ART OF JAVA WEB DEVELOPMENT

STRUTS

TAPESTRY

COMMONS

VELOCITY

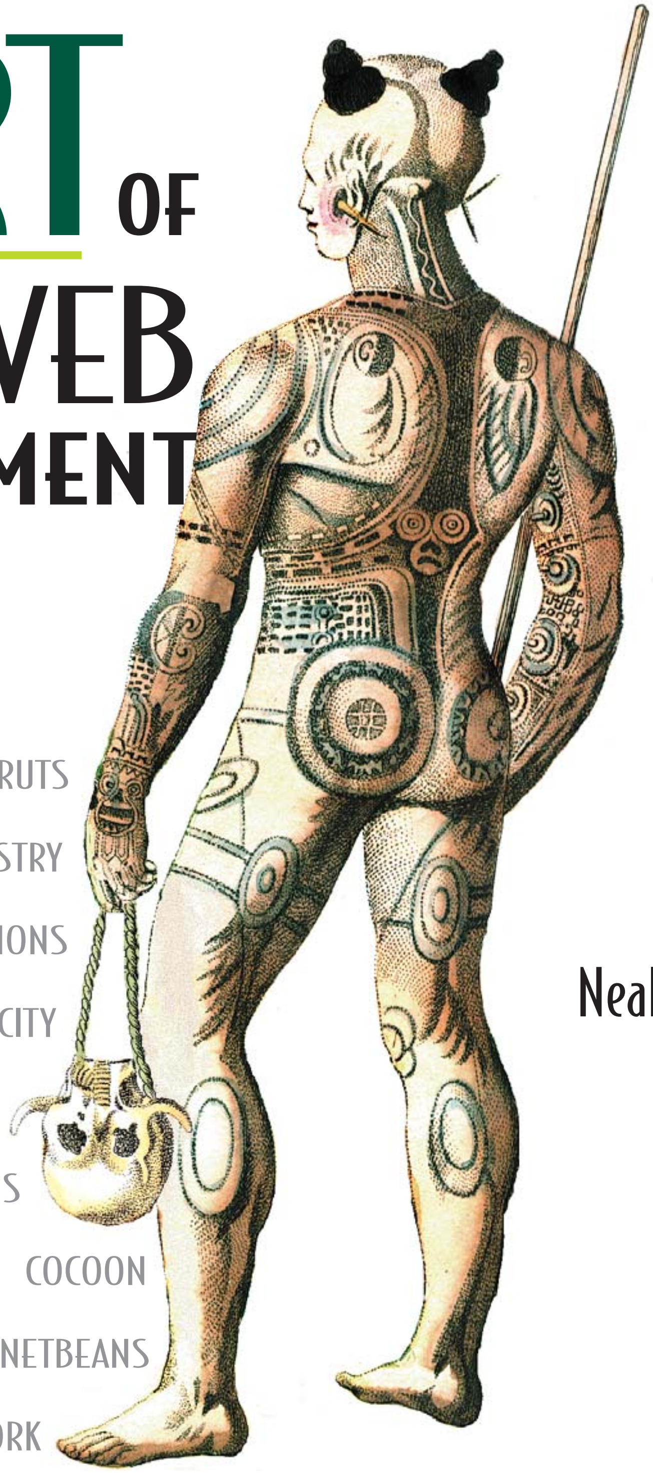
JUNIT

AXIS

COCOON

INTERNETBEANS

WEBWORK



Neal Ford

MANNING

brief contents

13 Handling flow

This chapter covers

- Application usability options
- Building undo operations
- Handling exceptions

In this chapter, we take a look at the usability and flow of a web application from a design standpoint. The greatest application in the world won't be used much if its flow doesn't meet the needs of its users, or if it doesn't handle exceptions gracefully and thus frustrates your users.

By studying flow, you can address both of these concerns. First, we look at how to reconcile often-requested usability elements (such as column sorting and page-at-a-time scrolling) with the design principles we've already discussed. We use the Model 2 version of the eMotherEarth e-commerce site introduced in chapter 4 as a base for this and future chapters.

You must also handle more infrastructural elements of flow, such as exception handling. Your application should be designed for robustness in the face of both user and application errors. In this chapter, you'll see how to build sortable columns, page-at-a-time scrolling, undo operations, and robust exception handling.

13.1 Application usability options

Users have an annoying habit of asking for features that seem easy and intuitive to use but that are difficult for the developer to implement. For example, two common features that users expect are sortable columns in tables and page-at-a-time scrolling. When adding bells and whistles to your application, you must avoid compromising its design and architecture. No matter how "pretty" it becomes, the developer who must maintain it later makes the final judgment on an application's quality.

13.1.1 Building the base: eMotherEarth.com

To illustrate these requests, an application must be in place. This and subsequent chapters use a simulated toy e-commerce site named eMotherEarth. The beginnings of this application appeared in chapter 2 to illustrate the evolution of web development from servlets and JSP. However, this version of the application is reorganized into a Model 2 application (see chapter 4). This section discusses the new architecture, and the following sections show how to incorporate usability options into a Model 2 application.

Packages

The application now appears in four major packages, shown in figure 13.1.

The boundary package contains two boundary classes, `ProductDb` and `OrderDb`, to persist the entities into the database. The application contains four entities: `Product`, `Order`, `Lineitem`, and `CartItem`. Only two boundary classes are required because Order and Lineitem are handled by the same boundary class; there is never a case in the application where you can add line items without adding an order, and the CartItem entity is never persisted. CartItem is a helper class that holds information until the time that an order is generated. The controller package contains the controller servlets for the application, and the util package contains miscellaneous utility classes, such as the database connection pool and the shopping cart.

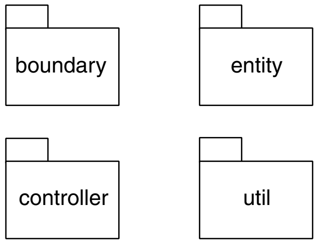


Figure 13.1 The Model 2 version of eMotherEarth.com is organized into four packages, each with different responsibilities.

For the sake of brevity, we show only the code that is unique to this application. The entire application is available with the source code archive as art_emotherearth_base. So, we won't show listings of classes that consist primarily of accessors and mutators and discuss only the interesting methods of the controller servlets.

Welcome

The first page of the application is a simple logon page, as shown in figure 13.2.

The welcome controller does more than just forward to a JSP with an entry field. It sets up global configuration items, like the database connection pool. Listing 13.1 shows the entire welcome controller.

Listing 13.1 The welcome controller

```
public class Welcome extends HttpServlet {

    public void init() throws ServletException {
        String driverClass =
                getServletContext().getInitParameter("driverClass");
        String password =
                getServletContext().getInitParameter("password");
        String dbUrl =
                getServletContext().getInitParameter("dbUrl");
        String user =
                getServletContext().getInitParameter("user");
        DBPool dbPool =
                createConnectionPool(driverClass, password, dbUrl,
                                     user);
        getServletContext().setAttribute("dbPool", dbPool);
    }

    private DBPool createConnectionPool(String driverClass,
                                        String password,
                                        String dbUrl,
                                        String user) {
```

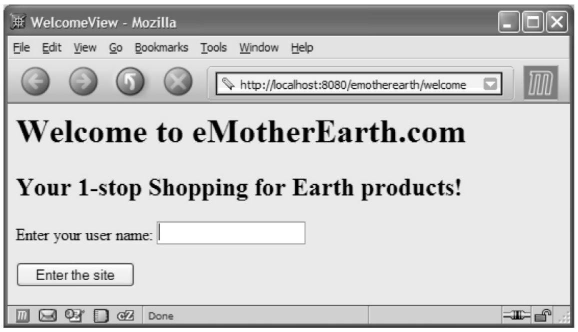
```
        DBPool dbPool = null;
        try {
            dbPool = new DBPool(driverClass, dbUrl, user, password);
        } catch (SQLException sqlx) {
            getServletContext().log(new java.util.Date() +
                                    ":Connection pool error", sqlx);
        }
        return dbPool;
    }

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
            throws ServletException, IOException {
        RequestDispatcher dispather =
                request.getRequestDispatcher("/WelcomeView.jsp");
        dispather.forward(request, response);
    }
}
```

The real action in the welcome controller occurs before the `doGet()` method is called. This method gets configuration parameters from the web.xml file and uses them to create the database connection pool that is utilized by the remainder of the application. Once the pool is created, it is added to the global collection. The `doGet()` method does nothing but forward directly to the view for the welcome.

Catalog

The next page of the application shows the user a catalog of all the items available for purchase. This page is shown in figure 13.3.

