Assessing the Effectiveness of the Model View Controller Architecture for Creating Web Applications

Nick Heidke, Joline Morrison, and Mike Morrison
Department of Computer Science
University of Wisconsin-Eau Claire
Eau Claire, WI 54702
morrisjp@uwec.edu

Abstract

The Model View Controller (MVC) architecture has been widely embraced as an approach for developing Web-based applications that contain a server-side programming component. The bulk of the published literature on MVC Web applications to date describes the architecture and underlying specification of specific systems, but does not specifically address the effectiveness of MVC for developing these applications. This research seeks to fill this gap by comparing MVC to other widely-used Web development methods in terms of development time, maintainability, and the ability to support communication among designers and programmers by contrasting a non-MVC Web application with an MVC-based Web application, and highlighting the advantages and disadvantages of each approach.

Introduction

The Model View Controller (MVC) software engineering architecture has been widely embraced as an approach for developing Web-based systems that contain a server-side programming component, particularly for those requiring database access. MVC isolates the business logic from the user interface, with the goal of creating applications that are easier to manage and maintain because designers can modify the visual appearance of the application and programmers can modify the underlying business rules with fewer harmful side effects. The bulk of the published literature on MVC Web applications describes the architecture and underlying specification of these systems, but does not specifically address the effectiveness of the architecture. The purpose of this research is to compare MVC to other widely-used Web development methods in terms of development time, maintainability, and ability to support and enhance interaction among designers and programmers. As an initial step towards achieving this goal, we develop a sample Web application using a traditional (non-MVC) Web programming approach, Java Server Pages. Next, we develop the same application using two MVC frameworks, Struts and Java Server Faces. We then compare and contrast the applications and underlying development process, and highlight the advantages and disadvantages of each approach.

The first section of this paper describes previous research in this domain and highlights the absence of studies comparing the effectiveness of MVC to non-MVC Web applications. The next section describes a simple Web application developed using a traditional (non-MVC) approach, Java Server Pages. We highlight the maintenance problems that exist in this application. Next we provide an overview of MVC architectures for Web development, and describe the architecture and code of the same application developed using an MVC framework. We repeat the process using two other MVC platforms (Struts and Java Server Faces). We compare and contrast the applications and underlying development process, and draw conclusions concerning the strengths and limitations of the MVC architecture for Web development.

Previous Research

The MVC framework is a well-established design pattern that is widely used for applications with a user interface component (1). Several papers describe applications of the MVC framework for Web application development. For example, GuangChun et al. (2) describe an approach whereby the model component is a JavaBean, the View is created using Java Server Pages (JSPs), and the Controller is created using a Java servlet. Leff and Rayfield (3) describe an MVC Web application that provides flexibility in how the application is partitioned between the client and the server. Sauter et. al (4) describe an MVC application that adapts to multiple client devices, including computers, printers, and handheld devices.

A number of proprietary Web-oriented technologies have been developed using MVC, including Struts (2) and Maverick (3). Barrett and Delany (6) describe a non-proprietary MVC framework using XML and XSLTs.

The consensus among most authors is that the MVC framework provides significant advantages for Web based applications: less coupling, higher cohesion, easier maintenance, and enhanced scalability. Selfa et. al (3) state that MVC provides an added advantage for systems with a database component because it allows the same information to be presented in a variety of different formats by utilizing different information views. (It should be noted that the system developed by Selfa et. al was developed using ASP.NET, which uses a separate controller for each of the views in the Web site. This is a deviation from the standard MVC architecture, which prescribes a single controller for all views.) None of this work substantiates the advantages of the MVC architecture using specific comparisons between MVC and non-MVC Web applications that systematically identify the advantages and disadvantages of each approach.

In summary, a conceptual gap exists between the current published research and conclusions about the effectiveness of the MVC architecture. No studies address MVC's time to delivery, agility, or ability to enhance interaction among designers and programmers. Each of these qualities should be considered in a non-MVC architecture as well to compare results at a more objective level.

Web Development Using a Non-MVC Approach

Dynamic Web sites were originally created by writing programs that ran on a Web server and used commands such as `out.println("<h2>Class List</h2>");` to send HTML output to a browser. Common Gateway Interface (CGI) and Java Servlets are examples of technologies that use this approach. This approach forces the HTML commands to be tightly interleaved within the program commands.

