be stuck with unsupported technology.

From a practical standpoint, however, vendor's enhancements are often just that: enhancements. Using vendor-specific APIs can often make your application easier to build, more efficient, and more robust. So is there a happy middle ground? There is. While using vendor-specific APIs is not an antipattern, vendor lock-in is.

The most important step in avoiding lock-in is understanding which APIs are generic and which are vendor-specific. At the minimum, clearly document all vendor dependencies. Also, make your best effort to avoid letting the structure of the API influence overall design too much, particularly if you think you might have to eventually abandon the vendor.

A better solution is to hide the vendor complexities by defining an interface with an abstract definition of the vendor's methods and then implementing that interface for the particular vendor you have chosen. If you need to support a new vendor, you should be able to simply reimplement the interface using the new vendor's methods or generic ones if necessary.

So how does this knowledge help us avoid memory leaks? A memory leak occurs when a string attaches an object that is no longer in use to an object that is still in use. This connection wouldn't be so bad, except that the misattached object could itself be connected to a whole hairball of objects that should otherwise be discarded. Usually, the culprit is a long-lived object with a reference to a shorter-lived object, a common case when using collections.

A collection is an object that does nothing more than organize references to other objects. The collection itself (a cache, for example) usually has a long lifespan, but the objects it refers to (the contents of the cache) do not. If items are not removed from the cache when they are no longer needed, a memory leak will result. This type of memory leak in a collection is an instance of the *Leak Collection antipattern*.

In Chapter 5, we saw several instances of caches, including the Caching Filter pattern. Unfortunately, a cache with no policy for expiring data constitutes a memory leak. Consider Example 12-1, a simplified version of our caching filter.

Example 12-1. A simplified CacheFilter

```
public class CacheFilter implements Filter {
    // a very simple cache
    private Map cache;

    public void doFilter(ServletRequest request,
        ServletResponse response, FilterChain chain)
        throws IOException, ServletException {
        ...

        if (!cache.containsKey(key)) {
            cache.put(key, data);
        }

        // fulfill the response from the cache
        if (cache.containsKey(key)) {
            ...
        }
    }

    public void init(FilterConfig filterConfig) {
        ...
        cache = new HashMap();
    }
}
```

Nowhere in this code is there a `remove()` call to match the `put()` call that adds data to the cache. Without a cache expiration policy, the data in this cache will potentially use all the available memory, killing the application.

Reclaiming lost memory

The hardest part of dealing with Java memory leaks is discovering them in the first place. Since the JVM only collects garbage periodically, watching the size of the application in memory isn't very reliable. Obviously, if you're seeing frequent out-of-memory type errors, it's probably too late. Often, commercial profiling tools are your best bet to help keep an eye on the number of objects in use at any one time.

In our trivial example, it should be clear that adding data to the cache and never removing it is a potential memory leak. The obvious solution is to add a simple timer that cleans out the cache at some periodic interval. While this may be effective, it is not a guarantee: if too much data is cached in too short a time, we could still have memory problems. A better solution is to use a Java feature called a *soft reference*, which maintains a reference to an object but allows the cached data to be garbage-collected at the collector's discretion. Typically, the least-recently used objects are

collected when the system is running out of memory. Simply changing the way we put data in the cache will accomplish this:

```
cache.put(key, new SoftReference(data));
```

There are a number of caveats, the most important being that when we retrieve data, we have to manually follow the soft reference (which might return null if the object has been garbage-collected):

```
if (cache.containsKey(key)) {
    SoftReference ref = (SoftReference) cache.get(key);
    Object result = ref.get();

    if (result == null) {
        cache.remove(key);
    }
}
```

Of course, we could still run out of memory if we add too many keys to the cache. A more robust solution uses a reference queue and a thread to automatically remove entries as they are garbage-collected.

In general, the most effective way to fight memory leaks is to recognize where they are likely to be. Collections, as we have mentioned, are a frequent source of leaks. Many common features—such as attribute lists and listeners—use collections internally. When using these features, pay extra attention to when objects are added and removed from the collection. Often, it is good practice to code the removal at the same time as the addition. And, of course, make sure to document pairs of adds and removes so that other developers can easily figure out what you did.

Presentation Tier Antipatterns

The Model-View-Controller pattern, covered in Chapter 3, is the fundamental organizing principle for the presentation tier. The building blocks of the MVC pattern create an overall map of the application, making it easier to understand and extend. Preserving the separation of model, view, and controller is essential to maintaining these advantages.

It should be no surprise, then, that the presentation tier antipatterns have to do with the breakdown of model-view-controller separation. While these antipatterns are specific to the presentation tier, be aware that other antipatterns, such as Leak Collection and Excessive Layering, can affect the presentation tier as well.