While the Welcome page strongly resembles the original version of the application from chapter 2, the Catalog page has some significant changes. First, it allows the user to click on the column heads to sort the items based on that column. Second, it offers multiple pages of items. Instead of showing all the items at the outset


WelcomeView - Mozilla
File Edit View Go Bookmarks Tools Window Help
http://localhost:8080/emotherearth/welcome
Done

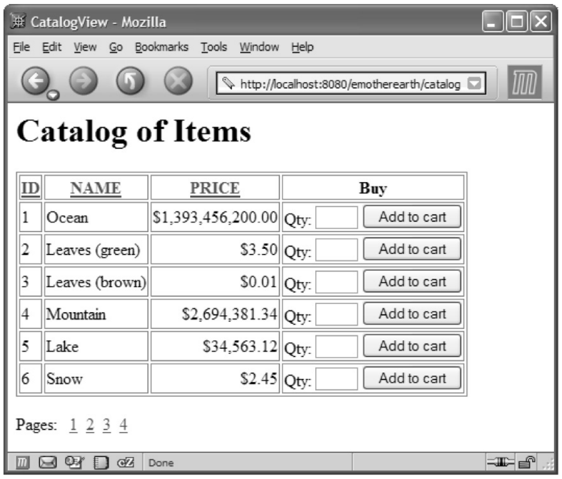
Welcome to eMotherEarth.com

Your 1-stop Shopping for Earth products!

Enter your user name:

Enter the site

Figure 13.2
This page allows the user to log on, while the servlet underneath sets up the web application.


CatalogView - Mozilla
File Edit View Go Bookmarks Tools Window Help
http://localhost:8080/emotherearth/catalog
Done

Catalog of Items

ID	NAME	PRICE	Buy
1	Ocean	$1,393,456,200.00	Qty: Add to cart
2	Leaves (green)	$3.50	Qty: Add to cart
3	Leaves (brown)	$0.01	Qty: Add to cart
4	Mountain	$2,694,381.34	Qty: Add to cart
5	Lake	$34,563.12	Qty: Add to cart
6	Snow	$2.45	Qty: Add to cart

Pages: 1 2 3 4

Figure 13.3
The Catalog page shows users the first of several pages of items they can buy from the site.

(a potentially long list), it shows a subset with hyperlinks at the bottom that allow the user to choose the display page.

Catalog is the workhorse controller in the application because it must execute the code that makes all the display techniques possible. Ideally, the JSP should have as little logic as possible—all the "real" code should execute in the controller. Figure 13.4 shows a UML sequence diagram highlighting the classes and methods called by the catalog controller. The real work in the controller is split up among the methods that appear in the sequence diagram. The `doPost()` method, which is fired from the Welcome page, appears in listing 13.2.

Listing 13.2 The catalog controller's doPost() method breaks the work down into smaller chunks.

```
public void doPost(HttpServletRequest request,
                   HttpServletResponse response) throws
        ServletException, IOException {

    HttpSession session = request.getSession(true);
    ensureThatUserIsInSession(request, session);
    ProductDb productDb = getProductBoundary(session);
    int start = getStartingPage(request);
    int recsPerPage = Integer.parseInt(getServletConfig().
                      getInitParameter("recsPerPage"));
    int totalPagesToShow = calculateNumberOfPagesToShow(
            productDb.getProductList().size(), recsPerPage);
    String[] pageList =
            buildListOfPagesToShow(recsPerPage,
                                   totalPagesToShow);
        List outputList = productDb.getProductListSlice(start,
                recsPerPage);
        sortPagesForDisplay(request, outputList);

        bundleInformationForView(request, start, pageList,
                                 outputList);
        forwardToView(request, response);
    }
```

The catalog controller makes sure the user is in the session. If the user isn't in the session (for example, upon the first invocation of the page), the `ensureThatUserIsInSession()` method adds the user to the session, pulling the name from the request collection. Either way, this method guarantees that the user is in the session.

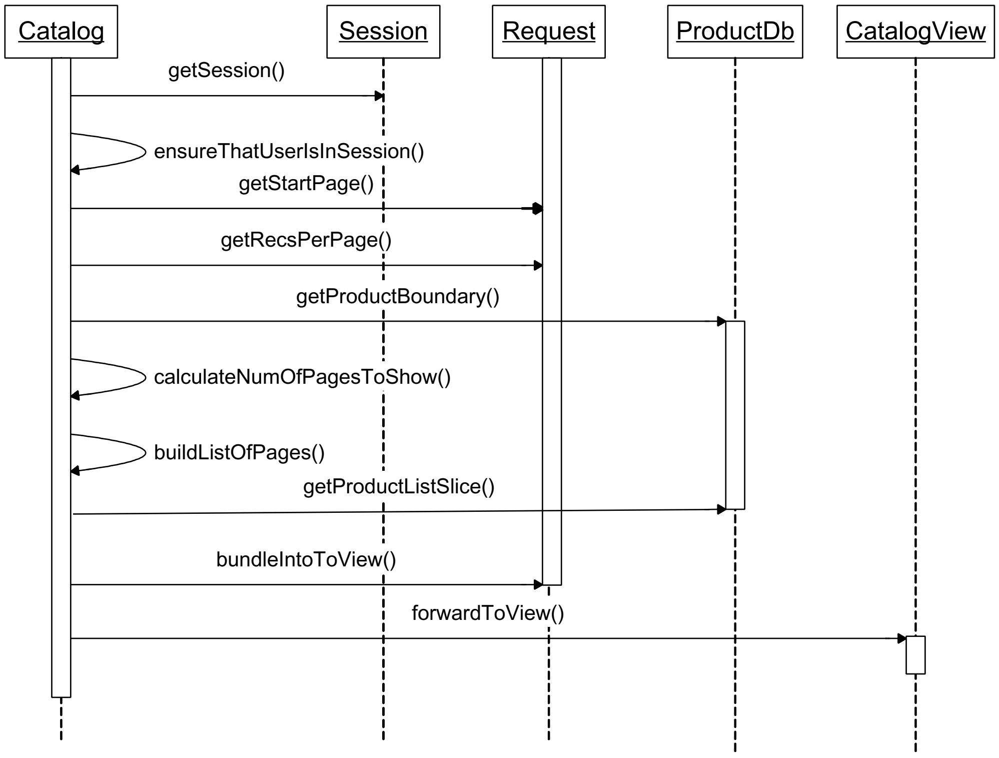


Figure 13.4 This sequence diagram shows the interactions and method calls from the catalog controller.

Next, the servlet starts to gather the components and information needed to build the display for the user. It calls the `getProductBoundary()` method to get the boundary class for product entities. This method is shown in listing 13.3.

Listing 13.3 The getProductBoundary() method either retrieves or creates a product boundary object.

```
private ProductDb getProductBoundary(HttpSession session) throws
        NumberFormatException {
    ProductDb products = (ProductDb) session.getAttribute(
            "productList");

    if (products == null) {
        products = new ProductDb();
        products.setDbPool(
                (DBPool) getServletContext().getAttribute(
                "dbPool"));
        session.setAttribute("productList", products);
    }
    return products;
}
```

The product boundary class encapsulates access to individual product entities, which it pulls from a database. All the data access code appears in the boundary class, leaving the product entities to include only product-specific domain information. The `ProductDb` class includes a property that is a `java.util.List` of `Product` entities. Figure 13.5 illustrates the relationship between these classes.

The application is designed so that every user gets a copy of this product boundary object. The controller's `getProductBoundary()` method is designed to place a copy of this object in the user's session upon first request. This behavior is a design decision whose goal is to ensure that every user has a copy of the object. The design represents a classic trade-off of memory versus speed. Although this strategy occupies more memory (a boundary object per user), the speed of access to the data is faster. If we wanted to create a more scalable application, we would handle the boundary differently. Chapters 14 and 15 include discussions of various caching and pooling mechanisms that are alternatives to this approach. The design decision to cache the boundary object in the user's

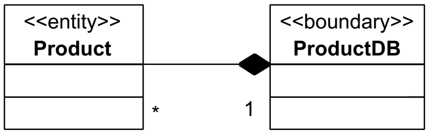


Figure 13.5 The `ProductDb` class includes an aggregation of `Product` objects and delivers them via a method that returns a `java.util.List`.

session highlights the fact that performance and scalability must illuminate every decision made in a web application.

13.1.2 Page-at-a-time scrolling

The page-at-a-time scrolling interface technique concerns the volume of information and the usability of the application. If you have a data-driven web application, you don't want to inundate the user with several thousand records on a single page. Most web sites handle this with page-at-a-time scrolling. When using this technique, the user sees only a single page worth of data and a list of pages. If users want to see more records, they can navigate to another page.

To implement this technique, the controller gathers some values from the request collection to help determine the number of pages to show at the bottom of the page. It calls the `getStartPage()` method, which appears in listing 13.4.

Listing 13.4 The getStartMethod() from the controller calculates the starting page number.

```
private int getStartingPage(HttpServletRequest request) {
    String recStart = request.getParameter("start");
    int start = 0;

    if (recStart != null)
        start = Integer.parseInt(recStart);
    return start;
}
```

This method pulls the `start` parameter from the request, parses it, and returns it. This parameter is available because the view encodes it into self-posting requests back to the controller for this page. Note that this method is designed to work in cases where the `start` parameter is not available (such as the first invocation of the page).

