Ch 6: Model View Controller

- Overview
- MVC Basics
- JDBC
- A Case Study: On-Line Shopping Application
  - Introduce concepts of MVC design architecture
  - Demonstrate the Java implementation of MVC architecture
- Introduce concepts of JDBC
- Provide step by step tutorials on MVC design and development
Overview

The Model-View-Controller (MVC) object-oriented architecture originally came from Smalltalk-80 as a methodology to separate user interface presentation from the underlying data. The MVC is very similar to the Presentation-Abstraction-Control (PAC) model. The purpose of MVC is to decompose the whole system into three subsystems (modules) that are Model, View, and Controller. It is also called a component-based architectural style because each module can be implemented by software components: data components, presentation components, input controls, control dispatch, and business process components. Each module in the MVC architecture has its own responsibility. Project team members with different expertise can work more efficiently in their own areas. For example, graphic professionals work on presentation of GUI interface module, programmer professionals work on input processing such as authentication, flow logic, job dispatching in the Controller module, and data processing and data base professionals can focus on the Model module to provide all the data the Web application needs.
The connections between modules are also well defined in MVC. The Controller takes the inputs and does the authorization and authentication checking, dispatches incoming requests to the corresponding module or sub-module in Model by instantiating new instances of data sources in Model module and calling the methods provided by the data source objects. The data source forwards the controls to a specific representation module and lets that module be in charge of rendering the data retrieved from the data source objects. The data-Model module may generate events to notify the Model listeners when the data is changed. The View module listens to the data-Model module as its event listener. When such an event occurs the View module needs to update its presentation views.
Since the MVC architecture is object oriented, each module may consist of number of components. MVC does not restrict an application to single view and controller. Typically for Java technology, most designers would like to have multiple JSP pages in a View module, multiple JavaBeans in a Model module, and necessary number of Servlet classes in a Controller module. Generally, one JavaBean needs to have a corresponding data base table to support it. We will discuss Java Data Base Connectivity (JDBC) topic in this chapter to show how the database supports the JavaBean in MVC. Since we have already introduced Java Servlet, JSP, and JavaBean technologies, this is a good time for us to explore the Java implementation of MVC architecture.
MVC Module-1 architecture combines Controller and View together into a single module to take care of input and output processing; the rest of the tasks are handled in Data-Model module. In a Java implementation, the front module is implemented using JSP technology and back-end module is implemented with JavaBean technology. JSP on the Web server accepts requests, instantiates the required JavaBeans and generates HTML pages for response. We know that each JavaBean object will usually have a relational table to support it.
Figure 5.1 Overall MVC-1 Architecture

web browser

1 request

JSP ‘s (controller & view)

2

javaBeans (model)

3

db

4 response
Figure 5.2 Java MVC-1 Architecture of the Simple Example
Here is a simple MVC-1 Web application example implemented by JSP and JavaBeans.

There are two JSP pages in the front-module of this MVC-1 architecture: *mvc-1.jsp* and *hello.jsp*. As we discussed before, there may be many components working together in a single module. There is one JavaBean in the data module; that is the *myBean* JavaBean class. Let’s look at the front module first.
MVC Type-1 Architecture (cont.)

```jsp
<%@ page import= "myPackage.*" %>
<jsp:useBean id="myBean" class="myPackage.MyBean"
    scope="request"/>
<jsp:setProperty name="myBean" property="*"/>
<html>
<body>
<form method="get">
<input type="text" value="userName">
<input type="submit" value="Submit">
</form>
<%   if ${param['username']} != null) { %>
<%@ include file = "hello.jsp" %>
<%
}
<%@ include file = "hello.jsp" %>
</body>
</html>
```
This module first creates a new instance of `myPackage.MyBean` JavaBean class with the `<jsp:useBean>` action tag. Then it takes user input from the request form and assigns the input value to the bean property by the `<jsp:setProperty name="myBean" property="*"/>` action tag. If there is an input string rather than nothing it will include another page `hello.jsp` to render the output. The `hello.jsp` page shown below simply uses an expression language notation `${myBean.userName}` to get the data from the JavaBean in the end-module where the data is set by the first `mvc-1.jsp` page. It is clear that the front-end module of MVC-1 does both input processing and result presentation and back-end module does all the business logic and data processing.
MVC Type-1 Architecture (cont..)

hellop.jsp

<b>
    Hello ${myBean.userName}!
</b>

The following <i>MyBean.java</i> shows a JavaBean declaration which complies with the JavaBean convention. Every JavaBean has a default public constructor; it has all its properties in private; it provides public methods to read from or write to its private properties; the names of methods meet the convention pattern in the formats of getXXX() and setXXX() where XXX is the name of its property; it must implement the serializable interface to have its persistent state. The following fragment shows the MyBean JavaBean used in this example.
MVC Type-1 Architecture (cont.)