The Magic Servlet

When servlets were first introduced, developers immediately saw the potential of combining the robust Java environment with an efficient mechanism for serving dynamic content. Server-side Java's killer app was JDBC, a powerful and high-level

mechanism for communicating with databases. Over time, technologies like JNDI and JMS were added, allowing J2EE to talk easily and directly to a large number of enterprise information systems.

Because it was suddenly so easy to talk to databases, directory servers, and messaging systems, many developers were tricked into thinking that their applications would be simple, too. Who needed a complicated design when reading a row from a database was a one-line operation? Unfortunately, as applications grew in scope, complexity crept back in—but the design to handle it did not.

The typical symptom of complexity outpacing design is the *Magic Servlet antipattern*, in which a single servlet handles all aspects of a given request. On the surface, the magic servlet seems like a reasonable encapsulation. It captures all the logic needed to handle a request in a single, convenient class. But a magic servlet is also large, complex, difficult to maintain, and impossible to reuse. Typically, these servlets have a huge amount of code in a single `doGet()` or similar method.

Imagine you are given the task of putting an LDAP-based corporate address book on the web. Example 12-2 shows a typical magic servlet solution, using the servlet to read all the names and phone numbers in the LDAP directory.

Example 12-2. A magic servlet

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.naming.*;
import javax.naming.directory.*;
import java.util.*;

public class MagicServlet extends HttpServlet {
    private DirContext peopleContext;

    // setup LDAP access
    public void init(ServletConfig config)
        throws ServletException {
        super.init(config);

        Properties env = new Properties();
        env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
        env.put(Context.PROVIDER_URL, "ldap://localhost/o=jndiTest");
        env.put(Context.SECURITY_PRINCIPAL, "cn=Manager, o=jndiTest");
        env.put(Context.SECURITY_CREDENTIALS, "secret");

        try {
            DirContext initialContext = new InitialDirContext(env);
            peopleContext = (DirContext)initialContext.lookup("ou=people");
        } catch (NamingException ne) {
            ne.printStackTrace();
            throw new UnavailableException("Error inializing LDAP", ne);
        }
    }
}
```

Example 12-2. A magic servlet (continued)

```
// close LDAP
public void destroy() {
    try {
        peopleContext.close();
    } catch(NamingException ne) {
        ne.printStackTrace();
    }
}

// "magic" function is model, view and controller
protected void doGet(HttpServletRequest request,
    HttpServletResponse response)
throws ServletException, IOException {
    // view logic
    response.setContentType("text/html");
    java.io.PrintWriter out = response.getWriter();
    out.println("<html>");
    out.println("<head>");
    out.println("<title>Servlet</title>");
    out.println("</head>");
    out.println("<body>");
    out.println("<table>");

    // model logic
    try {
        NamingEnumeration people = peopleContext.list("");
        while(people.hasMore()) {
            NameClassPair personName = (NameClassPair)people.next();
            Attributes personAttrs = peopleContext.getAttributes(personName.getName());
            Attribute cn = personAttrs.get("cn");
            Attribute sn = personAttrs.get("sn");
            Attribute phone = personAttrs.get("telephoneNumber");

            out.println("<tr><td>" + cn.get() + " " +
                sn.get() + "</td>" +
                "<td>" + phone.get() +
                "</td></tr>");
        }
    } catch(Exception ex) {
        out.println("Error " + ex + " getting data!");
    }

    // back to view logic
    out.println("</table>");
    out.println("</body>");
    out.println("</html>");
}
}
```

The `doGet()` method talks directly to the data source, LDAP. It also writes directly to the output. This example is relatively simple, since it doesn't handle input validation

or any kind of error handling, for starters. Adding those functions would make the servlet even more complicated and specialized.

The problem with a magic servlet is not a lack of encapsulation, but rather *improper* encapsulation. The first rule of objects is that they should perform a single function well. The scratch test is whether you can describe what an object does in one phrase. For our magic servlet, the best we could say is “it reads phone numbers from a database and formats the results as HTML.” Two phrases at least, and awkward ones at that.

Refactoring the magic servlet

To fix the Magic Servlet antipattern, we need to break our servlet into multiple objects, each with a specific task. Fortunately, we already have a pattern for solving just this type of problem: the Model-View-Controller. MVC is covered extensively in Chapter 3, so we won’t go into too much detail here. The salient point is the separation of the interaction into (at least) three pieces.

The model is responsible for all interactions with persistent storage. We will create a simple JavaBean, `Person`, with accessor methods for fields `firstName`, `lastName`, and `phoneNumber`. Then we create a class, `LdapPersonCommand`, to read from LDAP and expose the values as a collection of class `Person`. The command implements a generic `PersonCommand` interface:

```
public interface PersonCommand {
    // initialize the command
    public void initialize(HttpSession session) throws NamingException;

    // execute the query
    public void runCommand();

    // get the result of the query as a List of class Person
    public List getPeople();
}
```

The view, a simple JSP page, uses an instance of class `PersonCommand` placed in request scope and the Java Standard Tag Libraries to generate a table of names and phone numbers. `PersonView.jsp` is shown in Example 12-3.

Example 12-3. PersonView.jsp