Users must specify the page they want through the view, which is specified by the series of hyperlinks at the bottom of the page. The values of these hyperlinks (in other words, the generated HTML for them) are shown in listing 13.5.

Listing 13.5 The page links at the bottom allow the user to navigate between pages.

```
<p> Pages: &nbsp;
<a href='catalog?start=0'>1</a>&nbsp;
<a href='catalog?start=6'>2</a>&nbsp;
<a href='catalog?start=12'>3</a>&nbsp;
<a href='catalog?start=18'>4</a>&nbsp;
```

Each of the page links contains a reference to the controller (catalog) and the starting record for that page. You will notice in listing 13.5 that each page starts six records beyond the previous page. The `getStartPage()` method of the controller pulls the `start` parameter value from the request and uses it to calculate which records should appear on the page. The number of records per page is set through a servlet configuration parameter. In this case, it is set to six records per page. The next line of code in the controller is the retrieval of that value from the `servlet-Config` object.

The next method called by the controller is the `calculateNumberOfPages-ToShow()` method, which appears in listing 13.6.

Listing 13.6 This method calculates the number of pages it will take to show all the requested records.

```
private int calculateNumberOfPagesToShow(int numInList,
                                         int recsPerPage) {
    int totalToShow = numInList / recsPerPage;

    if (numInList % recsPerPage != 0)
        ++totalToShow;
    return totalToShow;
}
```

The `calculateNumberOfPagesToShow()` method accepts the total number of records available and the requested records per page, and then calculates the number of pages required. Note that the contingency of having a last page that isn't completely full is handled with the use of the modulus operator (`%`) to ensure that enough pages exist.

The next method called is `buildListOfPagesToShow()`, which builds up an array of strings containing the displayable hyperlinks. This method is shown in listing 13.7.

Listing 13.7 This method builds the list of hyperlinks embedded at the bottom of the page.

```
private String[] buildListOfPagesToShow(int recsPerPage,
                                        int totalPagesToShow) {
    String[] pageList = new String[totalPagesToShow];
    StringBuffer work = new StringBuffer(20);
    int currentPage = 0;

    for (int i = 0; i < totalPagesToShow; i++) {
        work.setLength(0);
        work.append("<a href='catalog?start=").append(
                currentPage).append("'>").append(i + 1).append(
                "</a>&nbsp;");
            pageList[i] = work.toString();
            currentPage += recsPerPage;
        }
        return pageList;
    }
```

The `buildListOfPagesToShow()` method builds up a list of hyperlinks with the appropriate page and start record information embedded in them. It iterates over a list up to the total number of pages to show, building a `StringBuffer` with the appropriate hyperlink and display data. Eventually, it returns the array of strings that includes the page list. This page list is passed to the view in a request parameter (it is one of the parameters to the `bundleInformationForView()` method).

The view extracts that information and places it on the bottom of the page. Listing 13.8 shows the snippet of code at the bottom of the page that builds this list of pages.

Listing 13.8 The CatalogView page uses the pageList to build the list of hyperlinks at the bottom.

```
<%-- show page links --%>
<p> Pages: &nbsp;
<%
    String[] pageList = (String[]) request.getAttribute("pageList");
    if (pageList != null) {
        for (int i = 0; i < pageList.length; i++) {
            out.println(pageList[i]);
        }
    }
%>
```

The scriptlet in listing 13.8 walks over the `pageList` passed from the controller and outputs each of the links. The spacing is already built into the HTML in the `pageList`, simplifying the job of the scriptlet code.

Using JSTL

The kind of scriptlet code that appears in listing 13.8 is generally not required if you are using a modern JSP specification (and of course a servlet container that supports this specification). The JSP Standard Tag Library (JSTL) includes custom JSP tags that handle common chores like iteration. The JSTL version of this code is shown in listing 13.9

Listing 13.9 JSTL provides custom tags to help with iteration over a collection.

```
<% pageContext.setAttribute("pageList",
        (String[]) request.getAttribute("pageList")); %>
<p> Pages: &nbsp;
<c:forEach var="page" items="${pageList}">
    <c:out value="${page}" escapeXml="false"/>
</c:forEach>
```

This code is much better than the scriptlet alternative because the custom JSTL tag handles conditions like null properties by just ignoring the tag. This is a case in which using JSTL does greatly improve the readability and maintainability of your code without compromising the clean separation of model, view, and controller.

JSTL is also used to show the rows of data on this view page. Listing 13.10 contains the code that displays the products available for purchase.

Listing 13.10 The Catalog page uses JSTL tags to help the readability of the JSP code.

```
<%
    Integer start = (Integer) request.getAttribute("start");
    int s = start.intValue();
%>
Catalog of Items
</h1>
<table border=1>
    <tr><th><a href="catalog?sort=id&start=<%= s %>">ID</a></th>
    <th><a href="catalog?sort=name&start=<%= s %>">NAME</a></th>
    <th><a href="catalog?sort=price&start=<%= s %>">PRICE</a></th>
    <th>Buy</th></tr>
    <c:forEach var="product" items="${outputList}">
    <tr>
        <td><c:out value="${product.id}"/></td>
        <td><c:out value="${product.name}"/></td>
        <td align='right'>
            <c:out value="${product.priceAsCurrency}"/>
        </td>
        <td>
            <form action="showcart" method="post">
                Qty: <input type="text" size="3" name="quantity">
                <input type="hidden" name="id"
                            value=<c:out value="${product.id}"/>>
                <input type="submit" value="Add to cart">
            </form>
        </td>
    </tr>
    </c:forEach>
</table>
```

You can contrast this code with the similar code in chapter 3 (section 3.5.2 and listing 3.16). This version is much better because the JSTL tag is used to output the values of the individual properties of the bean passed to this page by the controller. The version in listing 3.16 used a custom tag to output the HTML directly from Java code. In listing 13.10, a presentation expert has full access to the fields and can make changes to the look and feel of the application without touching any Java code.

Another powerful feature of JSTL is the ability to use dot notation to access embedded property values of objects. Consider the ShowCart JSP page for this Model 2 version of eMotherEarth. It appears in listing 13.11.

Listing 13.11 The ShowCart JSP uses JSTL to access the embedded product object in the CartItem class.

```
<%
    pageContext.setAttribute("cartItems", cart.getItemList());
%>
<table border=1>
    <tr>
        <c:forEach var="col" items="ID,NAME,PRICE,QUANTITY,TOTAL">
            <th><c:out value="${col}"/></th>
        </c:forEach>
    </tr>
    <c:forEach var="cartItem" items="${cartItems}">
    <tr>
        <td><c:out value="${cartItem.product.id}"/></td>
        <td><c:out value="${cartItem.product.name}"/></td>
        <td><c:out value="${cartItem.product.priceAsCurrency}"/></td>
        <td><c:out value="${cartItem.quantity}"/></td>
        <td><c:out value="${cartItem.extendedPriceAsCurrency}"/></td>
    </tr>
    </c:forEach>
    <tr>
        <td>&nbsp;</td>
        <td>&nbsp;</td>
        <td>&nbsp;</td>
        <td align='right'>Grand Total =</td>
        <td align='right'><%= cart.getTotalAsCurrency() %></td>
    </tr>
</table>
```

The `CartItem` and `Product` classes are related to each other. The `CartItem` class encapsulates a `Product` object so that it won't have to duplicate the information already encapsulated by `Product`. The `ShoppingCart` class composes the `CartItem` class because it includes a collection of `CartItems`. It is a composition relationship

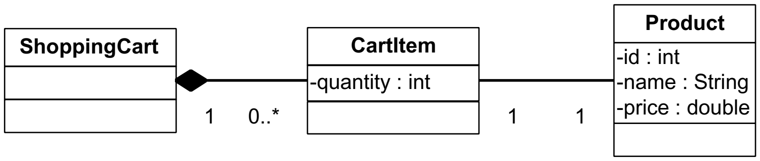


Figure 13.6 The `ShoppingCart`, `CartItem`, and `Product` classes are related. `ShoppingCart` composes `CartItem`, which has a one-to-one association with a `Product`.

rather than an aggregation because the `ShoppingCart` class is responsible for the creation and destruction of the `CartItem` objects. The relationship between these classes is illustrated in figure 13.6.

Because of the relationship between `CartItem` and `Product`, you may find it difficult to cleanly access the encapsulated `Product` object. Using regular iteration scriptlets, you end up with code that looks like listing 13.12.