To enable Web designers to work from an HTML perspective rather than a program code perspective, new approaches were developed that start with the HTML code and then allowed server-side programming commands to be interleaved within the HTML commands. Technologies using this approach include Active Server Pages (ASP), Java Server Pages (JSP), and PHP (Hypertext Preprocessor). These technologies allow Web designers to create static pages, then turn them over to programmers who add server-side programming code as needed. As with CGI and servlets, the program code and HTML code are tightly interleaved, and designers and programmers must work on the same files. This results in tightly coupled pages with low cohesion, which is not desirable.

To compare these approaches to an MVC approach, we first developed a sample non-MVC Web/database application using JSP and with a MySQL database. Figure 1 shows the code for the initial JSP (DisplayProduct.jsp), which retrieves and displays a series of product records from a database table. The non-HTML code loads the java.sql library (Line 1) and creates a database connection, statement, result set, and query (Lines 10-22).

The interleaved commands on lines 25-31 process the retrieved data and insert it into an HTML table using a while loop.

```
 1<%@ page import="java.sql.*,java.text.*" %>
 2<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 3                       "http://www.w3.org/TR/html4/loose.dtd">
 4<html>
 5<head>
 6  <link href="CSS/classStyles.css" rel="stylesheet" type="text/css" />
 7  <title>Display Product</title>
 8</head>
 9<body>
10<%// add the library with database functionality
11  Class.forName("com.mysql.jdbc.Driver").newInstance();
12  Connection cn =
13      DriverManager.getConnection("jdbc:mysql://stef.cs.uwec.edu/xxxxxx",
14                                  "xxxxxx", "xxxxx");
15  Statement stmt = cn.createStatement();
16  String query = "SELECT prod_desc, prod_cost, prod_price " +
17                 "FROM candy_product";
18  try {
19    // create the result set object
20    ResultSet rs = stmt.executeQuery(query);
21    NumberFormat f = NumberFormat.getCurrencyInstance();
22%>
23   <table class="dataTable">
24     <tr><th>Product</th><th>Cost</th><th>Price</th></tr>
25<% while (rs.next()) { %>
26     <tr>
27       <td align="left"><%=rs.getString("prod_desc") %></td>
28       <td align="right"><%=f.format(rs.getFloat("prod_cost")) %></td>
29       <td align="right"><%=f.format(rs.getFloat("prod_price")) %></td>
30     </tr>
31<% } %>
32     <tr>
33       <td align="left">
34         <input type="button" value="Add product"
35                onclick="location.href='InputInsert.jsp';" />
36       </td>
37       <td> </td>
38       <td> </td>
39     </tr>
40   </table>
41<% } catch (Exception e) { %>
42     Server error: <%= e.getMessage() %>
43<% } finally { cn.close(); } %>
44</body>
45</html>
```

Figure 1: DisplayProduct.jsp code listing

This first code sample illustrates *intra-crosscutting* (9), whereby commands for functionality (within <% %> markers) and presentation (HTML tags) are interleaved, or tangled, throughout the code in a single page. This code also demonstrates *inter-crosscutting*, which occurs when commands within a page depend on the underlying resource structure (9). Lines 13-14 and 16-17 illustrate this: the SQL query syntax depends on the selected database and its underlying structure. It also occurs in Lines 27-29, where the commands that display the data values on the page rely on the names of the underlying database fields.

The code in Figure 1 also demonstrates another type of inter-crosscutting, in which the control and functionality of the application spans multiple pages. On Line 35, the button's onclick event references the InputInsert.jsp page.