MyBean.java

package myPackage;
import java.io.*;

public class MyBean implements serializable {
    private String userName;

    public void MyBean(){
        userName= null;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }

    public String getUserName() {
        return this.userName;
    }
}
MVC Type-2 Architecture (cont.)

The MVC type-2 architecture is a better fit for more complex Web application design. It has a dedicated controller module which is in charge of user request processing such as authorization and authentication, deciding flow control dispatching such as selection of presentation views, as well as selection and instantiation of data models. The controller module is programming-centric oriented since there may be complex logic controls in a large application. There may be many classes working together in the controller module. Java Servlet is a typical technology used in controller module for processing-intensive tasks. There is no processing logic in the presentation view module. The view module is only responsible for retrieving any data objects that are created by the Servelts and generating dynamic contents to clients. There may be many view pages in a view module. The view module is page-centric oriented. The clean separation of presentation from data processing and request processing results in clear division of the roles and responsibilities of developers and page graphics designers in a development team. The more complex the application is, the more benefits the MVC type-2 architecture will bring in.
MVC Type-2 Architecture (cont.)

The following diagram shows a typical MVC type-2 architecture used in the middle-tier (Web server) of a Web application.

1. web browser \(\rightarrow\) request
2. request \(\rightarrow\) Servlet (Controller)
3. forward \(\rightarrow\) JSP' (View)
4. JDBC \(\rightarrow\) db
5. response \(\leftarrow\) web browser
A Simple MVC-2 example

The following example illustrates a simple implementation of an MVC architecture where there is only one Java class in each of the three modules in the MVC architecture. The MyBean JavaBean class plays the role of model, MyServlet Servlet class plays the role of controller, and the fromServlet JSP plays a role of view in the MVC architecture. Figure 5.4 shows the architecture diagram of this Web application. This example emphasizes the MVC so we omit the user input interfaces. The myServlet Servlet set a username and stores this name in a JavaBean named myBean, then transfers the control to a JSP page named fromServlet.jsp which retrieves the username from the myBean and displays on a Web page.
A simple example of MVC architecture

```
myServlet.java

set userName

forward

display userName

fromServlet.jsp

Create Bean
Store userName in Bean

JavaBean

userName

MyBean.java

get userName
```
MyBean is a JavaBean class responsible for storing and providing data for business processing. This data JavaBean has one userName private property and two public methods to read from and write to this userName property.

MyBean.java
package myPackage;
public class MyBean {
    private String userName;
    public MyBean(){
        username="";
    }
    public void setUserName(String userName) {
        this.userName = userName;
    }
    public String getUserName() {
        return this.userName;
    }
}
MVC Type-2 Architecture (cont.)

The MyServlet Servlet class in the controller module sets the userName property of myBean by hard coding, stores this bean as an attribute (beanInfo) of the session implicit object and dispatches the control to fromServlet.jsp of the view module in MVC architecture.

MyServlet.java
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import myPackage.MyBean;
public class MyServlet extends HttpServlet {
    public void service(HttpServletRequest request,
                         HttpServletResponse response)
            throws ServletException, IOException {

        MyBean myBean = new MyBean();
        myBean.setUserName("Kai");
        HttpSession session = request.getSession();
        session.setAttribute("beanInfo", myBean);
        RequestDispatcher rd;
        rd = getServletContext().getRequestDispatcher
            ("/fromServlet.jsp");
        rd.forward(request, response);
    }
}

The *fromServlet.jsp* in the view module just retrieves userName property stored in the myBean with the beanInfo id and displays userName on the resulting page.

```jsp
<jsp:useBean id="beanInfo"
    class="myPackage.MyBean" scope="session"/>

<html>
<body>
    <b>
        Hello <jsp:getProperty name="beanInfo"
            property="userName"/>
    </b>
</body>
</html>
```

Or you can use JSP EL notation in the JSP file as follows.

```jsp
<html>
<body>
    <b>
        Hello ${beanInfo.userName}!
    </b>
</body>
</html>
```
Java DataBase Connectivity (JDBC) is a standard that describes how to connect to and talk to a database from within a Java application or applet. JDBC is a layer of abstraction that allows users to choose between databases. It provides cross-DBMS connectivity to a wide range of SQL databases. It allows you to access virtually any SQL database engine with the single JDBC API. JDBC allows you to write database applications as Java applications, Java Applets, Servlets, or EJB without having to concern yourself with the underlying details of a particular database.
The JDBC API provides Java programmers with a uniform interface to a wide range of relational databases. All Java database related classes and interfaces are packaged together in the API package of `java.sql` which includes both the JDBC interfaces and the JDBC driver manager.

A JDBC driver is a set of Java classes that implement the JDBC interfaces and translate vendor-neutral JDBC calls to vendor-specific DBMS commands so that JDBC can talk to any specific SQL database. JDBC drivers are provided by database vendors. A JDBC driver can be pre-installed by downloading. The JDBC driver must register with the JDBC driver manager. JDBC driver manager will connect to a given database upon requests from Java application or Java applet.
The figure 5.5 shows the connection of Java to a database by JDBC drivers.
JDBC API

The JDBC API provides a set of interfaces and classes for Java programmers to make a connection to a given database via the JDBC driver manager, prepare a SQL statement, make the query or update requests by the prepared SQL statement, get the results and save them in the `resultSet`, and go over the data in the `resultSet`.