Listing 13.12 The embedded objects make iteration complex.

```
<table border=1>
<tr><th>ID</th><th>NAME</th><th>PRICE</th>
        <th>QUANTITY</th><th>TOTAL</th></tr>
<%
    Iterator iterator = cart.getItemList().iterator();
    while (iterator.hasNext()) {
        CartItem ci = (CartItem) iterator.next();
        pageContext.setAttribute("ci", ci);
        Product p = ci.getProduct();
        pageContext.setAttribute("p", p);
%>
<tr><td><jsp:getProperty name="p" property="id" /></td>
<td><jsp:getProperty name="p" property="name" /></td>
<td align='right'><jsp:getProperty name="p"
                        property="priceAsCurrency" /></td>
<td align='right'><jsp:getProperty name="ci"
                        property="quantity" /></td>
<td align='right'><jsp:getProperty name="ci"
                        property="extendedPriceAsCurrency" /></td>
</tr>
<%
    }
%>
<tr><td>&nbsp;</td><TD>&nbsp;</td><TD>&nbsp;</td>
<td align='right'>Grand Total =</td>
<td align='right'><%= cart.getTotalAsCurrency() %></td>
</tr>

</table>
```

In the iteration code, to be able to access both `CartItem` and `Product` through the standard JSP tags, you must add both to the `pageContext` collection as you iterate over the collection.

JSTL makes this job much easier. The syntax for embedded objects is much cleaner because you can directly access the embedded object using dot notation. The code in listing 13.11 performs the same task but is less cluttered by the use of the JSTL `forEach` tag instead of handcrafted iteration. Note that the chain of method calls follows the same standard Java guidelines. To get to the `Name` property of the product embedded inside `cartItem`, you write the following Java code:

```
cartItem.getProduct().getName()
```

This code is exactly equivalent to the JSTL code:

```
cartItem.product.name
```

In other words, the JSTL tag isn't looking for a public member variable when using the dot notation but rather a method that follows the standard Java naming convention for accessing methods.

13.1.3 Sortable columns

Users are accustomed to being able to manipulate data that they see on the screen. Most applications allow them to do so to one degree or another. Selective sorting is a facility that users are familiar with from such applications as spreadsheets and databases. When the user clicks on the title for a particular column, all the results are sorted based on that column.

As with much of the functionality users have come to expect in traditional applications, implementing this kind of dynamic behavior is more difficult in the HTTP/HTML-governed world of web applications. For a Model 2 application, the sorting is provided by the model, and the selection must be specified through the view. Like the page-at-a-time scrolling technique, sorting is handled through hyperlinks that post back to the Catalog page, passing a parameter indicating the desired sorting criteria.

Listing 13.2, the code for the catalog controller's `doPost()` method, includes the method call that handles sorting. Named `sortPagesForDisplay()`, this method appears in listing 13.13.

Listing 13.13 This method handles the sorting of the records for display.

```
private void sortPagesForDisplay(HttpServletRequest request,
                                 ProductDb productDb,
                                 List outputList) {
```

```
    productDb.sortList(request.getParameter("sort"),
                       outputList);
}
```

The `sortPagesForDisplay()` method is called after the output list has already been generated. Note that it must appear after the code that decides what page's worth of records to show. The sorting must apply to the records that appear on the current page and not to the entire set of records from all pages. Thus, the sorting operation takes place on the list subset already generated by the previous methods.

The list for display is a `java.util.List` type, so the standard sorting mechanism built into Java is applicable. We need to be able to sort by a variety of criteria, so it is not sufficient to allow the `Product` class to implement the `Comparable` interface. The `Comparable` interface is used when you have a single sort criterion for a member of a collection. It allows you to specify the rules for how to sort the entities. The sort routines built into Java use these rules to determine how to sort the records. While it is possible to make the single `compareTo()` method of the `Comparable` interface handle more than one sort criterion, it is always a bad idea. This method becomes a long, brittle series of decision statements to determine how to sort based on some external criteria.

If you need to sort based on multiple criteria, you are much better off creating small `Comparator` subclasses. All the sort routines built into Java (for both the arrays and collections helpers) take an additional parameter of a class that implements the `Comparator` interface. This interface (minus the JavaDocs) appears in listing 13.14.

Listing 13.14 The Comparator interface allows the user to specify discrete sorting criteria.

```
package java.util;

public interface Comparator {

    int compare(Object o1, Object o2);
    boolean equals(Object obj);
}
```

For the `Product` sorting operation, you need the ability to sort on name, price, and ID. To that end, three `Comparator` implementers exist. Because of their similarity, only one of the three created for this application is shown (listing 13.15).

Listing 13.15 The PriceComparator class sorts Product objects based on price.

```
package com.nealford.art.emotherearth.util;

import java.util.Comparator;
import com.nealford.art.emotherearth.entity.Product;

public class PriceComparator implements Comparator {

    public int compare(Object o1, Object o2) {
        Product p1 = (Product) o1;
        Product p2 = (Product) o2;
        return (int) Math.round(p1.getPrice() - p2.getPrice());
    }

    public boolean equals(Object obj) {
        return this.equals(obj);
    }
}
```

The recipe for creating `Comparator`'s `compareTo()` methods is always the same: cast the two objects passed to you by the sort routine into the type of objects you are comparing, and then return a negative, positive, or zero number indicating which object appears before the other when sorted.

Once `Comparators` exist, the sorting routines can use them to sort arrays or collections. The `sortPagesForDisplay()` method from listing 13.13 looks for a request parameter named `sort`. The actual sorting is done in the boundary class for products. The method called from the controller, `sortList()`, appears in listing 13.16.

Listing 13.16 The sortList() method is a helper method that sorts the list based on the column name passed to it.

```
public List sortList(String criteria, List theList) {
    if (criteria != null) {
        Comparator c = new IdComparator();
        if (criteria.equalsIgnoreCase("price"))
            c = new PriceComparator();
        else if (criteria.equalsIgnoreCase("name"))
            c = new NameComparator();

        Collections.sort(theList, c);
    }
    return theList;

}
```

If it is present, the appropriate `Comparator` class is applied to the output list. This output list is bundled in a request parameter and sent to the View page for display by the controller. The View page doesn't have to perform any additional work to display the sorted records—all the sorting is done in the boundary class, called by the controller.

The last piece of the sorting puzzle resides in the view portion, where the user specifies the sort criteria. Listing 13.10 shows the CatalogView JSP. The sorting portion of that page appears in listing 13.17.

Listing 13.17 The sorting criteria are embedded in hyperlinks at the top of the page.

```
<%
    Integer start = (Integer) request.getAttribute("start");
    int s = start.intValue();
%>
Catalog of Items
</h1>
<table border=1>
    <tr><th><a href="catalog?sort=id&start=<%= s %>">ID</a></th>
    <th><a href="catalog?sort=name&start=<%= s %>">NAME</a></th>
    <th><a href="catalog?sort=price&start=<%= s %>">PRICE</a></th>
    <th>Buy</th></tr>
```

The hyperlinks in listing 13.17 supply two values for reposting to the catalog controller. The first is the sort criteria to apply, and the second is the starting page. When the user clicks on one of these hyperlinks, the page reposts to the catalog controller, which uses these parameters to modify the contents of the page before redisplaying it.

Note that, as much as possible, the real workflow part of the application is performed in the controller. The data portions of the application are performed in the model classes. The view is very lightweight, handling display characteristics and supplying values, which allows the user to change the view via parameters sent to the controller.

Using factories

The `sortList()` method uses a simple set of `if` comparisons to determine which `Comparator` to apply to the list. This is sufficient for a small number of criteria but quickly becomes cumbersome if a large number of options are available. In that case, a factory class simplifies the code in the boundary class by handling the decision itself. An example of such a factory class appears in listing 13.18.

Listing 13.18 The ComparatorFactory class offloads the decision process to a singleton factory.

```
package com.nealford.art.emotherearth.util;

import java.util.Comparator;

public class ProductComparatorFactory {
    private static ProductComparatorFactory internalReference;

    private ProductComparatorFactory() {
    }

    public static ProductComparatorFactory getInstance() {
        if (internalReference == null)
            internalReference = new ProductComparatorFactory();
        return internalReference;
    }

    public synchronized final Comparator getProductComparator(
                String criteria) {
        String className = this.getClass().getPackage().getName() +
                '.' + toProperCase(criteria) + "Comparator";
        Comparator comparator = null;
        try {
            comparator = (Comparator) Class.forName(className).
                         newInstance().;
        } catch (Exception defaultsToIdComparator) {
            comparator = new IdComparator();
        }
        return comparator;
    }

    public String toProperCase(String theString) {
        return String.valueOf(theString.charAt(0)).toUpperCase() +
                theString.substring(1);
    }
}
```

Builds Comparator name from string parameter

Dynamically instantiates Comparator

Defaults to idComparator if an exception occurs

The `ProductComparatorFactory` class is implemented as a singleton object (so that only one of these objects will ever be created) via the static `getInstance()` method and the private constructor. This factory uses the name of the sort criteria to match the name of the `Comparator` it dynamically creates. When the developer sends a sort criterion (like `name`) to this factory, the factory builds up a class name in the current package with that criterion name plus "Comparator." If an object based on that class name is available in the classpath, an instance of that `Comparator` is returned. If not, the default `IdComparator()` is returned.