```
<%@page contentType="text/html"%>
<%@taglib uri="/jstl/core" prefix="c"%>

<html>
<head><title>Addresses</title></head>
<body>
<jsp:useBean id="personCommand" scope="request"
    class="PersonCommand" />
<table>
<c:forEach var="person" items="${personCommand.people}">
    <tr><td><c:out value="${person.firstName}"/></td>
```

Example 12-3. PersonView.jsp (continued)

```
<td><c:out value="\${person.lastName}"/></td>
<td><c:out value="\${person.phoneNumber}"/></td>
</tr>
</c:forEach>
</table>
</body>
</html>
```

As you can see, our JSP is entirely focused on view management. It contains only the minimal amount of JSTL code needed to handle looping and output.

All this abstraction leaves us with a very simple servlet, as shown in Example 12-4.

Example 12-4. PersonServlet.java

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import javax.naming.*;

public class PersonServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        try {
            PersonCommand personCommand = new LdapPersonCommand();
            personCommand.initialize();
            personCommand.runCommand();

            request.setAttribute("personCommand", personCommand);
        } catch (NamingException ne) {
            throw new ServletException("Error executing " + "command", ne);
        }

        RequestDispatcher dispatch =
            getServletContext().getRequestDispatcher("/PersonView.jsp");
        dispatch.forward(request, response);
    }
}
```

This servlet is responsible for coordinating the response but not for performing the dirty work of reading LDAP or generating HTML. The major advantages of this solution are in extensibility. We could easily replace `LdapPersonCommand` with a class that reads from a database or legacy system. Writing a separate JSP to output to WAP or WML or a different language would also be easy and require no modification of the servlet. All of this should be familiar from previous chapters. The point is that the problem can be solved. You can resolve this antipattern a piece at a time. In this case, you might want to isolate the view first, and come back and separate the model and controller when you have time.

Monolithic/Compound JSPs

When JSPs were first introduced, some developers recognized their potential to replace the awkwardness of embedding HTML output in a servlet with the convenience of scripting. Many decided that since JSPs were parsed into servlets internally, they were just a new syntax for servlets. So they built servlets—including access to models, control logic, and output—as JSPs. Example 12-5 shows a worst-case JSP design.

Example 12-5. LoginPage.jsp

```
<%@page contentType="text/html"%>
<%@page import="antipatterns.LoginManager" %>
<html>
<%
    LoginManager lm = new LoginManager();
    ServletRequest req = pageContext.getRequest();
    if (!lm.doLogin(req.getParameter("username"),
        req.getParameter("password"))) {
%>
<head><title>Login Page</title></head>
<body>
    <form action="/LoginPage.jsp" method="post">
        User name: <input type="text" name="username"><br>
        Password: <input type="password" name="password"><br>
        <input type="submit">
    </form>
</body>
<% } else { %>
<head><title>First Page</title></head>
<body>
    Welcome to the page!
<% } %>
</body>
</html>
```

This example has the same problems as a magic servlet. It performs operations belonging to the model, view, and controller. The example above combines the functionality of JSP and servlets, but represents the worst of both worlds from the point of view of maintenance. The HTML is confusing, often duplicated, and spread throughout the file in an opaque way. The Java code is embedded awkwardly—again, hard to find and follow.

There are actually two common antipatterns in Example 12-5. The *Monolithic JSP antipattern* occurs when JSP pages do more than their fair share of the work, often acting as both view and controller. While this example doesn't interact directly with the model via JNDI or JDBC, it would be a logical extension, further destroying the separation of model and controller.

A closely related antipattern is the *Compound JSP antipattern*, in which multiple JSP files are crammed into one using Java conditionals. In the example above, our code tests the return value of a call to `doLogin()` and displays different pages based on the result.

Fixing monolithic and compound JSPs

JSP's advantage is its simplicity. As a tag-based language, it should be familiar to web designers who no longer need to understand the complexities of Java development in order to build web pages. It is also meant to integrate easily into existing tools, such as WYSIWYG HTML editors.

Monolithic and compound JSPs take away those advantages. It would take a pretty sophisticated editor to separate the two pages contained in Example 12-5. And web designers working with it would have to understand the embedded code, at least enough so as not to break it.

Since monolithic JSPs are really just another type of magic servlet, the solution is pretty much the same. The JSP should be refactored into three separate pieces, with a servlet acting as the controller, the JSP as the view and JavaBeans as the model. We won't go through the whole exercise of converting our monolithic JSP to MVC, since the result is so similar to the previous example and those in Chapter 3.