We can summarize any JDBC operation into the following steps:

1. Connection
   Load and Register the driver
   - `Class.forName("oracle.jdbc.driver.OracleDriver");` or
     `DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());`

   Connect to the database
   - `Connection conn =
     DriverManager.getConnection("jdbc.oracle:thin@host.port:dbname", "Scott", "Tiger");`

   // thin driver is a pure java driver which can be downloaded for a Java Applet to use.
   // `jdbc.oracle:thin` is the URL for this JDBC connection
JDBC API (cont.)

2. SQL Statement

- Prepare a SQL statement (query or update)

  ```java
  Statement st = conn.createStatement();
  ```

- Access SQL database by execute the SQL statement

  ```java
  ResultSet rs = st.executeQuery(sql-query-statement); or
  Int count = st.executeUpdate(sql-update-statement);
  ```

For example, the sql-query-statement may look like:

  ```sql
  select * from customers;
  ```
3. Processing of returned results

- Obtain metadata

```java
ResultSetMetaData md = rs.getMetaData();
String colName = md.getColumnName(i);
```

- Navigate the query resultSet

```java
while(rs.next()){
    String name = rs.getString(colName);
    ...}
```
JDBC API (cont.)

By default the result set cursor points to the row before the first row of the result set. A call to `next()` gets the first result set row. If there are no more rows unchecked `next()` will return a false value. You can navigate the cursor by calling one of the following ResultSet methods:

- `beforeFirst()` : Default position. Puts cursor before the first row of the result set.
- `first()` : positions the cursor on the first row of the result set.
- `last()` : positions the cursor before the last row of the result set.
- `afterLast()` : positions the cursor beyond last row of the result set.
- `absolute(pos)` : positions the cursor at the row number position where `absolute(1)` is the first row and `absolute(-1)` is the last row.
- `relative(pos)` : positions the cursor at a row relative to its current position where `relative(1)` moves row cursor one row forward.

4. Close database connection
   - Close the connection to release the resource
     
     `Conn.close();`
JDBC Drivers

A JDBC driver is a program which processes JDBC calls from any Java application and translates them to fit specific requirements of certain data sources. JDBC drivers simplify the Java database access activities just like any other type drivers such as printer drivers, network drivers, scanner drivers, and so on. The JDBC drivers themselves may be written purely in Java or partially in Java, db independent or db dependent, network protocol specific or non-network protocol specific, fat or thin, Java Applet supported or Java application only.
Type-1 JDBC Driver

The JDBC-ODBC bridge type driver translates Java JDBC code to ODBC binary code (C++) which in turn translates it into DBMS commands for the target database. This type driver must be installed at client site and can not be downloaded via network since it is not written purely in Java. This driver is not very efficient because of the overhead of this additional layer. Only Java applications not Java Applets can use this type of driver. If there is an ODBC driver available at client site this driver can be a good choice since ODBC driver is an open database connection driver to many vendor databases. The type-1 driver lets you access any ODBC data source from a Java application, but not from a Java Applet. The name of the driver class is:

sun.jdbc.odbc.JdbcOdbcDriver. In Figure 5.5 you can see that a JDBC/ODBC driver takes JDBC calls and translates them to ODBC calls., an ODBC driver (supports many vendor databases) in turn translates the ODBC calls to database specific calls.
Type-1 JDBC Driver

Java Application

JDBC API

Java/C++

JDBC/ODBC Bridge Driver

db independent
Locally installed
Not pure java

ODBC calls

C++

ODBC Driver

db

DBMS1 commands

DBMSn commands
Type-2 JDBC Driver

Java Application

JDBC API

Java/C++

JDBC/Native Driver

db independent
Locally installed
Not pure java
Fat

Native DBMS calls

Proprietary DBMS

Driver

DBMS commands

db
Type-2 JDBC Driver (cont.)

The native-API partly-Java driver type uses Java code to call native (non-Java) code to a DBMS-specific API, `jdbc:oracle:oci` for `oracle.jdbc.driver.OracleDriver` is an example of such driver that Oracle provides. The vertical dash line indicates the division of client site and server site.

This type driver can make use of existing DBMS-specific drivers; therefore it is more efficient than a type-1 driver. Due to the use of native code on the client, and because the driver is DBMS specific a Java Applet cannot use this type driver. This type also requires prior installation of client software. Compared with type-4 driver this type driver is called a fat driver.
Type-3 JDBC Driver

This net-protocol pure Java driver type translates JDBC calls into a DBMS-independent network protocol that is then translated to a specific DBMS protocol by the middleware.