Using a factory in this way allows you to add new sorting criteria just by adding new classes to this package with the appropriate name. None of the surrounding code has to change. This is one of the advantages to deferring such decisions to a factory class, which can determine which instances to return.

This factory could be improved by removing the reliance on the name of the class. A superclass `Comparator` with a method indicating to what fields it is tied would remove the reliance on the name of the class matching the name of the criteria. In that case, the factory would iterate through all the potential `Comparators` and call the `getField()` method until it finds the appropriate `Comparator` object. This is easier if all the `Comparators` reside in the same package so that the factory could iterate over all the classes in that package.

13.1.4 User interface techniques in frameworks

Implementing page-at-a-time scrolling and sortable columns in the frameworks from part 2 is accomplished with varying degrees of difficultly. Some of the frameworks already include this behavior, whereas InternetBeans Express prevents it.

Struts

Using Struts to build the user interface elements that we've seen in the previous sections is easy. In fact, the code presented in this chapter works with few modifications. In Struts, you move the controller code to actions, but the model and view code remains the same. Of course, you can move the iteration and other display characteristics to Struts tags, but the fundamental code remains the same. Because Struts is close to a generic Model 2 application, the framework doesn't interfere with building code like this.

Tapestry

Tapestry already encapsulates the two user interface elements discussed in the previous sections. The built-in table component supports both page-at-a-time scrolling and sortable columns (see chapter 6, figure 6.6). The sortability in Tapestry is accomplished through interfaces that define the column headers. This behavior highlights one of the advantages of an all-encompassing framework like Tapestry. Chances are good that it already implements many of the common characteristics you would build by hand in other frameworks. The disadvantage appears when you want to build something that isn't already there. Because the framework is more complex, it takes longer to build additions.

WebWork

Like Tapestry, WebWork also includes a table component that features sortable columns and page-at-a-time scrolling (see chapter 7, figure 7.3). Although implemented differently from Tapestry, this behavior is still built into the framework. Even though WebWork generally isn't as complex as Tapestry, it still requires a fair amount of work to build something that isn't already supported.

InternetBeans Express

The architecture of InternetBeans Express effectively prevents this kind of customization without digging deeply into the components that make up the framework. While building applications quickly is this framework's forte, customizing the behavior of those applications is not. This is a shortcoming of overly restrictive frameworks and is common with Rapid Application Development (RAD).

Velocity

Our user interface code could easily be written using Velocity. Velocity's syntax would simplify the view portion of the code even more than JSTL. Generally, Velocity isn't complex enough to prevent adding features like the ones in this chapter. Because it is a simple framework, it tends to stay out of your way.

Cocoon

Using Extensible Server Pages (XSP), it shouldn't be difficult to build our user interface techniques in Cocoon. XSP generally follows similar rules to JSP, so the user interface portion isn't complicated. Because the web portion of Cocoon relies on Model 2, the architecture we presented in the previous sections falls right in line with a similar Cocoon application.

13.2 Building undo operations

Another common flow option in traditional applications is the ability to perform an undo operation. This feature is usually implemented as a conceptual stack, where each operation is pushed onto the stack and then popped off when the user wants to undo a series of operations. The stack usually has a finite size so that it doesn't negatively affect the operating system. After all, an infinite undo facility must either consume more memory or build a mechanism to offload the work to permanent storage of some kind.

Undo may also encompass traditional transaction processing. Ultimately, transactions that roll back can be thought of as sophisticated undo operations for a set of tables when the operation is unsuccessful. Either a database server or an application server working in conjunction with a database server normally handles transaction processing. You have two options when building undo operations for a web application: either using database transaction processing or building an in-memory undo.

13.2.1 Leveraging transaction processing

Most database servers handle transactions for you, at varying degrees of sophistication. The Java Database Connectivity (JDBC) API allows you to handle transactions via the `setAutoCommit()` method, which determines whether every atomic operation occurs within a transaction or if the developer decides the transaction boundaries. If the developer controls the transactions, then either a `commit()` or a `rollback()` method call is eventually issued. Modern JDBC drivers (those that support the JDBC 3 API) will also allow you to create save-points and roll back to a save-point within a larger transaction.

Transactions in Model 2 applications

In a Model 2 application, the transaction processing and other database-related activities occur in the boundary classes. In fact, if you ever find yourself importing `java.sql.*` classes into other parts of the application, you have almost certainly violated the clean separation of responsibilities.

In the eMotherEarth application, the transaction processing occurs within the `Order` boundary class. It must ensure that both order and line item records are completely written or not at all. The `addOrder()` method composes all the other methods of the class and appears in listing 13.19.

Listing 13.19 The OrderDb boundary class's addOrder() method

```
public void addOrder(ShoppingCart cart, String userName,
                     Order order) throws SQLException {
    Connection c = null;
    PreparedStatement ps = null;
    Statement s = null;
    ResultSet rs = null;
    boolean transactionState = false;
    try {
        c = dbPool.getConnection();
        transactionState = c.getAutoCommit();
        int userKey = getUserKey(userName, c, ps, rs);
        c.setAutoCommit(false);
        addSingleOrder(order, c, ps, userKey);
        int orderKey = getOrderKey(s, rs);
```

```
            addLineItems(cart, c, orderKey);
            c.commit();
            order.setOrderKey(orderKey);
        } catch (SQLException sqlx) {
            s = c.createStatement();
            c.rollback();
            throw sqlx;
        } finally {
            try {
                c.setAutoCommit(transactionState);
                dbPool.release(c);
                if (s != null)
                    s.close();
                if (ps != null)
                    ps.close();
                if (rs != null)
                    rs.close();
            } catch (SQLException ignored) {
            }
        }
    }
```

The `addOrder()` method retrieves a connection from the connection pool and saves the transaction state for the connection. This behavior allows the transaction state to be restored before it is placed back into the pool. If you are creating your own connections every time you need one, you don't have to. If you are reusing connections from a pool or cache, you should also make sure that they go back into the pool with the same state they had when they came out.

The `addOrder()` method gets a connection, starts a transaction implicitly by calling `setAutoCommit(false)`, and calls the `addSingleOrder()` method. After obtaining the key of the new order, it adds the line items associated with this order and commits the transaction. If any operation fails, a `SQLException` is generated and the entire operation is rolled back.

None of the code in any of the called methods is in any way unusual—it is typical JDBC code for entering values into a table. Note that all database access, including the transaction processing, occurs in the boundary class. The boundary class accepts entity objects and handles persisting them into the database. It would be easy to change database servers (even to change to something radically different, like an object-oriented database server) and modify the code in this boundary class only. Chapter 12 describes the process of taking a Model 2 application and porting it to Enterprise JavaBeans by making changes to only the boundary classes.

Handling generated keys

One behavior that is not handled in a standard way across database servers is key generation. Most database servers have a facility for generating keys automatically. However, key generation is not part of the ANSI SQL standard, so each database server is free to implement it in any way it likes. In our sample, this detail is handled in the `addOrder()` method via the call to `getOrderKey()`, which uses the features specific to MySQL to retrieve the last-generated key. Listing 13.20 shows the `getOrderKey()` method.

Listing 13.20 The getOrderKey() method retrieves the last key generated for this connection to the database.

```
private int getOrderKey(Statement s, ResultSet rs) throws
        SQLException {
    rs = s.executeQuery("SELECT LAST_INSERT_ID()");
    int orderKey = -1;
    if (rs.next())
        orderKey = rs.getInt(1);
    else
        throw new SQLException(
                "Order.addOrder(): no generated key");
    return orderKey;
}
```

MySQL includes a built-in stored procedure that returns the last key generated for this connection to the database. This procedure protects against a large number of concurrent users inserting new records because it returns the key for the record associated with this connection. Notice that this forces our application to use the same connection across method calls because the key generation is tied to the database connection.

Because this procedure is not standardized across database servers, you should always be careful to isolate this behavior into its own method, decoupling it from the rest of the application. If you change database servers, you should be able to change this single method and not have to change the surrounding code. Separation of responsibilities and loose coupling works on both a micro and a macro level.

Transactions via JSTL

JSTL includes SQL-specific custom tags that allow transaction processing within the JSP. It works with the SQL-based tags also defined in JSTL. Listing 13.21 shows a couple of examples of using the transaction tag in JSTL.