Figure 2 shows the code for a second JSP (InputInsert.jsp), which contains Web form inputs that prompt the user to enter information about a new product. This page does not contain JSP directives or scriptlets, and thereby does not contain tangled code. However, it demonstrates inter-crosscutting on Line 9, which specifies the name of the JSP (ProductInsert.jsp) that will process the resource-dependent inputs specified on Lines 13, 15, and 17 (prod_desc, prod_price, and prod_cost).

```
 1<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://
 2<html>
 3<head>
 4  <link href="CSS/classStyles.css" rel="stylesheet" type="text/css" />
 5  <title>Insert Product</title>
 6</head>
 7<script type="text/javascript" src="Includes/InputInsert.js"></script>
 8<body>
 9<form name="frmInsert" action="ProductInsert.jsp">
10<table width="300px" class="dataTable">
11  <tr><th colspan="2" align="center">Add a New Product</th></tr>
12  <tr><td>Description </td>
13      <td><input type="text" name="prod_desc"></td></tr>
14  <tr><td>Our Cost </td>
15      <td><input type="text" name="prod_cost"></td></tr>
16  <tr><td>Retail Price </td>
17      <td><input type="text" name="prod_price"></td></tr>
18  <tr><td> </td>
19      <td><input type="submit" value="Submit" /></td></tr>
20</table>
21</form>
22</body>
23</html>
```

Figure 2: Code listing of InputInsert.jsp

Figure 3 shows the code listing for ProductInsert.jsp, which receives the input parameters from InputInsert.jsp and then inserts them into the database. This listing illustrates the tangled code that results from intra-crosscutting. It also contains inter-crosscutting page

dependencies: when writing lines 19-21, the developer must be aware of the form input names specified on lines 13, 15, and 17 in the previous (InputInsert.jsp) page. This results in scattered code (9), where interdependent specifications exist across multiple files. Furthermore, all of the database connection objects (connection, statement, and resultset) are repeated for every page.

```
 1<%@ page import="java.sql.*" %>
 2<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 3                       "http://www.w3.org/TR/html4/loose.dtd">
 4<html>
 5<head><title>Product Insert</title></head>
 6<body>
 7<%// add the library with database functionality
 8  Class.forName("com.mysql.jdbc.Driver").newInstance();
 9  // create the connection object (use your account and password)
10  Connection cn =
11      DriverManager.getConnection("jdbc:mysql://stef.cs.uwec.edu/STUDENT",
12                                  "STUDENT", "S5333");
13  //create the prepared statement
14  try {
15    PreparedStatement pstmt = cn.prepareStatement("INSERT INTO candy_product " +
16                                                  "(prod_desc, prod_cost, " +
17                                                  " prod_price) " +
18                                                  "VALUES (?, ?, ?)");
19    pstmt.setString(1, request.getParameter("prod_desc"));
20    pstmt.setString(2, request.getParameter("prod_cost"));
21    pstmt.setString(3, request.getParameter("prod_price"));
22    pstmt.execute();
23    response.sendRedirect("DisplayProject.jsp");
24  } catch (Exception e) { %>
25  Server error: <%= e.getMessage() %>
26<%} finally {
27    cn.close();
28  }%>
29</body>
30</html>
```

Figure 3: Code listing of ProductInsert.jsp

In summary, tangled code created by intra-crosscutting interleaves the commands for the application's functionality and presentation across multiple pages. This approach makes both the programming code and the HTML markup more difficult to understand, maintain, and debug. It also requires both Web designers and programmers to be involved in the Web development and maintenance process.

Scattered code, created by inter-crosscutting, uses specifications that depend on an underlying resource (a database, other pages, etc.). This makes the application difficult to maintain when a change is made to the database or other pages, because the changes must be reflected across all affected Web pages. While this is manageable in small applications, it becomes difficult to manage when Web applications have many interdependent pages. The problems of tangled code and scattered code led to the development and use of the MVC architecture for dynamic Web applications, which the next section describes.

Web Application Development Using the MVC Architecture

.

Figure 4 illustrates an MVC architecture created using Java-based technologies. In this architecture, user HTTP requests are routed through a controller, which is typically implemented as a servlet.