Compound JSPs present a slightly different problem, but again a similar solution. In Example 12-5 we effectively have two pages: the login page and the welcome page. We should separate these into two separate JSP pages (in this case, HTML pages would work as well). The selection logic should go in the controller servlet, which contains code like:

```
if (lm.doLogin(username, password)) {
    nextPage = "/welcome.jsp";
} else {
    nextPage = "/login.jsp";
}

RequestDispatcher dispatch =
    getServletContext().getRequestDispatcher(nextPage);
dispatch.forward(request, response);
```

Overstuffed Session

The session, which tracks users as they navigate a web site, is a major feature of the J2EE presentation tier. Objects like the user's security permissions can be stored in the session when a user first logs in, and used several pages later to determine what the user can access. While maintaining session information is quite convenient, it is not without its pitfalls. Putting too much data or the wrong kind of data into the session leads to the *Overstuffed Session antipattern*.

The first danger of this antipattern is from data with a short lifespan. Since the session is implemented as a collection, we have to be on the lookout for a variant of the Leak Collection antipattern. Putting objects in the wrong scope, as we have done in Example 12-6, can lead to a pseudo memory leak.

Example 12-6. The LeakyServlet's doGet() method

```
protected void doGet(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
    HttpSession session = request.getSession();

    // create the bean
    AddressBean address = new AddressBean();
    address.setFirst(request.getParameter("first"));
    address.setLast(request.getParameter("last"));

    // pass the bean to the view
    session.setAttribute("antipatterns.address", address);

    // instantiate the view
    RequestDispatcher dispatcher =
        getServletContext().getRequestDispatcher("/View.jsp");
    dispatcher.forward(request, response);
}
```

As the snippet shows, we have placed a bean called `AddressBean` (which contains state data that is only used in this request) in session scope. This bean, however, will be kept around in the user's session for as long as the user is connected.

It's hard to see this kind of scenario as a real memory leak. The user's session expires when they log out, and the memory is reclaimed. But each object adds up, and if lots of users store lots of irrelevant beans in session scope, the total number of users that can connect at any one time will be reduced.*

The second danger is from data with a long lifespan. It is very common to store complex state data—such as shopping carts or user preferences—in the user's session, assuming the session will stay alive for as long as the user is using the site. Unfortunately, this is not always the case: if the user takes a break to respond to email or read another web page, or if the web server crashes, the user may unexpectedly lose all the data. This can be a real problem: just ask anyone who has painstakingly selected all the components for his new computer system only to have them disappear while he is looking for his credit card.

* If you use Struts or a similar web application framework, pay particular attention to this issue. The default Form Bean scoping in Struts is to the session, rather than to the request, so it's easy to end up with a large number of extra objects in memory for each active user.

Unstuffing the session

Fortunately, as far as antipatterns go, fixing an overstuffed session isn't too difficult. For short-lived data, make sure you default to request scope for all your objects, and only use session scope when it is explicitly required. Also, remember that it is possible to remove items from the session if you know they will no longer be used.

Long-lived data should generally be migrated to the business tier. This precaution has a number of advantages, including persistence between logins and across server restarts and transactions. Recently, there has been a trend toward applications with a database solely used by the presentation tier. While it may be overkill, this method provides a convenient (and usually high-speed) place to store user and session information without affecting the design of the business tier.

EJB Antipatterns

Our final set of antipatterns deal with Enterprise JavaBeans. EJBs are a powerful technology, but can also be complicated and heavyweight. Two of our antipatterns deal with the complexity of EJBs: the *Everything Is an EJB antipattern* describes when EJBs are appropriate to use at all, while the *Stateful When Stateless Will Do antipattern* describes when stateful session EJBs should be used. The *Round-Tripping antipattern* covers common performance problems in client-server applications, and often turns up when you're using remote EJBs.

Everything Is an EJB

There is a common antipattern called the *golden hammer*. A golden hammer starts life as a good solution to a recurring problem—the kind of solution that design patterns are made from. Eventually, however, the golden hammer starts getting used because it is the most familiar to developers, not because it's the best solution. Like a carpenter with only one saw, the golden hammer may get the job done, but it doesn't give the cleanest or easiest cut.

In many cases, EJBs are just such a golden hammer. Developers—especially developers with a history of database development—tend to see entity EJBs as the solution to every problem. Need security? Create a username and password bean. Need an address? Create an address bean.

Unfortunately, EJBs are not the solution to every problem. Like any other technology (and EJBs are a complex technology, at that), EJBs have both costs and benefits. EJBs should only be used when *their benefits outweigh the costs in solving the problem at hand*. This is an important concept, so let's look at each aspect separately.

The first part of applying this concept is to understand the benefits of EJBs. The central idea behind Enterprise JavaBeans, particularly entity beans, is to create an “abstract persistence mechanism.” EJBs provide a generic, object-oriented way to

manage data without worrying about the details of what's underneath. So we describe the data we want to store, along with transaction characteristics, security mechanisms, and so on, and the EJB container worries about translating it into whatever database or other storage mechanism we happen to use. On top of that, the container makes the beans available remotely, enforces the security constraints, and optimizes access to the beans using pools and other mechanisms. It seems like a pretty good deal.