Listing 13.21 JSTL includes a transaction tag that works with the SQL tags.

```
<h2>Creating table using a transaction</h2>

<sql:transaction dataSource="${example}">
  <sql:update var="newTable">
    CREATE TABLE PRODUCTS (
      ID INTEGER NOT NULL AUTO_INCREMENT,
      NAME VARCHAR(100),
      PRICE DOUBLE PRECISION,
      CONSTRAINT PK_ID PRIMARY KEY (ID)
     )
  </sql:update>
</sql:transaction>

<h2>Populating table in one transaction</h2>

<sql:transaction dataSource="${example}">
  <sql:update var="updateCount">
    INSERT INTO PRODUCTS (NAME, PRICE) values ("Snow", 2.45);
  </sql:update>
  <sql:update var="updateCount">
    INSERT INTO PRODUCTS (NAME, PRICE) values ("Dirt", 0.89);
  </sql:update>
  <sql:update var="updateCount">
    INSERT INTO PRODUCTS (NAME, PRICE) values ("Sand", 0.15);
  </sql:update>
</sql:transaction>
```

The ability to handle transactions directly within a JSP page is handy for small applications, but you should avoid using it in most applications. This facility was intended to make it easy for you to create web applications completely within JSP—without being forced to embed scriptlet code. One of its goals is to create RAD kinds of environments for JSP. The problem with this code is that it violates the tenets of Model 2 applications, namely the separation of responsibilities. While convenient, it introduces undesirable design flaws in your application. Therefore, I recommend that you don't use these tags, and use a cleaner Model 2 architecture instead.

13.2.2 Using the Memento design pattern

Transaction processing works nicely for information persisted in relational databases. It is the best kind of code to leverage—someone else wrote it, debugged it, and stands behind it! However, situations arise when you don't want to make use of transaction processing. For example, you may want to keep information in memory and not bother persisting it to permanent storage until a certain milestone is reached. The perfect example of this kind of information is the shopping cart in an e-commerce application. The shopper may never check out but instead abandon the shopping cart and wander away to another site without notifying your application. Beyond transaction-processing behavior, you might also want to make available undo behavior in your web application. This amounts to a kind of in-memory transaction processing, although the semantics are different.

Undo operations in traditional applications are typically handled via the Memento design pattern. The intent behind this pattern is to capture and externalize an object's internal state so that the object can be restored to the original state, all without violating encapsulation. Three participant classes exist for Memento, as shown in table 13.1.

Table 13.1 Participant classes of the Memento design pattern

Participant	Function
Memento	Stores the state of the original object and protects against access of that state by external objects.
Originator	Creates the `Memento` containing a snapshot of its state and uses the `Memento` to restore its state.
Caretaker	Holds onto the `Memento` without operating on it or spying on its internal state.

The relationship between these participants is illustrated in figure 13.7.

The `Originator` is the class whose state needs to be stored, and the `Memento` is where that state is stored. The `Caretaker` holds onto the `Memento` until the `Originator` needs it back. The `Caretaker` may encapsulate a collection of `Memento`s. When used for undo, the `Caretaker` usually keeps the `Memento`s in an undo stack.

Creating bookmarks in eMotherEarth

Using the Memento design pattern in a web application is slightly different than the implementation in traditional applications. This is a frequent side effect of applying design patterns to architectures beyond their original intent. For the

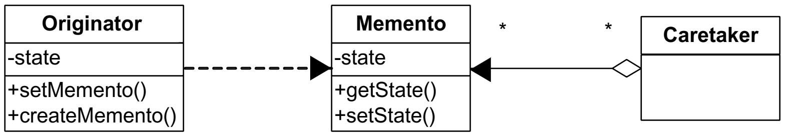


Figure 13.7 The participant classes in the Memento design pattern revolve around their relationship to the `Memento` class.

eMotherEarth application, we will allow the user to create bookmarks in their shopping cart. For example, the user can buy several related items, create a bookmark, and then later roll back to that bookmark. The bookmark facility uses a stack, which means users can create as many bookmarks as they like and unroll them in the reverse order from which they were created.

The first step is to create the `Memento` class. This class must access the private data of the `ShoppingCart` class without exposing it to the outside world. The best way to handle this in Java is with an *inner* class. Inner classes can access the private member variables of the outer class without exposing the encapsulated data to the rest of the world. The updated version of the `ShoppingCart` class is shown in listing 13.22.

Listing 13.22 The updated ShoppingCart class

```
package com.nealford.art.memento.emotherearth.util;

import java.io.Serializable;
import java.text.NumberFormat;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import com.nealford.art.memento.emotherearth.entity.CartItem;

public class ShoppingCart implements Serializable {
    private List itemList;
    private static final NumberFormat formatter =
            NumberFormat.getCurrencyInstance();

    public ShoppingCart() {
        itemList = new ArrayList(5);
    }

    public void addItem(CartItem ci) {
        itemList.add(ci);
    }

    public double getCartTotal() {
        Iterator it = itemList.iterator();
        double sum = 0;
        while (it.hasNext())
            sum += ((CartItem) it.next()).getExtendedPrice();
        return sum;
    }

    public String getTotalAsCurrency() {
        return formatter.format(getCartTotal());
    }

    public java.util.List getItemList() {
        return itemList;
    }
                                                          Sets a
    public ShoppingCartMemento setBookmark() {      <--   bookmark
        ShoppingCartMemento memento = new ShoppingCartMemento();
        memento.saveMemento();
        return memento;
    }

    public void restoreFromBookmark(ShoppingCartMemento memento) {   <--
        this.itemList = memento.restoreMemento();                   Restores a
    }                                                               bookmark

    public class ShoppingCartMemento {    <--  Stores state
        private List itemList;                 information

        public List restoreMemento() {
            return itemList;
        }

        public void saveMemento() {
            List mementoList = ShoppingCart.this.itemList;
            itemList = new ArrayList(mementoList.size());
            Iterator i = mementoList.iterator();
            while (i.hasNext())
                itemList.add(i.next());
        }
    }

}
```

The important change to the `ShoppingCart` class is the inclusion of the inner class `ShoppingCartMemento`. It includes a single private member variable of type `List`. This is the variable that will hold the current state of the shopping cart list when a bookmark is set. The `restoreMemento()` method simply returns the list. The `saveMemento()` method is responsible for taking a snapshot of the state of the shopping cart. To do this, it must access the private member variable from the outer shopping cart class. The syntax for this in Java uses the class name followed by `this`, followed by the member variable:

```
List mementoList = ShoppingCart.this.itemList;
```

Even though `itemList` is private in `ShoppingCart`, it is available to the inner class. This relationship is perfect for the Memento pattern, where the `Memento` needs access to the private member variables of the `Originator` without forcing the `Originator` to violate encapsulation.

The `ShoppingCart` class has two new methods: `setBookmark()` and `restoreFromBookmark()`. The `setBookmark()` method creates a new `Memento`, saves the current state, and returns it. The `restoreFromBookmark()` method accepts a `Memento` and restores the state of the `itemList` back to the list kept by the `Memento`.

The Caretaker

For a web application, the session object is the perfect `Caretaker` for the `Memento`. It is tied to a particular user and contains arbitrary name-value pairs. However, saving a single `Memento` isn't very useful, and saving a stack of `Mementos` is just as easy as saving one. So, in the eMotherEarth application we allow the user to keep a stack of `Mementos`. This process is managed by the controller servlet. The updated `doPost()` method in the ShowCart controller servlet appears in listing 13.23.

Listing 13.23 The ShowCart controller acts as the Memento Caretaker.

```
public void doPost(HttpServletRequest request,
                   HttpServletResponse response) throws
        ServletException, IOException {
    RequestDispatcher dispatcher = null;
    HttpSession session = redirectIfSessionNotPresent(
            request, response, dispatcher);
    ShoppingCart cart = getOrCreateShoppingCart(session);
    Stack mementoStack = (Stack) session.getAttribute(
            MEMENTO_STACK_ID);
    if (request.getParameter("bookmark") != null)
        mementoStack = handleBookmark(cart, mementoStack);
    else if (request.getParameter("restore") != null)
        handleRestore(session, cart, mementoStack);
    else
        handleAddItemToCart(request, session, cart);
    if (mementoStack != null && !mementoStack.empty()) {
        request.setAttribute("bookmark", new Boolean(true));
        session.setAttribute(MEMENTO_STACK_ID, mementoStack);
    }
    dispatcher = request.getRequestDispatcher("/ShowCart.jsp");
    dispatcher.forward(request, response);
}
```

The ShowCart controller servlet now has three distinct paths of execution. The first path is the one from the previous version: adding an item to the shopping cart and forwarding to the show cart view. Two additional execution paths have been added. The first allows the user to set a bookmark, and the second allows the user to restore from a bookmark. The path of execution is determined by request parameters that are encoded if the show cart JSP reposts to this page. The body of the `doPost()` method checks for these request parameters and routes control appropriately.

The `handleBookmark()` method (listing 13.24) is invoked if the user has decided that he or she wants to bookmark the shopping cart.

