How JavaServer Pages Can Use Design Patterns

James W. Cooper

You probably have been reading more and more about JavaServer pages (JSPs) in recent months, and in this column I'll go into more depth on how you can use them. To review, JavaServer pages are an extension to Java Servlets. They run on the web server and can be used to generate dynamic web pages in response to any information that might be available. Most often, JSPs are used to respond to a filled out HTML form which you submit to the web site for processing. Then, rather than writing a CGI script to interpret that form, you use a JSP to do it in a simpler and more elegant fashion.

JSPs operate in conjuction with a non-visual JavaBean, which is simply a class with a bunch of get and set methods for various parameters on your input form. For example, if there is an input field called "name"

```
<input type="text" name="name" size="25">
```

you need to have setName and getName methods within the JavaBean to receive that data. There must be a set*Xxx* method for every named field in the form. There need not be get*Xxx* methods unless you need to retrieve the value of that field back from the bean.

Let's consider the case of a simple input form where you can enter your name either as *firstname lastname* or as *lastname, firstname*. We want to write a server process that can take these two cases apart and return the first and last names reliably. Of course, this is a ridiculously simple program that you could write in JavaScript on the client input page just as well, but we'll use it as a model for more complex code you might want to write in real life.

A Names Factory

We'll start by considering the Bean we have to write. It will decide which order the name is entered in and select one of two classes. We'll start by defining a base Namer class which has getFirst and getLast methods, but makes no decisions on name order:

```
package Names;
//A simple base class for both orders of names
public class Namer {
   //store the names here
  protected String first, last;

  //return the first name
  public String getFirst() {
      return first;
  }
  //return the last name
  public String getLast() {
      return last;
  }
}
```

Then, we derive the FirstFirst and LastFirst classes from it: each having a different constructor. Here is the first name class.

```
package Names;
public class FirstFirst extends Namer {

    //separates the leading first name
    public FirstFirst(String s) {
        int i = s.lastIndexOf (" ");
        if(i > 0) {
            first = s.substring(0, i).trim();
            last = s.substring(i + 1).trim();
        }
        else {
            first = "";
            last = s;
        }
    }
}
```

and here is the analogous LastFirst class:

```
package Names;
public class LastFirst extends Namer {
   //makes ht last name all to the left of the first comma
   public LastFirst(String s) {
        int i = s.lastIndexOf (",");
        if(i > 0) {
            last = s.substring(0, i).trim();
            first = s.substring(i + 1).trim();
        }
        else {
            first = "";
            last = s;
        }
    }
}
```

Now the class that decides which of these classes to invoke is the NamingBean class. Since it chooses one of several related classes, we refer to it as a Factory class:

```
package Names;
public class NamingBean {
    private Namer namer = null;

    //This selects one of two Namer subclasses
    public void setName(String nm) {
        int i =nm.indexOf (",");
        if( i >0 )
            namer = new LastFirst(nm);
        else
            namer = new FirstFirst(nm);
    }
    //return the last name
    public String getLast() {
       return namer.getLast ();
    }
    //return the first name
    public String getFirst() {
       return namer.getFirst ();
    }
```

```
    //get the instance of the Namer class
    public Namer getNamer() {
        return namer;
    }
}
```

The critical part of the NamingBean class is that in the setName method it creates an instance either of FirstFirst or of LastFirst and stored that instance in the variable *namer*. This setName method is called automatically when our form is posted. The getFirst and getLast methods just return the value computed by whichever instance of the two classes currently occupies the *namer* variable. With that in mind, we can create the JSP itself.

The JavaServer Page

This page declares that it uses the NamingBean class and sets up a form to submit with a text field where you can enter a name. Note that there is no *action=* modifier to the form. Instead, it just sends the form values to the bean we specify.

```
<%@ page language="java" import="Names.NamingBean" %>
<jsp:setProperty name="NameBean" property="*"/>

<html>
<body bgcolor="c0c0ff">
<center>
<h2>Enter your name below</h2>

<form method="post">
<input type="text" name="name" size="25">
<br>
<input type="submit" value="Submit">
</form>

<% if (NameBean.getNamer() != null) { %>

<br><b>First name:</b> <%= NameBean.getFirst()%>
<br><b>Last name:</b>  <%= NameBean.getLast()%>

<% } %>

</body>
</html>
```

The bottom half of the JSP consists of an if test to see if the *getNamer* method returns null or not. If it is null, setHName has not yet been called and no value has been entered for a name. If it is not null, the values for the first and last name are printed as part of the form. You can see the form displayed before and after the Submit button is clicked in Figures 1 and 2.

Figure 1 – The JSP before submitting it.

Figure 2 – The JSP after submitting it.