The other significant aspect of the benefit/cost concept involves understanding that there are also substantial costs to using EJBs. EJBs add significant complexity to an application. EJBs must be created, looked up, and referenced using JNDI, and a dizzying array of home, remote, and local interfaces. While containers use caching and pooling to help, EJBs are often a memory and performance bottleneck. And, of course, you often need to negotiate with the purchasing department to buy the container.

Choosing whether to use EJBs comes down to one key issue: solving the problem at hand. EJBs are a great solution when their benefits line up with the features you need. If you don't need to use an existing database, using container-managed persistence can be quite efficient.* If you need to add transactions to otherwise transactionless storage, or add secure remote access to your data, EJBs are also a good idea. Too frequently, however, EJBs are used when none of these features are needed.

For example, let's go back to our web-based address book application based on an LDAP database. At the under-designed extreme, we have the magic servlet we saw earlier. While the magic servlet is not robust or extensible enough, solving the problem with entity EJBs is overkill. Figure 12-2 shows a sketch of the EJB solution: an address command in the presentation tier uses an address EJB, which in turn communicates with the underlying LDAP directory.

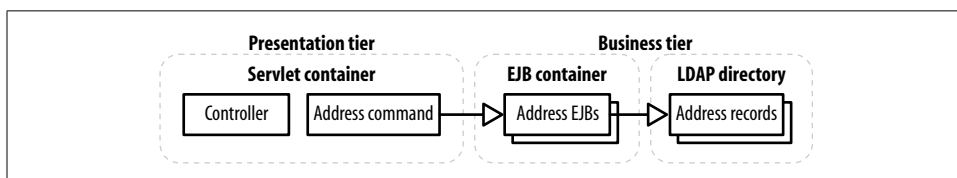


Figure 12-2. A poor use of entity beans

What are the EJBs adding? Chances are, the LDAP directory already provides some of the basic EJB features, such as remote access to the data, security, and maybe even transactions. If the `AddressCommand` is well defined, it will provide a level of abstraction between the controller and the underlying LDAP. Since we're stuck with the

* Contrary to popular belief, CMP can be far more efficient than BMP in recent EJB implementations, due to built-in pooling and caching. Advanced database concepts like views and indexes, however, are still not fully matched with CMP, and mapping existing schemas to CMP can be painful, particularly when there's a lot of data normalization involved.

existing LDAP database anyway, the EJBs are not really protecting us from changes to the underlying storage. In the end, there are no real advantages to using EJBs in this case, and the costs in terms of development and performance outweigh the benefits.

Escaping EJBs

What are our options, if EJBs are out of the picture? If it's entity beans we gave up, usually the answer is just plain JavaBeans. A command interface looks a whole lot like an EJB, anyway: there are usually setter methods to describe the arguments to the command, business methods to perform the actual command, and getter methods to read the results. We simply build a `Command` like we would a BMP entity bean, by communicating directly with a database. Since the fundamental bean structure is preserved, it's usually a simple task to switch back to EJBs if we need the features later.

There are, of course, other alternatives. For instance, storing data in XML files or as serialized objects works for simple applications. In more complex cases, Java Data Objects (JDO) provide a simple way to persist arbitrary Java objects without many of the complexities of EJBs.* A flexible but more complicated option is to use a variation of the Service Locator pattern (described in Chapter 9) in conjunction with façades to hide whether beans are local or remote at all.

On the system logic side, consider turning stateful session beans to stateless session beans (see the section “Stateful When Stateless Will Do”), and evaluate your stateless session beans to determine whether they should be EJBs. If not, think about replacing them with business delegates.

The following criteria can help determine whether you can safely convert your session beans into business delegates:

- Transaction management is locally controlled, such as with a transaction wrapper (see Chapter 10).
- Entity beans are not being used.
- The session bean is just a gateway to an additional remote service (such as a DAO or PAO).
- A small number of clients (such as a presentation tier) will be using the business component.
- Business processes all have to run in the same environment.

If most or all of these conditions are true, there won't necessarily be much of a benefit to using session beans. Transaction management, in particular, is often a simpler problem than it is sometimes portrayed as: EJBs make it easy to distribute transactions across multiple resources, including databases and MOM middleware, but if all

* In fact, some developers have suggested that using JDO in conjunction with session EJBs provides a better architecture than BMP entity EJBs, but we'll leave that debate for a different book.

your transaction management needs are focused on simple database calls (or invoking DAOs), a transaction wrapper is often sufficient.

If your session bean is acting as a session façade (see Chapter 9), look carefully at what it is providing access to. One of the prime advantages of a façade is that you can run a lot of your code much closer to the entity beans it acts on. If you aren't using entity beans, you lose this benefit. The same applies if the session façade spends most of its time dealing with resources that are themselves remote: if all the façade does is forward requests to a database, you might as well have the presentation tier connect to the database directly.