This driver type uses pure-Java client software; it can therefore be downloaded automatically by Java Applets, the client driver is DBMS independent; but it needs a middle tier support. This type diver is similar to a server-side driver since part of it resides on a middle-ware server as shown in figure
Type-4 JDBC Driver

The DBMS-protocol pure-Java driver type translates JDBC calls into a network protocol that is used directly by the DBMS.

The advantage of this type is that it uses pure-Java client software; it can therefore be downloaded automatically by Java applets; this type driver is platform independent and it is a thin driver, `jdbc:oracle:thin` for `oracle.jdbc.driver.OracleDriver`. The disadvantage is that the driver is vendor specific.
MySQL database management system was released in January 1998. It is an open source relational database management system (RDBMS). MySQL is free, and may be downloaded from at MySQL. The current version is 5.0. After you get the MySQL binary distribution download, you can use WinZip command to unzip the .zip archive and "tar" unix command to uncompress the .tar. archive. Once you extracted the distribution archive, you can install the JDBC mysql driver by placing mysql-connector-java-[version]-bin.jar with its full path in your classpath environment variable. If you use the this driver with the JDBC DriverManager, the class name is "com.mysql.jdbc.Driver". In this chapter We choose MySQL for the target of JDBC.
Example 1: A Simple Java JDBC Command-line Application

This simple Java program illustrates how JDBC works with a database. The JDBC driver used in this example is a type-3 mysql driver. The customers data table is built in the test MySQL database. The schema of the table consists of three columns: customer_id, name, and Phone. Only three rows of customer data are stored in the table.

The following script file is used to create a customers table in the test database and insert three rows into this table.

At the MySQL prompt, type the command “source create_customers” to create the table and insert the data, and then verify it by the select SQL command. You will see the following screen.
After we set up the database table, we can develop Java JDBC application against this table. Here is a desktop command-line Java application which simply displays all data in the table.
Example 1 (cont.)

Simple.java
import java.sql.*;
public class Simple{
    public static void main(String argv[])
        throws Exception {
        int numCols;
        String query = "select * from Customers";
        // All JDBC operations require try ... catch
        try{
            String userName = "root";
            String password = "abc123";
            String url = "jdbc:mysql://localhost/test";
            // load JDBC driver
            Class.forName("com.mysql.jdbc.Driver").newInstance();
            // connect to mysql database test
            Connection conn = DriverManager.getConnection
                (url, userName, password);
        }
Example 1 (cont.)

// Make a JDBC select query statement specified above
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(query);
   // Get the total numbers of columns in the table
numCols = rs.getMetaData().getColumnCount();
   // Look over each row in the table
while(rs.next()){
    for(int i=1; i<=numCols; i++){
        System.out.print(rs.getString(i) + " | ");
        System.out.println();
    }
}
   // Close resultSet, statement, and connection
rs.close(); stmt.close(); conn.close();
}
catch(SQLException ex){
    System.out.println("Exceptions");}
This screen shows the execution result of this Java application command-line based program. It displays information of all customers in the table.
Example 2: A JDBC GUI Application

The following Java Windows-based application example shows a JDBC application with GUI interfaces. It works on the same database as the last example except that the GUI interface provides a mechanism for the user to make a query to find the information of a specific customer.

// QueryTest.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.sql.*;

public class QueryTest extends JFrame
    implements ActionListener{
    private JTextField t1, t2, t3;
    private JLabel l1, l2, l3;
    JButton b1;
    Connection conn;
public QueryTest() { // Construct a Windows frame
    super("Query Test");
    Container c = getContentPane();
    c.setLayout(new FlowLayout());
    try{
        String userName = "root";
        String password = "abc123";
        String url = "jdbc:mysql://localhost/test";
        Class.forName("com.mysql.jdbc.Driver");
        conn = DriverManager.getConnection(url, userName, password);
    } catch( ClassNotFoundException x)
    {System.out.println("DriverExceptions");}
    catch( SQLException x)
    {System.out.println("SQL Exceptions");}
    // Construct label and text field GUI components
    // t2 is for input query
Example 2 (Cont.)

```java
l1 = new JLabel( "Customer Id: ");
c.add(l1);
t1 = new JTextField( 15);
c.add(t1);
l2 = new JLabel( "Name: ");
c.add(l2);
t2 = new JTextField( 15);
c.add(t2);
l3 = new JLabel( "Phone: ");
c.add(l3);
t3 = new JTextField( 15);
c.add(t3);
b1 = new JButton("Execute");
c.add(b1);
```

// Registration of Execute button with the action
// listener so that actionPerformed method will be
// invoked when the button is pressed
Example 2 (Cont.)

```java
bl.addActionListener(this);
addWindowListener( new WindowAdapter(){
    public void windowClosing(WindowEvent e)
    {
        System.exit(0);
    }
});
setSize(300,160);
// Enable the frame
show();
}

public void actionPerformed(ActionEvent e) {
    // JDBC processing
    if (e.getSource() == bl) {
        int numCols;
        // Search for information for given CustomerId
        String query = "select * from customers where name like '" + t2.getText() + "'";
```
// Following JDBC code are almost identical with code
// of last example
try{
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(query);
    numCols = rs.getMetaData().getColumnCount();
    while(rs.next()){*
        t1.setText(rs.getString(1));
        t2.setText(rs.getString(2));
        t3.setText(rs.getString(3));
    }
    rs.close(); stmt.close(); conn.close();
} catch(SQLException ex){
    System.out.println("Exceptions");
}

public static void main(String args[]){
    new QueryTest();}
Example 2 (Cont.)