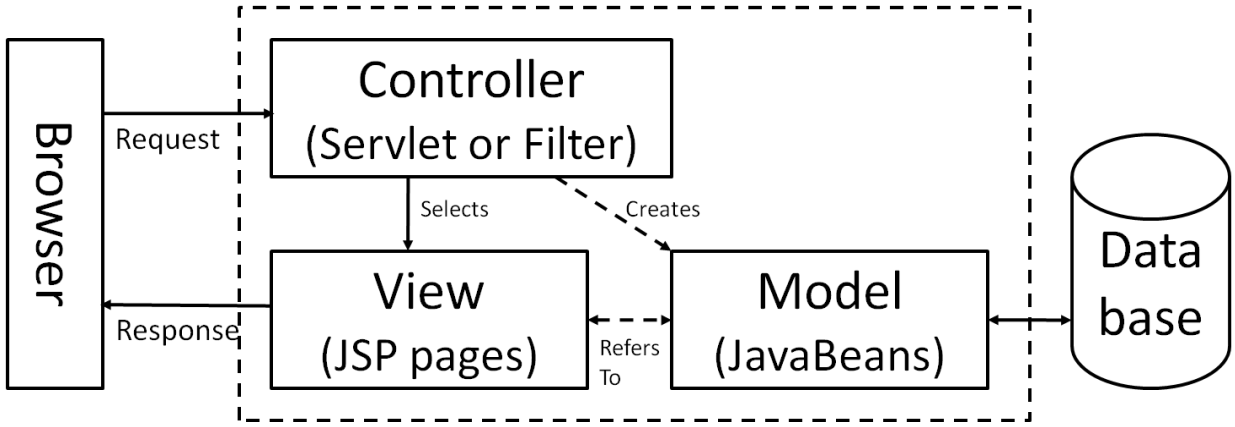

Figure 4: MVC Architecture using Java

The controller is responsible for processing the request, instantiating model classes if needed, and selecting the response to be returned to the browser. If using JSP pages for the view, the JSP page is able to retrieve data stored in the model's JavaBeans using the JSP Expression Language or by using special tags from the `java.sun.com/jsp/jstl/core` tag library.

This approach effectively separates the program code and HTML code and allows the programmers and designers to work far more independently. The programmers can write the controller and model code without interleaving the HTML, which is found in the view. Similarly, the view contains far less programming code than in non-MVC approaches. Although the JSP Expression language and tag libraries are a form of coding, they are less intrusive than adding scriptlets and directives to a JSP page.

Using an MVC approach, the DisplayProduct.jsp illustrated in Figure 1 is simplified to the listing shown in Figure 5. There are no scriptlets, and the only directive is Line 3's taglib reference. The page still has to use the JSP Expression language (${name} entries) and the Java Standard Tag Library's (jstl) core tag library to create a loop accessing the data collection stored in the model. Therefore, this approach reduces intra-crosscutting issues but does not eliminate them.

Inter-crosscutting issues remain as well. DisplayProducts.jsp relies on another resource, which is a JavaBean named product and is stored as a session variable. In addition, the page now refers to a controller action named 'input.action' when the button is clicked

(Line 22). Although close coordination between programmers and designers is still essential, they no longer have to work on the same files.

```
1<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
2                      "http://www.w3.org/TR/html4/loose.dtd">
3<%@ taglib uri='http://java.sun.com/jsp/jstl/core' prefix='c'%>
4<html>
5<head>
6  <link href="CSS/classStyles.css" rel="stylesheet" type="text/css" />
7  <title>Display Product</title>
8</head>
9<body>
10   <table class="dataTable">
11     <tr><th>Product</th><th>Cost</th><th>Price</th></tr>
12<c:forEach var="row" items="${product.data}">
13     <tr>
14       <td align="left">${row.prod_desc}</td>
15       <td align="right">${row.prod_cost}</td>
16       <td align="right">${row.prod_price}</td>
17     </tr>
18</c:forEach>
19     <tr>
20       <td align="left">
21         <input type="button" value="Add product"
22                onclick="location.href='input.action';" />
23       </td>
24       <td> </td>
25       <td> </td>
26     </tr>
27   </table>
28   <br />
29   ${product.error_msg}
30</body>
31</html>
```

Figure 5: View (DisplayProduct.jsp)