More Class and Less Containment

In the previous example, we used the Factory to generate an instance of one of the two Namer classes and store it inside the bean. This is OK, but it does mean that the bean has to contain the same getLast and getFirst methods that the Namer class does and simply pass them on. It would be better if we returned the actual instance of the Namer class and called its methods directly. This is what we do in the second version of the JSP shown below.

```
<%@ page language="java" import="Names.NamingBean" %>
<jsp:useBean id="NameBean" scope = "page" class="Names.NamingBean" />
<jsp:setProperty name="NameBean" property="*"/>

<html>
<body bgcolor="c0c0ff">
<center>
<h2>Enter your name below</h2>

<form method="post">
<input type="text" name="name" size="25">
<br>
<input type="submit" value="Submit">
</form>

<% if (NameBean.getNamer() != null) {
  Names.Namer nm = NameBean.getNamer();
  out.println("<br><b> First name:</b> "+ nm.getFirst());
  out.println("<br><b> Last name: </b> "+ nm.getLast());
}
%>

</body>
</html>
```

In this version, we use the getNamer method to return an instance of the Namer class and call its methods right in the JSP code. Note that we have to declare the class using the package name prefix and that the Namer class is separate from the NameBean class we used as our Bean. However, we have now used a Factory in the Bean to return one of several classes to our JavaServer Page and have thus developed an example of the flexibility of the JSP system we don't find in other scripting languages, such as ASP.

Accessing Databases

Let's suppose we want to create a web page to look up the grocery store having the lowest prices for various foods. We'll assume that comparison shoppers update the database regularly so we can comparison shop using our web site. In our simple example, we'll create an Access database of foods, stores and prices. Figure 3 shows the Food and Stores tables and Figure 4 shows the relational table between the food, the store and the prices.

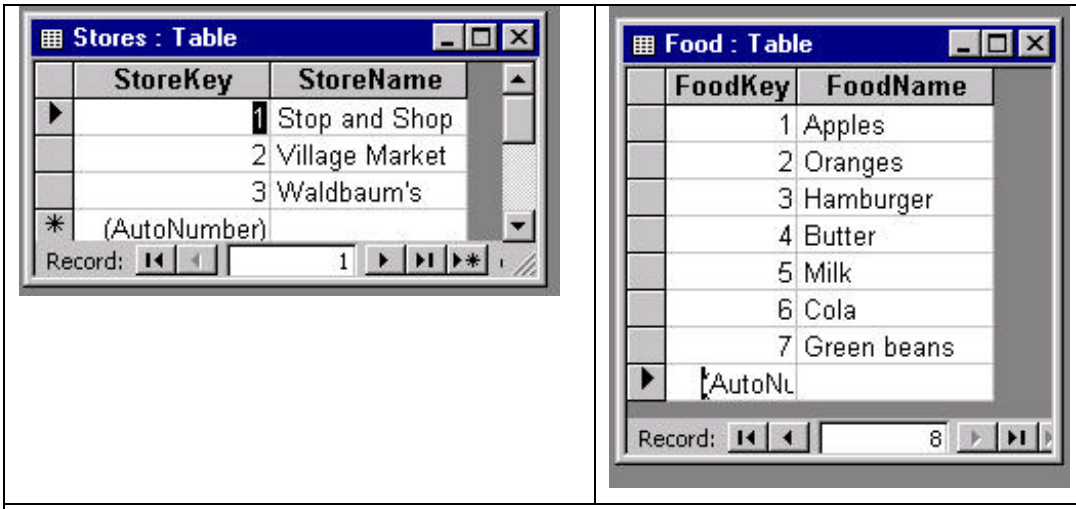
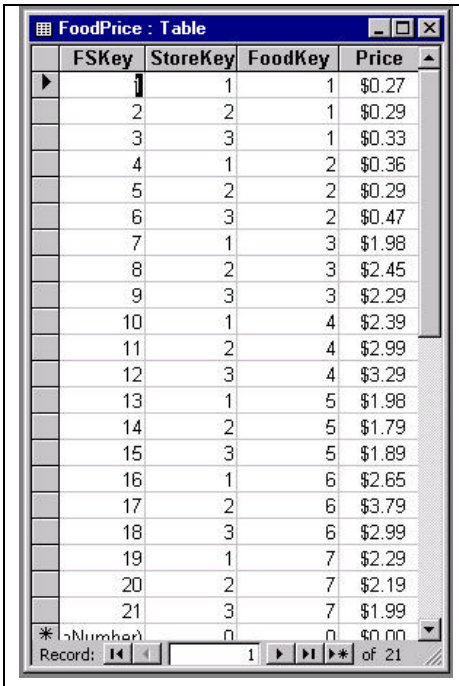