Listing 13.24 This method handles generating a bookmark and saving it.

```
private Stack handleBookmark(ShoppingCart cart,
                             Stack mementoStack) {
    if (mementoStack == null) {
        mementoStack = new Stack();
    }
    mementoStack.push(cart.setBookmark());
    return mementoStack;
}
```

The `handleBookmark()` method checks to see if a stack already exists; if not, it creates one. In either case, it generates a new `Memento` from the `cart` object and pushes it onto the stack. The symmetrical `handleRestore()` method (listing 13.25) does the opposite—it pops the `Memento` off the stack and restores the `cart` contents.

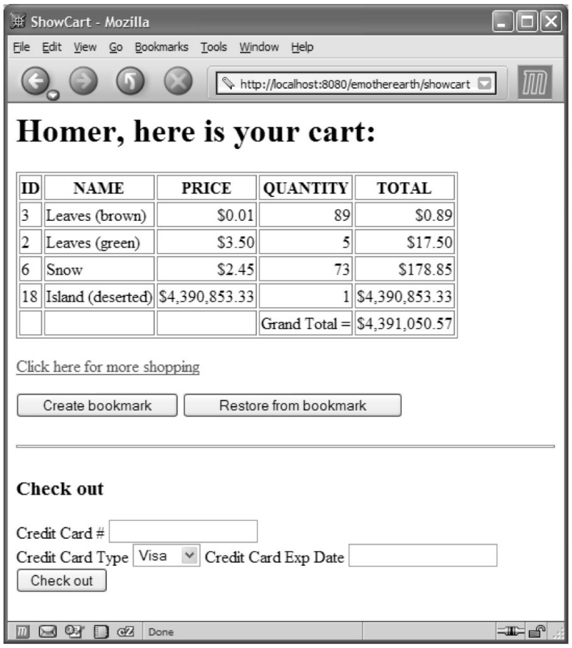
Listing 13.25 The handleRestore() method restores the state back to the most recent bookmark.

```
private void handleRestore(HttpSession session,
                           ShoppingCart cart,
                           Stack mementoStack) {
    if (mementoStack == null)
            return;
    cart.restoreFromBookmark(
            (ShoppingCart.ShoppingCartMemento)
            mementoStack.pop());
    if (mementoStack.empty()) {
        session.removeAttribute(MEMENTO_STACK_ID);
    }
}
```

The `handleRestore()` method also removes the `Memento` stack from the session if the stack is empty, effectively relieving the session from its caretaker role.

The user interface for the shopping cart must change marginally to provide the user with a way to create and restore bookmarks. To that end, we've added a Create Bookmark button and, in case the `Memento` stack exists, we've added a Restore From Bookmark button as well. The updated user interface appears in figure 13.8.

The last portion of the controller servlet that manages bookmarks appears near the bottom of the `doPost()` method. It checks to see if a `Memento` stack exists


ShowCart - Mozilla
File Edit View Go Bookmarks Tools Window Help
http://localhost:8080/emotherearth/showcart
Done

Homer, here is your cart:

ID	NAME	PRICE	QUANTITY	TOTAL
3	Leaves (brown)	$0.01	89	$0.89
2	Leaves (green)	$3.50	5	$17.50
6	Snow	$2.45	73	$178.85
18	Island (deserted)	$4,390,853.33	1	$4,390,853.33
			Grand Total =	$4,391,050.57

Click here for more shopping

Create bookmark | Restore from bookmark

Check out

Credit Card #
Credit Card Type Visa Credit Card Exp Date
Check out

Figure 13.8
The user interface for the ShowCart page now incorporates buttons for managing bookmarks.

and, if it does, it adds a request parameter as a flag to the view to create the Restore button. It also updates the session with the current `Memento` stack.

The user interface JSP checks to see if the request parameter is available and shows the Restore button if it is. The updated portion of the ShowCart JSP appears in listing 13.26.

Listing 13.26 The ShowCart JSP must check to see if restoring from a bookmark should be presented as an option.

```
<form action="showcart" method="post">
<input type="submit" name="bookmark" value="Create bookmark">
<%
    if (request.getAttribute("bookmark") != null) {
%>
<input type="submit" name="restore" value="Restore from bookmark">
<%
    }
%>
</form>
```

The user interface currently does not provide any visual feedback indicating which records appear at each bookmark marker. It is certainly possible to color-code the records or provide some other indication of the bookmark boundaries.

As with other user interface techniques in Model 2 applications, most of the work appears in the model and controller, with supporting elements in the JSP. Undo using the Memento design pattern is fairly easy to implement in web applications because of the ready availability of the session, which is an ideal caretaker. The use of inner classes helps achieve the original intent of the pattern, exposing the inner workings of the `Originator` only enough to enable the snapshot and restoration through the `Memento`.

13.2.3 Undo in frameworks

Because most of the activity in building undo with transaction processing appears in the boundary classes, it is easy to add it to the Model 2 frameworks. InternetBeans Express also facilitates this type of undo operation because the data-aware components are transaction aware. Thus, adding transaction processing to that framework is even simpler (it consists of setting a property).

Using Memento is also easy in Model 2 frameworks. For the lighter-weight ones, the same pattern of code that appears in the previous section works because they all support the standard web APIs, like `HttpSession`. The other medium-to-heavyweight frameworks also support using Memento, albeit with different mechanisms for the caretaker. In Tapestry, the caretaker moves to the `Visit` object, which is available to all the pages. In WebWork, it moves to WebWork's own session object, which is similar in intent but different in implementation to the standard `HttpSession`. Cocoon supports `HttpSession`, so no change is necessary.

13.3 Using exception handling

Java developers are familiar with exception handling and how exception-handling syntax works in the language, so I won't rehash that material here. However, many developers are reluctant to create their own exception classes. It is also important to distinguish between fundamental types of exceptions.

13.3.1 The difference between technical and domain exceptions

The Java libraries define a hierarchy of exception classes, starting with `Throwable` at the top of the tree. Most methods in libraries in Java throw exceptions tuned to the kinds of potential problems in that method. All these exceptions fall into the broad category of *technical exceptions*. A technical exception is one that is raised for some technical reason, generally indicating that something is broken from an infrastructure level. Technical exceptions are related to the area of how you are building the application, not why. Examples of technical exceptions are `ClassNotFoundException`, `NullPointerException`, `SQLException`, and a host of others. Technical exceptions come from the Java libraries or from libraries created by other developers. Frequently, if you use a framework developed by others, they have included technical exceptions in their methods to indicate that something is either broken or potentially broken.

Domain exceptions are exceptions that relate to the problem domain you are writing the application around. These exceptions have more to do with a business rule violation than something broken. Examples of domain exceptions include `ValidationException`, `InvalidBalanceException`, `NoNullNameException`, and any other exception you create to signify that some part of the application is violating its intended use. Domain exceptions are ones you create yourself and use within the application to help with the application flow.

13.3.2 Creating custom exception classes

Java makes it easy to create your own exception classes. At a minimum, you can subclass the `Exception` class and provide your own constructor that chains back to the superclass constructor. Listing 13.27 shows an example of such a lightweight exception class.

Listing 13.27 A custom exception that provides a new child of the Exception class

```
public class InvalidCreditCardNumberException extends Exception {
    public InvalidCreditCardNumber(String msg) {
        super(msg);
    }
}
```

Instead of creating a lightweight class like this, it is possible to generate a new `Exception` object and pass the error message in it:

```
throw new Exception("Invalid Credit Card Number");
```

The problem with this approach is not the generation of the exception but the handling of it. If you throw a generic exception, the only way to catch it is with a catch block for the `Exception` class. It will catch your exception, but it will also catch every other exception that subclasses `Exception`, which encompasses most of the exceptions in Java. You are better off creating your own exception subclasses to handle specific problems. There is no penalty for creating lots of classes in Java, so you shouldn't scrimp on exception classes.

If you extend `Exception`, you must provide a `throws` clause in any method where your exception might propagate. Checked exceptions and the mandated `throws` clause are actually one of the better safety features of the Java language because they prevent developers from delaying writing exception-handling code. This type of code isn't glamorous, so many developers like to put it off or avoid it. Other languages (such as C++) make it all too easy to do this. The checked exception mechanism in Java forces developers to handle exceptions where they occur and deal with them. Often, developers will say something like, "I know I should have some error-handling code here—I'll come back later and add it." But "later" never comes because one rarely has the luxury of extra time at the end of a project.

If you feel you must short-circuit the propagation mechanism in Java (and occasionally there are legitimate reasons for doing so), you can create your exception to subclass `RuntimeException` instead of `Exception`. `RuntimeException` is the parent class for all unchecked exceptions in Java, such as `NullPointerException`, `ArrayIndexOutOfBoundsException`, and many more. The semantic distinction between `Exception` and `RuntimeException` lies with their intended use. `RuntimeException` and its subclasses are bugs lying in the code, waiting for repair. They are unchecked because the developer should correct the code and the application cannot reasonably handle them. While it is possible to create domain exceptions based on `RuntimeException`, it is not recommended. `RuntimeExceptions` represent a flaw in the infrastructure code of the application and shouldn't mix with domain exceptions. Forcing developers to handle checked domain exceptions is not a burden but an opportunity afforded by the language to make your code more robust.