The controller servlet is shown in Figure 6. It scans submitted addresses and uses an if/else if structure to determine actions and responses. The controller determines if the model is needed and selects the next view. It still relies on information encoded into the URL by the JSP pages (the action="insert.action" in Figure 5).

```
public class ControllerServlet extends javax.servlet.http.HttpServlet
                              implements javax.servlet.Servlet {
   static final long serialVersionUID = 1L;
   protected void doGet(HttpServletRequest request, HttpServletResponse response)
             throws ServletException, IOException {
       doPost(request, response);
   }
   protected void doPost(HttpServletRequest request, HttpServletResponse response)
             throws ServletException, IOException {
       String uri = request.getRequestURI();
       int lastIndex = uri.lastIndexOf("/");
       String action = uri.substring(lastIndex + 1);
       String dispatchURL = null;

       if(action.equals("display.action")) {
           Product product = new Product();
           product.retrieveProductInformation(request, null);
           dispatchURL = "DisplayProduct.jsp";
       } else if(action.equals("insert.action")) {
           Product product = new Product();
           product.runInsert(request);
           dispatchURL = "DisplayProduct.jsp";
       } else if(action.equals("input.action")) {
           dispatchURL = "InputInsert.jsp";
       }
       if(dispatchURL != null) {
           RequestDispatcher rd = request.getRequestDispatcher(dispatchURL);
           rd.forward(request, response);
       }
   }
}
```

Figure 6: Controller (ControllerServlet class)

The controller servlet processes URLs ending with `.action`, which is configured in a file named web.xml as shown in Figure 7. The URL pattern specified in line 14 is `*.action`. This could be set to * which would intercept everything; however, that would increase the server's processing load.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://j
         id="WebApp_ID" version="2.5">
   <servlet>
     <description></description>
     <servlet-name>ControllerServlet</servlet-name>
     <servlet-class>MVCPkg.ControllerServlet</servlet-class>
  </servlet>
  <servlet-mapping>
     <servlet-name>ControllerServlet</servlet-name>
     <url-pattern>*.action</url-pattern>
  </servlet-mapping>
  <!-- Don't allow direct access to jsp pages -->
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>Deny Direct Access</web-resource-name>
            <description></description>
            <url-pattern>*.jsp</url-pattern>
        </web-resource-collection>
        <auth-constraint>
        <role-name>Denied</role-name>
        </auth-constraint>
    </security-constraint>
    <security-role>
        <role-name>Denied</role-name>
    </security-role>
```

Figure 7: Controller (web.xml)

Figure 8 illustrates the model, which in this example consists of a single class named Product. This example doesn't have nor need a ProductInsert.jsp page as before, because its processing is done by the Product class's runInsert method. Any data retrieved in Product that might be displayed within a JSP page must be saved as a class variable with an associated getter method. In addition, the instance of the Product class (or class variable) has to be saved as a session object prior to calling the JSP page. Assuming these guidelines are followed, the JSP page can access the data as shown in Figure 5.

```
public class Product implements Serializable {
  private static final long serialVersionUID = 1L;
  private long prod_id;
  private String prod_desc;
  private float prod_cost;
  private float prod_price;
  private String formattedCost;
  private String formattedPrice;
  private String error_msg;
  private ArrayList<Product> data;
  private Connection cn = null;
  public long   getProd_id()        { return prod_id; }
  public String getProd_desc()      { return prod_desc; }
  public float  getProd_cost()      { return prod_cost; }
  public float  getProd_price()     { return prod_price; }
  public String getError_msg()      { return error_msg; }
  public String getFormattedCost()  { return formattedCost; }
  public String getFormattedPrice() { return formattedPrice; }
  public void runInsert(HttpServletRequest request) {
    this.error_msg = "Success";
    try {
      Class.forName("com.mysql.jdbc.Driver");
      cn = DriverManager.getConnection("jdbc:mysql://stef.cs.uwec.edu/xxxxxx",
                                       "xxxxxx", "xxxxxx");
      PreparedStatement pstmt =
      cn.prepareStatement("INSERT INTO candy_product (prod_desc, prod_cost, prod_price)"+
                          "VALUES (?, ?, ?)");
      pstmt.setString(1, request.getParameter("prod_desc"));
      pstmt.setString(2, request.getParameter("prod_cost"));
      pstmt.setString(3, request.getParameter("prod_price"));
      pstmt.execute();
    } catch (Exception e) {
      this.error_msg = e.getMessage();
    }
    retrieveProductInformation(request, cn);
  }
  public void retrieveProductInformation(HttpServletRequest request, Connection cn) {
    this.data = new ArrayList<Product>();
    NumberFormat f = NumberFormat.getCurrencyInstance();
    try {
      Class.forName("com.mysql.jdbc.Driver");
      if(cn == null || cn.isClosed()) {
        cn=DriverManager.getConnection("jdbc:mysql://stef.cs.uwec.edu/STUDENT",
                                       "STUDENT", "S5333");
      }
      String query = "SELECT prod_id, prod_desc, prod_cost, prod_price " +
                     "FROM candy_product ORDER BY prod_desc";