Let’s run this Java Windows application.

After starting up the QueryTest.class class with the Java interpreter you will see the following Java frame.
From these two examples you can see that the JDBC technology makes it much easier for a Java client program to process table data in a relational database on a database server.
5.4 Case Study: An Online Shopping Cart Implementation

This section presents a Java MVC architecture design for an online shopping cart application. Diagram 5.10 illustrates the Model module, View module, and Controller module, the connections between the modules, and the back-end database support as well. There are a variety of ways to implement an online store. Many functionalities of an online store are not included in this implementation such as customer information processing, shipping and handling processing, accounting processing, and etc. We have simplified the implementation in order to focus on the MVC design architecture so you will have a better picture of it. We divide the system into three sub-systems which will be discussed in the following sections. Figure 5.10 depicts the overall MVC architecture of this DVD online store Web application.
Figure 5.10 DVD online store architecture
5.4.1 View Module of DVD Online Shopping Cart Application

In this section, we present the presentation logic (View module) which has the following responsibilities:

- enabling client browsing of the DVD catalog
- selecting the items and adding them to the shopping cart
- removing items from the shopping cart
- displaying the shopping cart contents
- checking out.

The view module is also responsible for displaying a confirmation message after clients check out or displaying error messages when an error occurs.

First, let’s look at the front page, ShowProductCatalog.jsp which displays the DVD catalog. The initial page looks like the one below.
# DVD Catalog

<table>
<thead>
<tr>
<th>DVD Names</th>
<th>Rate</th>
<th>Year</th>
<th>Price</th>
<th>Quantity</th>
<th>AddCart</th>
</tr>
</thead>
<tbody>
<tr>
<td>SecretWindow</td>
<td>PG-13</td>
<td>2004</td>
<td>14.95</td>
<td></td>
<td>AddToCart</td>
</tr>
<tr>
<td>Shrek2</td>
<td>PG</td>
<td>2004</td>
<td>19.95</td>
<td></td>
<td>AddToCart</td>
</tr>
<tr>
<td>SpiderMan</td>
<td>PG-13</td>
<td>2004</td>
<td>18.95</td>
<td></td>
<td>AddToCart</td>
</tr>
<tr>
<td>Martin</td>
<td>R</td>
<td>2004</td>
<td>13.95</td>
<td></td>
<td>AddToCart</td>
</tr>
</tbody>
</table>
The *ShowProductCatalog.jsp* is a JSP file which displays the DVD catalog from the database. The ProductDataBean JavaBean is responsible for getting the catalog data from the database. The first time clients browse this page, the `<jsp:useBean>` tag will instantiate an instance of the ProductDataBean which makes a connection to a database to load the catalog data and display them as shown above. This page also includes another *DisplayShoppingCart.jsp* page as part of itself. If the cart is empty then nothing is displayed. The details of the JavaBean classes here will be discussed in the Model module section. When the client adds any item to the shopping cart, the parameter data is saved in the implicit request object and the request is sent to the *addToShoppingCart.java* Servlet for processing.
<%@ page import = "java.util.*"
       import="cart.*,java.net.*,java.text.*" %>
<jsp:useBean id = "data" scope= "request"
         class = "cart.ProductDataBean" />
<html>
<body>
<!-- Call getProductList() of the ProductDataBean to get
     the DVD product
     catalog and put it on the productList
-->
<% List productList = data.getProductList();
    Iterator prodListIterator = productList.iterator(); %>
<p><center><h1>DVD Catalog</h1>
<table border="1">
<tr>
    <th>DVD Names</th>
</tr>
<tr>
    <td>DVD Names</td>
</tr>
</table>
</center></p>
<th>Rate</th>
<th>Year</th>
<th>Price</th>
<th>Quantity</th>
<th>AddCart</th>
</tr></thread>