Clustering is frequently cited as a reason for using both session and entity EJBs, but you'll often find in these cases that it's either entirely unnecessary or can be done more cheaply. Pointing multiple instances of a web app at the same data store might be frowned upon, but is often the easiest solution. This is particularly true if the number of total clients is small: you don't need to worry about creating infrastructure that can handle 10,000 connections at once if all you really need is to provide business services to a couple of web servers.

Round-Tripping

Compared to the speed of local execution, using a network is extremely slow. That may sound like the sales-pitch for huge SMP servers, but it's not. Distributed architectures are essential to providing applications that scale to Internet demands. But even within a distributed architecture, performance can be dramatically improved by doing work locally.

The communication between the presentation and business tiers is a common source of performance problems within distributed architectures. Whether it's remote EJBs, directories, or databases, the cost of maintaining and using remote data is easy to lose track of, especially when development takes place on a single machine.

The *Round-Tripping antipattern* is a common misuse of network resources. It occurs when a large amount of data, like the results of a database lookup, needs to be transferred. Instead of sending back one large chunk of data, each individual result is requested and sent individually. The overhead involved can be astonishing. Each call requires at least the following steps:

1. The client makes the request.
2. The server retrieves the data.
3. The server translates the data for sending over the network.
4. The server sends the data.
5. The client translates the data from the network.

Round-tripping occurs when this sequence is repeated separately for each result in a large set.

The Round-Tripping antipattern is most often seen with remote entity EJBs. One of the features of EJBs is that they can be moved to a remote server, more or less transparently.* This power, however, is easy to abuse. Example 12-7 shows a command that reads addresses from a set of entity EJBs and stores the results locally.

Example 12-7. A PersonBean client

```
import java.util.*;
import javax.ejb.*;
import javax.rmi.*;
import javax.naming.*;
import javax.servlet.http.*;

public class EJBPersonCommand implements PersonCommand {
    private List people;
    private EJBPersonHome personHome;

    public void initialize(HttpSession session) throws NamingException {
        InitialContext ic = new InitialContext();
        Object personRef = ic.lookup("ejb/EJBPerson");

        personHome =
            (EJBPersonHome) PortableRemoteObject.narrow(personRef, EJBPersonHome.class);

        people = new Vector();
    }

    // read all entries in the database and store them in a local
    // list
    public void runCommand() throws NamingException {
        try {
            Collection ejbpeople = personHome.findAll();

            for(Iterator i = ejbpeople.iterator(); i.hasNext();) {
                EJBPerson ejbPerson = (EJBPerson)i.next();
                people.add(new Person(ejbPerson.getFirstName(),
                    ejbPerson.getLastName(),
                    ejbPerson.getPhoneNumber()));
            }
        } catch(Exception ex) {
            ...
            return;
        }
    }

    public List getPeople() {
        return people;
    }
}
```

* Whether it's more or less usually depends on which EJB container you are using.

The code looks innocuous enough. The `PersonHome` interface is used to find all people in the database, which are returned as instances of the `EJBPerson` EJB. We then loop through all the people, reading their various attributes and storing them in a local `List`.

The problem is that when this client and the `Person` EJB are not on the same machine, each call to `EJBPerson.getXXX()` requires a call across the network. This requirement means that, in this example, we're making $3n$ round trips, where n is the number of people in the database. For each trip, we incur the costs of data marshaling, the actual transfer, and unmarshaling, at the very least.

Reducing round-tripping

Fortunately, round-tripping is not hard to recognize. If you suddenly find performance problems when you move an application onto multiple servers, or find your intranet saturated, chances are round-tripping is to blame.

To reduce round-tripping, we need to combine multiple requests into one. Our options are to modify the client or modify the server. On the client side, we can implement caches to make sure we only request data once, not hundreds of times. Obviously, this will only benefit us if the data is read more often than it is changed.

A more robust solution is to modify the server, letting it make many local calls before returning data over the network. In the EJB case, this involves two patterns we have already seen, the Data Transfer Object and the Façade. We replace our many remote calls to `EJBPerson.getXXX()` with a single call to a façade, which returns the data in a custom data transfer object. If it sounds complicated, don't worry, it's actually quite simple, as you can see in Figure 12-3.