Figure 3 – The Stores and Food database tables



FoodPrice : Table

FSKey	StoreKey	FoodKey	Price
1	1	1	$0.27
2	2	1	$0.29
3	3	1	$0.33
4	1	2	$0.36
5	2	2	$0.29
6	3	2	$0.47
7	1	3	$1.98
8	2	3	$2.45
9	3	3	$2.29
10	1	4	$2.39
11	2	4	$2.99
12	3	4	$3.29
13	1	5	$1.98
14	2	5	$1.79
15	3	5	$1.89
16	1	6	$2.65
17	2	6	$3.79
18	3	6	$2.99
19	1	7	$2.29
20	2	7	$2.19
21	3	7	$1.99
(AutoNumber)	0	0	$0.00

Record: 1 of 21

Figure 4 –The Food-Store-Price relations table.

We can easily write a database query in the SQL language to get the prices for any food in increasing order, but Access allows us to construct the query graphically as shown in Figure 5. We just select the FoodName, StoreName and Price columns from the three tables and put some name in the FoodName Criteria line and run the query using the "!" run button in Access. We can then switch to the SQL view and cut out the SQL and paste it into our Java program.

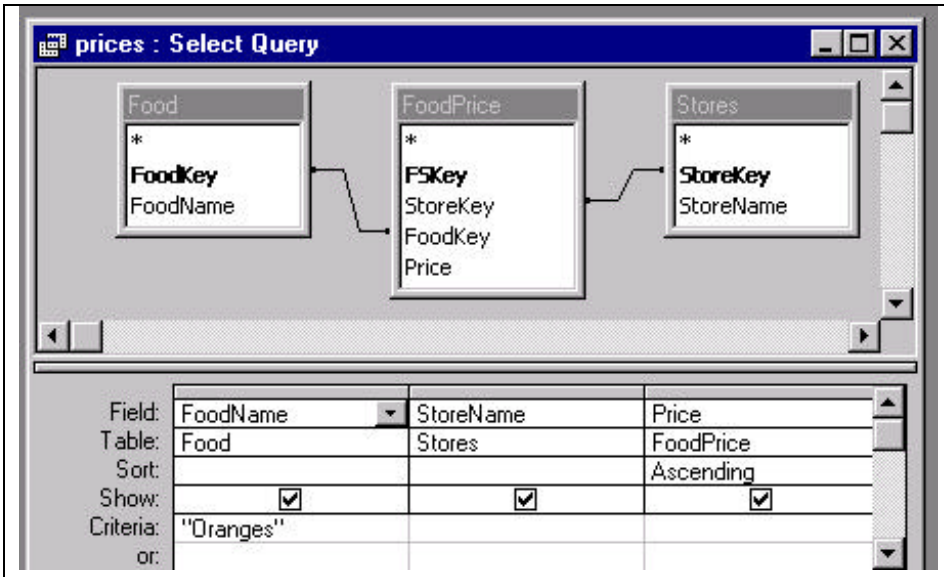


Figure 5 – The graphical query to list the prices of Oranges at all the stores in order of increasing price.

The actual SQL query Access generates is just

```
SELECT DISTINCTROW Food.FoodName, Stores.StoreName, FoodPrice.Price
FROM (Food INNER JOIN FoodPrice ON Food.FoodKey = FoodPrice.FoodKey)
INNER JOIN Stores ON FoodPrice.StoreKey = Stores.StoreKey
WHERE (((Food.FoodName)="Oranges"))
ORDER BY FoodPrice.Price;
```

The Database Bean

Our database bean is fairly simple. We just provide a setFood method and a method to get an Enumeration of the foods in the database.

```
package Groceries;
import java.net.URL;
import java.sql.*;
import java.util.*;
import java.awt.*;
import java.awt.event.*;

public class grocBean {
    Database db;        //database
    Results rs, fdrs;   //result sets
    Vector foods;       //foods listed here
    Enumeration eFood;  //an enumeration of the foods
    boolean queryDone;  //true if query done
    String food;        //current food

    public grocBean() {

        db = new Database("sun.jdbc.odbc.JdbcOdbcDriver");
        db.Open("jdbc:odbc:Grocery prices", null);
        queryDone = false;

    }
    //-----------------------------------
    public void getFoods() {
        System.out.println("getting foods");
        String query = "Select FoodName from Food Order by FoodName";
        fdrs = db.Execute (query);

        Vector foods =  new Vector();
        while (fdrs.hasMoreElements ()) {
            foods.add((String)fdrs.getColumnValue ("FoodName"));
        }
        eFood= foods.elements();
     }
    public boolean hasMoreFoods() {
        boolean hasMore = eFood.hasMoreElements ();
        if(! hasMore)
            fdrs.close ();
        return hasMore;
    }

    public String nextFood() {
        return (String)eFood.nextElement ();
    }
```

```
    //-----------------------------------
public void setFood(String  myfood) {
 food = myfood;
String queryText =
"SELECT DISTINCTROW FoodName, StoreName, Price " +
"FROM (Food INNER JOIN FoodPrice ON Food.FoodKey =FoodPrice.FoodKey) "+
"INNER JOIN Stores ON FoodPrice.StoreKey = Stores.StoreKey " +
"WHERE (((Food.FoodName)=\'" + food + "\')) ORDER BY FoodPrice.Price;";

       rs = db.Execute(queryText);
       queryDone = true;
    }
    //-----------------------------------
    public String getFood() {
        return food;
    }
    public boolean hasData() {
        return queryDone;
    }
    public String getPrice() {
        return rs.getColumnValue("Price");
    }
    public String getStore() {
     return rs.getColumnValue ("StoreName");
    }
    public void nextElement () {
        rs.nextElement ();
    }
    public boolean hasMoreElements() {
        boolean hasMore = false;
        if(rs != null)          {
            hasMore = rs.hasMoreElements ();
            if(! hasMore)
                rs.close ();
        }
        else
            hasMore = false;
        return hasMore;
    }
}
```