<%-- Display all DVD products row by row on the table, add an "addCart" button at the end of each row to allow clients to select --%>
<% while (prodListIterator.hasNext()){
    DVD movie = (DVD)prodListIterator.next();
    String movieQuantity = "movieQuantity"; %>
<tr>
<form name="addtoShoppingCart"
    action="/shoppingCart/servlet/addToShoppingCart"
    method="POST">
    <td><%= movie.getMovie() %></td>
    <td><%= movie.getRating() %></td>
    <td><%= movie.getYear() %></td>
</form>
</tr>
<table>
<thead>
<tr>
<th>&lt;td&gt; &lt;%= movie.getPrice() %&gt;&lt;/td&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;td&gt; &lt;input type = text name = &quot;movieQuantity&quot; size =&quot;5&quot; /&gt;&lt;/td&gt;</td>
</tr>
<tr>
<td>&lt;td&gt; &lt;input type=&quot;hidden&quot; name= &quot;movieName&quot; value=' &lt;%= movie.getMovie() %&gt;'&gt;</td>
</tr>
<tr>
<td>&lt;input type=&quot;hidden&quot; name= &quot;movieRate&quot; value=' &lt;%= movie.getRating() %&gt;'&gt;</td>
</tr>
<tr>
<td>&lt;input type=&quot;hidden&quot; name= &quot;movieYear&quot; value=' &lt;%= movie.getYear() %&gt;'&gt;</td>
</tr>
<tr>
<td>&lt;input type=&quot;hidden&quot; name= &quot;moviePrice&quot; value=' &lt;%= movie.getPrice() %&gt;'&gt;</td>
</tr>
<tr>
<td>&lt;input type=&quot;submit&quot; value=&quot;AddToCart&quot;&gt;</td>
</tr>
</tbody>
</table>

</form>      </tr>
Display the current shopping Cart by including

DisplayShoppingCart.jsp

<jsp:include page="DisplayShoppingCart.jsp"
flush="true" />

</center>
</body>
</html>
The *DisplayShoppingCart.jsp* is responsible for showing the contents of the current shopping cart. There is a “Remove” button at the end of each item row of the displayed shopping cart. This button connects to *RemoveItemServlet.java* in the controller module. It has a “Check out” button for clients to checkout. This button connects to the *checkoutServlet.java* in the controller module.

<%@ page import="cart.*,java.util.*,java.text.*" %>
<% ShoppingCart cart = (ShoppingCart) session.getAttribute("ShoppingCart");

    if (cart == null){
        cart = new ShoppingCart();
session.setAttribute("ShoppingCart", cart);

-- the ShoppingCart is stored as an attribute of implicit object shared by other Web components in the shopping session --
Vector items = cart.getItems();
if (items.size() != 0) {

-- Display the heading of the shoppingCart --
<h1>Shopping Cart</h1> <br>
<table border=4>
<tr><th>DVD Names<th>Rate<th>Year<th>Price
  <th>Quantity <th>Remove
<%     int numItems = items.size();
NumberFormat currency =
    NumberFormat.getCurrencyInstance();
for (int i=0; i < numItems; i++) {
    DVD item = (DVD) items.elementAt(i);
%>
<tr>
   
   <form action="/shoppingCart/servlet/removeItem"
      method="POST">
      <td><%= item.getMovie() %></td>
      <td><%= item.getRating() %></td>
      <td><%= item.getYear() %></td>
      <td><%= item.getPrice() %></td>
      <td><%= item.getQuantity() %></td>
      <td><input type="hidden" name="item" value='<%= i %>'>
         <input type="submit" value="Remove"></td>
   </form>
</tr>

<% } %>
</table>

<form action="/shoppingCart/servlet/checkout"
   method="POST">
   <input type="submit" name="Submit" value="Check out">
</form>

<% } %>
The *ShowConfirmation.jsp* is responsible for displaying a confirmation message and showing a total amount charged to the client. It then terminates the current session.

```jsp
<%@ page import="cart.*, java.text.*" %>
<html>
<body>
<h3>Your Order is confirmed!</h3>
<%DecimalFormat twoDigits = new DecimalFormat("0.00");
String totalPrice =
twoDigits.format(((ShoppingCart)session.getAttribute("ShoppingCart"))
.getTotalPrice());%>
<h3>The total amount is $<% totalPrice %></h3>
<% session.invalidate(); %>
</body>
</html>
```
5.4.2 Controller Module of On-Line DVD Shopping Cart Application