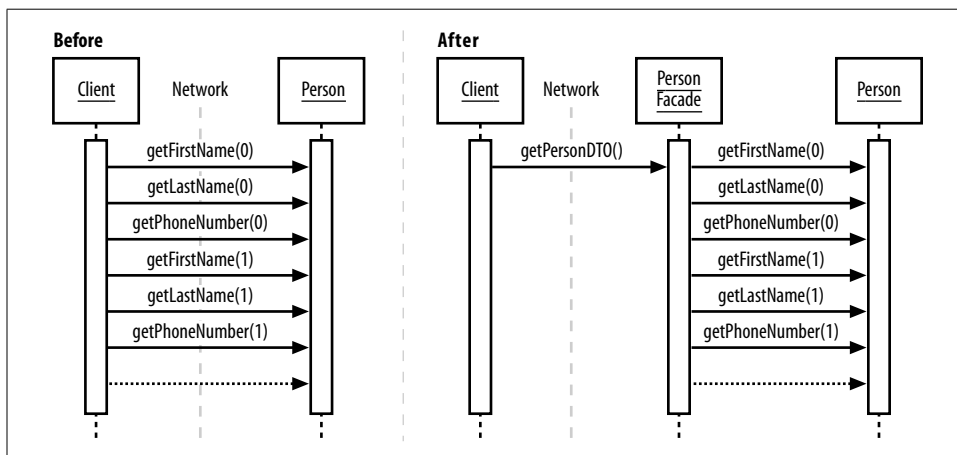


Figure 12-3. Reducing round-tripping

The first step is to define a DTO that encapsulates our data:

```
public class PeopleDTO implements Serializable {
    private List people;

    public PeopleDTO() {
        people = new Vector();
    }

    public List getPeople() {
        return people;
    }

    public void addPerson(Person person) {
        people.add(person);
    }
}
```

The next step is our façade. The façade in this case is a full-fledged, stateless session bean. The business methods of the façade match the finder methods of the original bean. Since we got the list of people in the original bean using the `findAll()` method, we will create a matching `findAll()` method in our session bean, which is shown in Example 12-8.

Example 12-8. A façade for EJBPerson

```
import javax.ejb.*;
import java.util.*;
import javax.naming.*;

public class PersonFacadeBean implements SessionBean {
    private SessionContext context;
    private LocalEJBPersonHome personHome;

    public void setSessionContext(SessionContext aContext) {
        context=aContext;
    }

    public void ejbActivate() {}
    public void ejbPassivate() {}

    public void ejbRemove() {}

    // find the local bean
    public void ejbCreate() {
        try {
            String beanName = "java:comp/env/ejb/local/Person";

            InitialContext ic = new InitialContext();
            personHome = (LocalEJBPersonHome) ic.lookup(beanName);
        } catch (Exception ex) {
            throw new EJBException("Error looking up PersonHome", ex);
        }
    }
}
```

Example 12-8. A façade for EJBPerson (continued)

```
// find all entries and store them in a local DTO
public PeopleDTO findAll()
throws FinderException {
    Collection c = personHome.findAll();
    PeopleDTO dto = new PeopleDTO();

    for (Iterator i = people.iterator(); i.hasNext();) {
        LocalEJBPerson ejbPerson = (LocalEJBPerson)i.next();

        dto.addPerson(new Person(ejbPerson.getFirstName(),
            ejbPerson.getLastName(),
            ejbPerson.getPhoneNumber()));
    }

    return dto;
}
}
```

The session bean basically performs the same loop as we did in our original client. Notice how the session bean uses the LocalEJBPerson interface instead of the EJBPerson interface. The local interface is a feature of EJB 2.0 that allows far more efficient operation for EJBs that are known to be in the same container. Using the LocalEJBPerson interface guarantees that round-tripping will not occur while we build the DTO.

The final step is to replace the original client with one using the DTO. As with the other steps, this is quite straightforward. We just replace the previous loop with a call to the façade:

```
public void runCommand() throws NamingException {
    try {
        PersonFacade facade = personFacadeHome.create();
        PeopleDTO peopleDto = facade.findAll();
        people = peopleDto.getPeople();
    } catch (Exception ex) {
        ex.printStackTrace();
        ...
    }
}
```

This method traverses the network just once, no matter how big the database is. The performance improvements from reduced round-tripping can be substantial. In one unscientific test of the example above, the time to transfer 1,000 addresses was reduced from over 4 minutes to 14 seconds.

In addition to the DTO pattern, you should also consider using a *data transfer row set* (see Chapter 7) to address these issues.

Stateful When Stateless Will Do

It is a common misconception that stateful and stateless session EJBs are basically the same thing. It makes sense: they're both types of session EJBs. As their name implies, stateful EJBs maintain a conversational state with clients, like a normal Java object, while stateless beans must be given all their state data each time they are called.

The major difference between stateful and stateless beans, however, is how they are managed by the container. A stateless bean is relatively simple to manage. Since operations on the bean do not change the bean itself, the container can create as many or as few beans as it needs. All the copies of a stateless bean are essentially equal. Not so with stateful beans. Every time a client makes a request, it must contact the same bean. That means that every client gets its own bean, which must be kept in memory somehow, whether the client is using it or not. The necessary management and storage makes a stateful session bean far more expensive for the container to manage than a stateless one.