      Statement stmt = cn.createStatement();
      ResultSet rs = stmt.executeQuery(query);
      while(rs.next()) {
        Product p = new Product();
        p.prod_id = rs.getLong("prod_id");
        p.prod_desc = rs.getString("prod_desc");
        p.prod_cost = rs.getFloat("prod_cost");
        p.prod_price = rs.getFloat("prod_price");
        p.formattedCost = f.format(p.prod_cost);
        p.formattedPrice = f.format(p.prod_price);
        this.data.add(p);
      }
    } catch (Exception e) {
      this.error_msg = e.getMessage();
    }
    request.setAttribute("product", this);
  }
}
```

Figure 8: Model (Product class)

Using the Struts MVC Framework

Struts is an MVC framework that provides a pre-written controller along with an XML based configuration file to match URLs with actions. Struts automatically instantiates the action class (Product in the previous example) and populates its properties with user inputs. This substantially simplifies the MVC controller coding shown in the previous example.

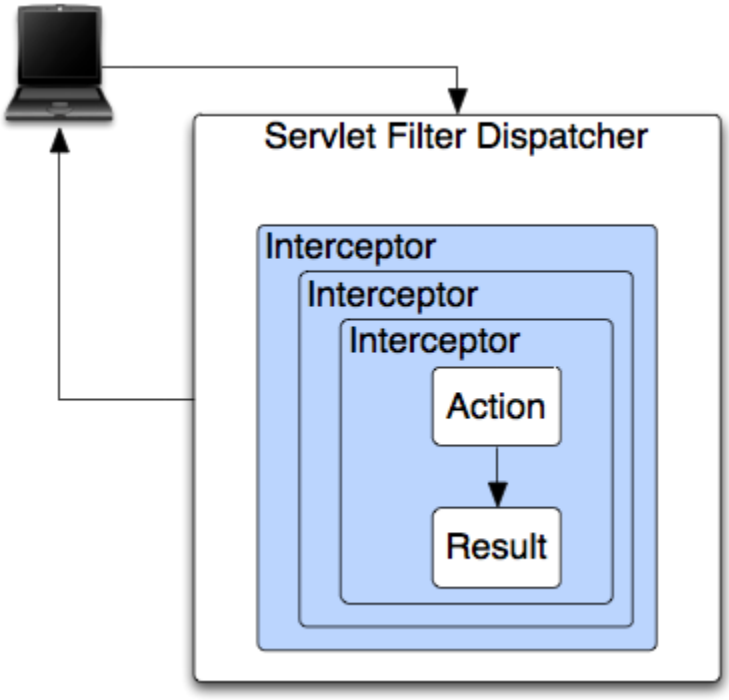

Figure 9: Struts Architecture in a Nutshell (10)

Figure 9 shows the Struts architecture. The browser requests a resource, the controller determines the appropriate action, the interceptors automatically apply common processing to the request (such as validation, authorization, etc.), the specified action method executes (usually storing or retrieving information from a database), and the result renders the output to the browser, typically as a JSP page.

A basic Struts Web application can be created to closely replicate the previous MVC example. The biggest change is the addition of the struts.xml configuration file as shown in Figure 10.