Since all Servlet classes are placed in the controller model and all of them have their URL-patterns, we include the web.xml here for your reference.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC 
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <servlet>
    <servlet-name>addToShoppingCartServlet</servlet-name>
    <servlet-class>
      cart.AddToShoppingCartServlet
    </servlet-class>
  </servlet>
</web-app>
```
<servlet>
    <servlet-name>removeItemServlet</servlet-name>
    <servlet-class>cart.RemoveItemServlet</servlet-class>
</servlet>
<servlet>
    <servlet-name>checkoutServlet</servlet-name>
    <servlet-class>cart.CheckoutServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>addToShoppingCartServlet</servlet-name>
    <url-pattern>/servlet/addToShoppingCart</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>removeItemServlet</servlet-name>
    <url-pattern>/servlet/removeItem</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>checkoutServlet</servlet-name>
    <url-pattern>/servlet/checkout</url-pattern>
</servlet-mapping>
</web-app>
The following code, 
*AddToShoppingCartServlet.java*, is a Servlet which is responsible for adding DVD items to a shopping cart. It uses both the DVD JavaBean and the ShoppingCart JavaBean. It gets all parameter data for a new DVD item from the front page via request object, instantiates a DVD instance and calls the *addItem* method of ShoppingCart Javabean to add an new DVD item to the shopping cart, and then transfers control to *ShowProductCatalog.jsp* which will be in charge of rendering the catalog and the updated shopping cart. This conforms to a typical pattern of an MVC controller.
// AddToShoppingCartServlet.java

generate code
String movieYear = request.getParameter("movieYear");
String price = request.getParameter("moviePrice");
int movieQuantity = Integer.parseInt(request.getParameter("movieQuantity"));
double moviePrice = Double.parseDouble(price);
// Create this DVD and add to the cart
DVD DVDItem = new DVD(movieName, movieRate,
    movieYear, moviePrice, movieQuantity);
HttpSession session = request.getSession();
// Get the cart
ShoppingCart cart = (ShoppingCart) session.getAttribute("ShoppingCart");
cart.addItem(DVDItem);
String url="/jsp/ShowProductCatalog.jsp";
ServletContext sc = getServletContext();
RequestDispatcher rd = sc.getRequestDispatcher(url);
rd.forward(request, response);}}
The following *RemoveItemServlet.java* is a Servlet which is responsible for removing DVD items from a shopping cart. It gets a DVD item to be removed from the front page via the request object and calls the removeItem(0) method of ShoppingCart JavaBean to remove the DVD item from the shopping cart, and then transfers the control to *ShowProductCatalog.jsp* which will be in charge of rendering the catalog and the updated shopping cart. This also conforms to a typical pattern of a MVC controller.

```java
// RemoveItemServlet.java
package cart;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
```
public class RemoveItemServlet extends HttpServlet {
    public void service(HttpServletRequest request,
                        HttpServletResponse response)
        throws IOException, ServletException {

        // Get the index of the item to remove
        int itemIndex =
            Integer.parseInt(request.getParameter("item"));
        HttpSession session = request.getSession();

        // Get the cart
        ShoppingCart cart = (ShoppingCart)
            session.getAttribute("ShoppingCart");
        cart.removeItem(itemIndex);

        // Display the cart and allow user to check out or
        // order more items
        String url="/jsp/ShowProductCatalog.jsp";
        ServletContext sc = getServletContext();
        RequestDispatcher rd =
                sc.getRequestDispatcher(url);
        rd.forward(request, response);   }}
The following `CheckoutServlet.java` is a Servlet which gets the ShoppingCart object and redirects control to the `ShowConfirmation.jsp` JSP page for rendering the confirmation message.

```java
// CheckoutServlet.java
package cart;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.io.*;