Since stateful EJBs work more like normal objects, it's a common mistake to use them when stateless beans could be used to achieve the same effect at a much lower cost. For example, we could build an `AddressBookEntry` entity EJB with local home interface:

```
public interface AddressBookEntryHome extends EJBLocalHome {

    // required method
    public AddressBookEntry
        findByPrimaryKey(AddressBookEntryKey aKey)
        throws FinderException;

    // find all entries in owner's address book
    public Collection findAll(String owner)
        throws FinderException;

    // add a new entry to owner's address book
    public AddressBookEntry create(String owner,
        String firstName, String lastName,
        String phoneNumber) throws CreateException;
}
```

To access this bean, we might decide to use a session bean as a façade, much like in the previous example. Unlike our previous example, however, this new façade must store the owner's name so that only entries in that user's personal address book are retrieved. We might therefore choose to build a stateful session bean, like the one shown in Example 12-9.

Example 12-9. A stateful façade

```
import javax.ejb.*;
import java.util.*;
import javax.naming.*;
```

Example 12-9. A stateful façade (continued)

```
public class AddressBookBean implements SessionBean {
    private SessionContext context;
    private String userName;
    private LocalAddressBookEntryHome abeHome;

    public void setSessionContext(SessionContext aContext) {
        context=aContext;
    }

    public void ejbActivate() {
        init();
    }

    public void ejbPassivate() { abeHome = null; }
    public void ejbRemove() { abeHome = null;}

    public void ejbCreate(String userName) throws CreateException {
        this.userName = userName;
        init();
    }

    public PeopleDTO findAll(String firstName, String lastName)
    throws FinderException {
        Collection c = abeHome.findAll(userName);
        PeopleDTO dto = new PeopleDTO();

        for (Iterator i = people.iterator(); i.hasNext();) {
            LocalAddressBookEntry entry =
                (LocalAddressBookEntry) i.next();

            dto.addPerson(new Person(entry.getFirstName(),
                entry.getLastName(),
                entry.getPhoneNumber()));
        }

        return dto;
    }

    private void init() throws EJBException {
        try {
            String name = "java:comp/env/ejb/local/Address";

            InitialContext ic = new InitialContext();
            abeHome = (LocalAddressBookEntryHome) ic.lookup(name);
        } catch (Exception ex) {
            throw new EJBException("Error activating", ex);
        }
    }
}
```

As it stands, this façade must be stateful, because the `userName` variable is set by the initial call to `create()`. If it were stateless, each call to `findAll()` could potentially be

dispatched to a bean that had been created with a different username, and chaos would ensue.

Unfortunately, because this bean is stateful, it requires more container resources to maintain and manage than a stateless version of the same thing. As with any stateful session bean, we have to wonder if there is a stateless bean that could do the same thing.

Turning stateful into stateless

Our example is admittedly trivial, so it should be pretty obvious that a slight change to the façade's interface could allow it to be stateless. If the username was passed into each call to `findAll()`, instead of the `create` method, this bean could be made stateless.

For simple cases, it usually suffices to simply pass in all the relevant state data with each call. On more complex objects, however, this method becomes prohibitively expensive in terms of managing all the arguments and sending them over the network with each call. For these complex scenarios, there are a number of different solutions, depending on the nature of the data:

Use a client-side adapter

When simply keeping track of arguments becomes difficult, it is often useful to build a simple helper on the client in order to store the arguments. Generally, this helper presents the same interface as a stateful session bean would and stores the arguments on the client side. This allows the helper to adapt the stateful calls into stateless ones by passing in the stored arguments.

Use a stateless façade and entity EJBs

Entity EJBs are often the best choice for storing state data, even if the data is not permanent. A good example of this is a shopping cart, which is often implemented using stateful session beans. Using entity EJBs instead can give a number of benefits, including better performance, the ability to refer to data from multiple servers (useful with load-balanced web servers), and better resiliency when the server crashes.

Cache data in the client

Stateful session beans are used as a server-side data cache, much like the `HttpSession` object on the client side. Often, developers prefer the server-side cache because it is considered more robust. In fact, a stateful session bean that stores data in server memory is no more reliable than storing the data in client memory, since the stateful bean will be destroyed if its client goes away anyway. In a web application, it is usually better to cache data locally in the `HttpSession` object than it is to cache the data remotely, since `HttpSession` is equally reliable and much faster.

If you've considered all these solutions and stateful beans still seem most appropriate, go ahead and use them. Stateful beans can provide a big performance boost when the beginning of a multiple-request session involves creating expensive resources that would otherwise have to be reacquired with each method call. There are a whole slew of problems that stateful session beans solve. They can be a powerful tool when used properly.

In this chapter, we have seen a number of common mistakes in application architecture, the presentation tier, and the business tier. While the specifics might differ, the same few principals apply to every case. Know your enemy: recognize and fix anti-patterns as early as possible in the design process. Know your tools: understand how the costs and benefits of technologies relate to the problem at hand—don't use them just because they are new and cool. And of course, document, document, document.