You may notice that we aren't using the lower level java.sql.* classes directly. Instead we have created a couple of higher level classes called Database and Results which we discussed in a column last year. Wrapping these classes in a simpler outer class is an example of the Façade pattern and simplifies our programming greatly.

The JSP for this query is just as simple as you might think. We enumerate the foods in the database and store them in a list box. When we submit the query, we detect which food is selected and return a table of the prices as various local stores.

```
<%@ page language="java" import="Groceries.grocBean" %>
<jsp:useBean id="grocBean" scope = "page" class="Groceries.grocBean" />
<jsp:setProperty name="grocBean" property="*"/>

<html>
```

```
<head><title>Grocery Prices</title></head>
<body bgcolor = "80ffff">
<!-- -----------------  -->
<center>
<form method = "post">
 <select name ="food" size = "10">
   <% grocBean.getFoods();
   while (grocBean.hasMoreFoods()) { %>
        <option> <%= grocBean.nextFood() %>
   <% } %>

 </select>
 <br>
 <input type=submit value="Submit">
</form>

<% if (grocBean.hasData()) { %>
  <h2> Prices of <%= grocBean.getFood() %> </h2>
  <table>
bbbbbbbbb<% } %>

<% while(grocBean.hasMoreElements()) { %>
    <tr>
    <td> <%= grocBean.getStore() %></td><td> <%= grocBean.getPrice() %>
</td>
    </tr>

<%}%>
  </table>
</center>

</body>
</html>
```

We show the results of a simple query in Figure 6.

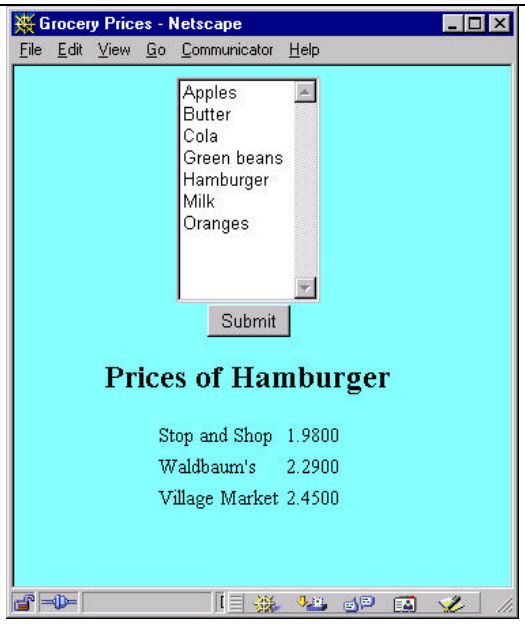


Figure 6 – The results of querying the database for Hamburger prices.

Trying out these examples

If you install the JSP JDK, you will have a tree that contains

```
Jswdk-1.0\examples\jsp
```

along with a whole lot of little directories underneath for each of the provided examples. Unzip the jsp code into a new "GrocPrices" and "Names" folder at this level.

The Java bean goes into the hierarchy

```
Jswdk-1.0\examples\Web-inf\jsp\beans
```

Create a folder called "Groceries" and "Names" and unzip the beans into it. You will also need to register the groceries.mdb file with ODBC and name it "Grocery Prices." Since this simple example is an Access database, this code will only run on Windows.

Conclusions

We've shown some more powerful uses of JavaServer Pages in this column. We showed how you can use a Factory to return a class to the clent page and use it to generate the final page results. Then we showed how you can build and access a database and display grocery prices on the client page with very little effort.