```
1<?xml version="1.0" encoding="UTF-8"?>
2<!DOCTYPE struts PUBLIC
3    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
4    "http://struts.apache.org/dtds/struts-2.0.dtd">
5
6<struts>
7    <constant name="struts.enable.DynamicMethodInvocation" value="false" />
8    <constant name="struts.devMode" value="true" />
9    <package name="StrutsPkg" namespace="/" extends="struts-default">
10       <action name="display"
11               class="StrutsPkg.Product"
12               method="retrieveProductInformation">
13           <result name="success">DisplayProduct.jsp</result>
14           <result name="error">Error.jsp</result>
15       </action>
16       <action name="input">
17           <result name="success">InputInsert.jsp</result>
18       </action>
19       <action name="insert"
20               class="StrutsPkg.Product"
21               method="runInsert">
22           <result name="success" type="redirect">
23               display.action
24           </result>
25           <result name="error">Error.jsp</result>
26       </action>
27   </package>
28</struts>
```

Figure 10: struts.xml Configuration File

In this code listing, the DOCTYPE tag specifies which version of struts is being used, and must match the version of the struts libraries used in the Web application. The constant tags on Lines 7 and 8 allow error messages to be displayed in the browser when Struts errors occur, and should be removed from a deployed application. The package element starting on Line 9 and closed on Line 27 specifies how URLs are matched to actions in the specified package. Struts allows one or more packages to be defined in struts.xml. If the namespace attribute isn't included, it defaults to "/". If the namespace isn't the default value, the namespace must be added to the URL accessing the actions in the package.

Actions optionally invoke methods in the model, and then invoke a view. If a method is invoked, its return value is compared to the name attribute in the action's results to determine the view. The default return values need to be typed as String. We aren't going to show the revised Product class code, however its methods have been rewritten to return strings like "success" and "error" depending on what happens within each method. Also the `request.setAttribute("product", this);` in Figure 8 is deleted since Struts will automatically do this. Notice on Lines 22 – 23 in Figure 10 that a result can chain to an action rather than to a view. If this is done, the state of the model in the initial action is preserved and sent to the chained action.

As with the MVC example, web.xml configures the application to route requests through the controller. Figure 11 shows web.xml with a Struts filter. With Struts, the controller is a filter rather than a servlet. This isn't a major difference from using a servlet, and could easily be incorporated into the previous MVC example. The reason for using a filter is that it is better suited to processing "all" requests, including those for static resources than a servlet.

```
3     <filter>
4       <filter-name>struts2</filter-name>
5       <filter-class>org.apache.struts2.dispatcher.FilterDispatcher</filter-class>
6     </filter>
7     <filter-mapping>
8       <filter-name>struts2</filter-name>
9       <url-pattern>/*</url-pattern>
10    </filter-mapping>
```

Figure 11: web.xml with a Struts filter

Struts provides additional features that are fairly easy to include in our application. For example, Struts supports interceptor classes, which are defined in a file named struts.xml, and can allow the action to proceed, or divert to a different action. For example, an attempt to access a page requiring the user to log on could be diverted to login.action, as shown in Figure 12. Interceptor classes are required to implement an intercept method that will be automatically called, and therefore aren't explicitly named in struts.xml.

```
9     <package name="StrutsPkg" namespace="/" extends="struts-default">
10        <interceptors>
11            <interceptor name="login" class="StrutsPkg.Login" />
12        </interceptors>
13        <action name="display"
14                class="StrutsPkg.Product"
15                method="retrieveProductInformation">
16            <interceptor-ref name="login" />
17            <result name="success">DisplayProduct.jsp</result>
18            <result name="error">Error.jsp</result>
19        </action>
```

Figure 12: struts.xml with an interceptor

Another additional Struts feature is its user interface tags. Figure 13 shows InputInsert.jsp rewritten to use these tags. The tag library is included on Line 2 using a JSP directive. Line 10 adds a form, and Lines 11-15 add entries into the default two column table created by a struts-tags form. The result is a page that looks identical to the one generated by Figure 2's JSP page but is created using less code. In some cases, struts-tags can eliminate loops populating pick lists that are often included in JSP pages. An example of this is shown in Figures 14 and 15 for a selection list.