13.3.3 Where to catch and handle exceptions

It is impossible to generalize too much about where exceptions occur and are handled in Model 2 applications. Entities typically throw domain exceptions; boundary classes and other infrastructure classes tend to throw technical exceptions. In both cases, the controller is usually where the exception is handled. For example, in the eMotherEarth application, each boundary class must have a reference to the database connection pool. If they don't, they throw an exception. For this purpose, a `PoolNotSetException` class resides in the project (listing 13.28).

Listing 13.28 This custom exception class is thrown when the pool property isn't set on one of the boundary classes.

```
package com.nealford.art.emotherearth.util;

public class PoolNotSetException extends RuntimeException {
    private static final String STANDARD_EXCEPTION_MESSAGE =
            "Pool property not set";

    public PoolNotSetException(String msg) {
        super(STANDARD_EXCEPTION_MESSAGE + ":" + msg);
    }
}
```

The custom exception class in listing 13.28 extends `RuntimeException` to prevent it from cluttering up controller code by forcing an exception catch. It also contains a predefined message, to which the users of this exception can add as they generate the exception. This exception is used in the `ProductDb` boundary class:

```
if (dbPool == null) {
    throw new PoolNotSetException("ProductDB.getProductList()");
}
```

Rethrowing exceptions

Often, you are writing low-level library code that is called from many layers up by application code. For example, if you are writing a `Comparator` class to make it easy to sort within a boundary object, you have no idea what type of application (desktop, web, distributed, etc.) will ultimately use your code. You must handle an exception, but you don't really know the proper way to handle it within the method you are writing. In these cases, you can catch the checked exception and rethrow it as another kind, either as a `RuntimeException` or as a custom domain exception. An example of this technique appears in the `getProductList()` method (listing 13.29) of the `ProductDb` boundary class.

Listing 13.29 The getProductList() method rethrows a SQLException rather than handling it.

```
public List getProductList() {
    if (dbPool == null) {
        throw new PoolNotSetException(
                "ProductDB.getProductList()");
    }
    if (productList.isEmpty()) {
        Connection c = null;
        Statement s = null;
        ResultSet resultSet = null;
        try {
            c = dbPool.getConnection();
            s = c.createStatement();
            resultSet = s.executeQuery(SQL_ALL_PRODUCTS);
            while (resultSet.next()) {
                Product p = new Product();
                p.setId(resultSet.getInt("ID"));
                p.setName(resultSet.getString("NAME"));
                p.setPrice(resultSet.getDouble("PRICE"));
                productList.add(p);
            }
        } catch (SQLException sqlx) {
            throw new RuntimeException(sqlx.getMessage());    <-- Rethrows an exception
        } finally {
            try {
                dbPool.release(c);
                resultSet.close();
                s.close();
            } catch (SQLException ignored) {
            }
        }

    }
    return productList;
}
```

Empty catch blocks

One of the frowned-upon tendencies in some Java developers is to create empty catch blocks to get code to compile. This is a bad thing because now the checked exception is raised and swallowed, and the application continues (or tries to continue) to run. Usually, the application will break in a totally unrelated place, making it difficult to track down the original error. For this reason, empty catch blocks are discouraged.

However, there is one situation where they make sense. If you look at the end of listing 13.29, the database code must close the statement and result set in the `finally` block. Both the `close()` methods throw checked `SQLExceptions`. In this case, as you are cleaning up, the worst thing that can happen is that the statement has already closed. In this case, it makes sense to include an empty catch block. To keep from having to write a comment to the effect of “I’m not lazy—this catch block intentionally left blank,” name the instance variable in the catch block `ignored`. This is a self-documenting technique that keeps you from having to document it because it is documented by the variable name.

Redirecting to an error JSP

One of the nice automatic facilities in JSP is the ability to flag a page as the generic error page for the application. If any unhandled exceptions occur from other JSPs, the user is automatically redirected to the error page specified at the top of the source page. The error page has access to a special implicit exception object so that it can display a reasonable error message.

When you're building Model 2 applications, the controller won't automatically forward to an error page if something goes wrong. However, you can still forward to the error page yourself and take advantage of the implicit exception object. Before you forward to the error page, you can add the exception with a particular name that the error page is expecting. The CheckOut controller in eMotherEarth handles an insertion error by redirecting to the JSP error page. See this code in listing 13.30.

Listing 13.30 The CheckOut controller forwards to the JSP error page to inform the user that an exception occurred.

```
try {
    orderDb.addOrder(sc, user, order);
} catch (SQLException sqlx) {
    request.setAttribute(
        "javax.servlet.jsp.jspException", sqlx);
    dispatcher = request.getRequestDispatcher("/SQLErrorPage.jsp");
    dispatcher.forward(request, response);
    return;
}
```

The JSP error page looks for a request attribute named `javax.servlet.jsp.jspException` to populate the implicit exception object. The destination page has no idea if the JSP runtime or the developer added this attribute. This approach allows you to consolidate generic error handling across the application. If you want more control over the application-wide exception handling, you can write your own controller/view pair to handle exceptions generically.

13.3.4 Exceptions in frameworks

The Model 2 frameworks' exception-handling code generally follows the guidelines we stated earlier. Entities typically throw domain exceptions, and boundary classes and other infrastructure classes typically throw technical exceptions. In both cases, the controller is where the exception is handled. The frameworks themselves frequently throw exceptions, which fall under the category of technical exceptions. These exceptions are best handled in the controller or controller proxy classes (i.e., an `Action` class).

Handling exceptions in the two frameworks that try to mimic the event-driven nature of desktop applications is more difficult. An exception in a desktop application represents a state, and the propagation depends on the current call stack. It is much harder to emulate this call stack state in a web application, because the user always sees a fully unwound call stack. Tapestry has good mechanisms in place for both mimicking event-driven behaviors and handling exceptions. InternetBeans Express makes artificial exception state management more difficult because it uses a thinner veneer over the components it uses.

13.4 Summary

Users tend to request features in web applications that they have seen in desktop or other web applications. Many of these requests relate to the flow of information in the application. Building usable web applications in Model 2 applications generally touch all three moving parts: the controller, the model, and the view. These three pieces work together to provide an attractive application.

The flexibility of Model 2 applications makes it easy to implement even the most complex user requirements. Keeping the application well partitioned and the parts separate requires diligent effort, but it pays off in the long run with easy-to-maintain and scalable applications.

In the next chapter, we look at performance in web applications and how to measure and improve it.

JAVA

ART OF JAVA WEB DEVELOPMENT

STRUTS, TAPESTRY, COMMONS, VELOCITY, JUNIT, AXIS, COCOON, INTERNETBEANS, WEBWORK

Neal Ford

So you've mastered servlets, JSPs, statelessness, and the other foundational concepts of Java web development. Now it's time to raise your productivity to the next level and tackle frameworks. Frameworks—like Struts, Tapestry, WebWork, and others—are class libraries of prebuilt parts that all web applications need, so they will give you a huge leg up. But first you'll need a solid understanding of how web apps are designed and the practical techniques for the most common tasks such as unit testing, caching, pooling, performance tuning, and more.

Let this book be your guide! Its author, an experienced architect, designer, and developer of large-scale applications, has selected a core set of areas you will need to understand to do state-of-the-art web development. You will learn about the architecture and use of six popular frameworks, some of which are under-documented. You will benefit from a certain synergy in the book's simultaneous coverage of both the conceptual and the concrete, like the fundamental Model 2 design pattern along with the details of frameworks, the how-tos of workflow, the innards of validation, and much more. In this book, combining the general and the specific is a deep *and* useful way to learn, even for those who have not used a framework before.

What's Inside

- Web frameworks analyzed
- How to incorporate Web services
- How-tos of
    - caching
    - pooling
    - workflow
    - validation
    - testing

Neal Ford is an architect, designer, and developer of applications, instructional materials, books, magazine articles, video presentations, and a speaker at numerous developers' conferences worldwide. He is Chief Technology Officer of The DSW Group, Ltd.

"Great combination of the three levels: patterns, frameworks, and code."

—Shahram Khorsand
NetServ Consulting Sweden

"Covers all facets of web application development.... This book is bold!"

—Eitan Suez
Founder, UpToData Inc.
Creator of DBDoc

"You have two options: read four or five books plus stuff from all over the Net—or read this one."

—Luigi Viggiano, co-founder,
Turin Java Users Group

"I really like what I'm reading ... nice style, very approachable."

—Howard M. Lewis Ship
Creator of Tapestry

MANNING $44.95 US/$67.95 Canada

54495
9 781932 394061
ISBN 1-932394-06-0