public class CheckoutServlet extends HttpServlet {
    public void service(HttpServletRequest request,
                         HttpServletResponse response)
        throws IOException, ServletException {
        HttpSession session = request.getSession();
```
// Get the cart
ShoppingCart cart = (ShoppingCart) session.getAttribute("ShoppingCart");
try{
    cart.completeOrder();
} catch(Exception e){
    e.printStackTrace();
}
response.sendRedirect(response.encodeRedirectURL("/shoppingCart/jsp/ShowConfirmation.jsp"));
There are three JavaBeans in this MVC model module: DVD.java, ShoppingCart.java, and ProductDataBean.java. All of them are used for presentations of input or output user interface in this Web application.

The following DVD.java JavaBean represents a single DVD item which can be added to a shopping cart or removed from a shopping cart. It has five properties: m_movie, m_rated, m_year, m_price, and quantity. It also provides access methods to these properties.

// DVD.java
package cart;
import java.io.*;
public class DVD implements Serializable {
    String m_movie;
    String m_rated;
    String m_year;
    double m_price;
    int quantity;
    public DVD() {
        m_movie = "";
        m_rated = "";
        m_year = "";
        m_price = 0;
        quantity = 0;
    }
    public DVD(String movieName, String movieRate, String movieYear, double moviePrice, int movieQuantity) {
        m_movie = movieName;
        m_rated = movieRate;
        m_year = movieYear;
        m_price = moviePrice;
        quantity = movieQuantity;
    }
}
m_movie = movieName;
m_rated = movieRate;
m_year = movieYear;
m_price = moviePrice;
quantity = movieQuantity;
}
public void setMovie(String title) {
    m_movie = title;
}
public String getMovie() {
    return m_movie;
}
public void setRating(String rating) {
    m_rated = rating;
}
public String getRating() {
    return m_rated;
}
public void setYear(String year) {
    m_year = year;
}

public String getYear() {
    return m_year;
}

public void setPrice(double p) {
    m_price = p;
}

public double getPrice() {
    return m_price;
}

public void setQuantity(int q) {
    quantity = q;
}

public int getQuantity() {
    return quantity;
}
The following *shoppingCart.java* is a JavaBean where all the items are stored in a Java vector (Java collection data structure). It provides all necessary access methods for shopping cart business logic processing such as `getItems()` which returns the shopping cart as a vector; `addItem()` adds an new item to the shopping cart and updates the quantity of the item in the cart; `removeItem()` removes an item from the shopping cart and updates the quantity of this item in the cart; `completeOrder()` method saves the shopping cart records in a database for future processing; and `getTotalPrice()` reports the total charges of this purchase to clients when the order is confirmed. A shopping cart is saved in the *shoppingCart* table in the *eshopdb* database. You can find all detail implementations of this Bean in the following *ShoppingCart.java* file.
// ShoppingCart.java

class ShoppingCart implements java.io.Serializable {
    private Connection connection;
    private PreparedStatement addRecord, getRecords;
    private Statement statement;
    private double totalPrice;
    static int CARTID = 1;
    protected Vector items;

    public ShoppingCart() {
        items = new Vector();
    }
}
public Vector getItems() {
    return (Vector) items.clone();
}

public void addItem(DVD newItem) {
    boolean flag = false;
    if (items.size() == 0) {
        items.addElement(newItem);
        return;
    }
    for (int i = 0; i < items.size(); i++) {
        DVD dvd = (DVD) items.elementAt(i);
        if (dvd.getMovie().equals(newItem.getMovie())) {
            dvd.setQuantity(dvd.getQuantity() +
                           newItem.getQuantity());
            items.setElementAt(dvd, i);
            flag = true;
            break;
        }
    }
}
if (newItem.getQuantity()>0 && (flag == false)) {
    items.addElement(newItem);
}

public void removeItem(int itemIndex) {
    items.removeElementAt(itemIndex);
}

public void completeOrder() throws Exception {
    Enumeration e = items.elements();
    connection = ProductDataBean.getConnection();
    statement = connection.createStatement();

    while (e.hasMoreElements()) {
        DVD item = (DVD) e.nextElement();
        String itemQuantity = "" + item.getQuantity();
        totalPrice = totalPrice + item.getPrice() * Integer.parseInt(itemQuantity);
    }
}
String movieName = item.getMovie();

String updateString =
    "INSERT INTO ShoppingCarts " +
    " VALUES (" + CARTID + ", ", " +
    item.getMovie() + ", ", " +
    item.getRating() + ", ", " +
    item.getYear() + ", ", " +
    item.getPrice() + ", ", " +
    item.getQuantity() + ");"
    statement.executeUpdate(updateString);
    CARTID ++;
}

public double getTotalPrice() {
    return this.totalPrice;
}
The *ProductDataBean.java* JavaBean represents the DVD online store catalog which is supported by the *products* table in the *eshopdb* database. The first time a client accesses the online DVD store, the catalog is loaded from the database by JDBC calls.

```java
// ProductDataBean.java

package cart;

import java.io.*;
import java.sql.*;
import java.util.*;
public class ProductDataBean implements Serializable {
    private static Connection connection;
    private PreparedStatement addRecord, getRecords;
```
public ProductDataBean() {
    try {
        String userName = "root";
        String password = "abc123";
        String url = "jdbc:mysql://localhost/test";
        Class.forName("com.mysql.jdbc.Driver").newInstance();
        connection = DriverManager.getConnection(url, userName, password);
        System.out.println("Database connection established");
    } catch(Exception e){e.printStackTrace();}
}

public static Connection getConnection() {
    return connection;
}

public static Connection getConnection() {
    return connection;
}

public ArrayList getProductList() throws SQLException {
    ArrayList productList = new ArrayList();
    Statement statement =
        connection.createStatement();
    ResultSet results = statement.executeQuery(
        "SELECT * FROM products");
    while (results.next()) {
        DVD movie = new DVD();
        movie.setMovie(results.getString(1));
        movie.setRating(results.getString(2));
        movie.setYear(results.getString(3));
        movie.setPrice(results.getDouble(4));
        productList.add(movie);
    }
    return productList;
}
5.4.4 Back-end Tier of On-Line DVD Shopping Cart Application

In this example we use two relational tables, *Products* and *ShoppingCarts*, in a MySQL database instance named *test* to support the *ProductDataBean* catalog and *ShoppingCart* data respectively. The following screen shows the table structures and data in these two tables.

The Products table has four records at this time while *ShoppingCart* table is empty.
Assume that the ROOT directory of this Web application is the `shoppingCart` directory under the `webapps` directory of the Tomcat server. All JSP files are placed in the `jsp` subdirectory of `shoppingCart` directory. All the Servlets are stored in the WEB-INF subdirectory of `shoppingCart` directory. We can use the URL `http://localhost/shoppingCart/jsp/ShowProductCatalog.jsp` to start a purchase from this online DVD store.
We add one “Secret Window” DVD to the shopping cart and see the change in the shopping cart.
We add two “Spider Man” and one “Martin” DVDs to the shopping cart afterward. Now, three items are in the cart.
After that we remove the first item from the shopping cart and only two items are left in the cart.
When we click on the “Check out” button to check out we see the confirmation message from the online DVD store.

Your Order is confirmed!
The total amount is $51.85