```
1<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http:/
2<%@ taglib prefix='s' uri='/struts-tags' %>
3<html>
4<head>
5  <link href="CSS/classStyles.css" rel="stylesheet" type="text/css" />
6  <title>Insert Product</title>
7</head>
8<script type="text/javascript" src="Includes/InputInsert.js"></script>
9<body>
10<s:form key="frmInsert" action="insert.action" cssClass="dataTable">
11  <tr><th colspan="2" align="center">Add a New Product</th></tr>
12  <s:textfield key="prod_desc" label="Description" />
13  <s:textfield key="prod_cost" label="Our Cost" />
14  <s:textfield key="prod_price" label="Retail Price" />
15  <s:submit value="Submit"/>
16</s:form>
17</body>
18</html>
```

Figure 13: struts-tags

```
<tr><td align=right>Department: </td>
    <td>
        <select size="3" name="d_id">
        <c:forEach var="line" items="${data}">
            <option value="${line.d_id}">${line.d_name}</option>
        </c:forEach>
        </select>
    </td>
</tr>
```

Figure 14: Populating a Selection List using a Loop

```
    <s:select key="d_id" label="Department"
              headerKey="0" headerValue="Select Department"
              list="data" listKey="d_id" listValue="d_name"/>
```

Figure 15: Using a struts-tags to Populate a Selection List

Struts retains intra- and inter-crosscutting issues. The code on Line 10 in Figure 13 shows inter-crosscutting whereby the page depends on an underlying resource (insert.action) that is part of the controller. View pages are still tightly coupled to their action methods by the names used in form inputs that are tied to the properties in the related JavaBean. For example, in Figures 14 and 15, data, d_id and d_name are properties of the associated JavaBean. The main advantage of Struts over a custom built MVC application is that much of the MVC architecture is already written saving a considerable amount of programming effort.

Using the Java Server Faces (JSF) MVC Framework

JSF is an MVC framework similar to Struts that provides a pre-written controller along with an XML- based configuration file to match URLs with actions. JSF adds a component model that has events and properties similar to the Java Swing library. Examples of JSF components include calendars and rich HTML inputs that would require a substantial amount of effort to create from scratch. As expected, there will a training cost to learn how to use the new features introduced in JSF.

The JSF faces-config.xml file is roughly equivalent to struts.xml in terms of functionality. The issues with intra and inter-crosscutting are similar to Struts, however. Rich inputs and components can reduce intra-crosscutting (less code intermingled with HTML) at the expense of learning how to use them. Inter-crosscutting is still an issue, however since rich inputs and components that interact with the model must still know the relevant property names used in the model.

Discussion and Conclusions

We have shown that non-MVC Web applications exhibit a high degree of inter- and intra-crosscutting. MVC Web applications virtually eliminate intra-crosscutting, and reduce inter-crosscutting to some extent. How does this impact application development time, maintainability, and interactions among programmers and designers?

The startup costs for MVC architectures can be high. Newcomers to the Java-based MVC will initially face the challenge of assembling a compatible set of tag and other libraries needed for the current versions of servlet and JSP technologies. There are also subtle but important differences in the syntax used in configuration files and tags that occur with each version of servlets, JSPs, and tag libraries. Once these are understood, development using MVC requires writing the controller or setting up controller configuration files in Struts or JSF.

If you create a customized (non-Struts or JSF) MVC environment, you must train your developers in the nuances of your environment. An advantage to using Struts or JSF over a custom MVC environment is that you can hire programmers with prior experience in these technologies. (Other MVC frameworks exist beyond Struts and JSF, including Ruby on Rails or Tapestry. However, the issues involved with any pre-built MVC framework, are similar to those identified for Struts and JSF.)

After the startup costs of learning how to configure, program, and design with a selected MVC framework are realized, development time in an MVC versus a non-MVC environment is likely to be similar for small projects. For larger projects, we anticipate MVC will be faster because the application will be more loosely coupled, have higher cohesion, and minimize intra and inter-crosscutting. We believe a large MVC site will be more easily maintained and modified over time than a large site developed using non-MVC approaches.

We are doubtful that MVC will provide significant benefits for small, less complicated Web sites, because the maintenance issues caused from interleaved (intra-crosscutting) programming code and HTML is offset by the complexity of setting up the MVC architecture. If a team working on a new Web site is familiar with an MVC technology, however, it make sense to use MVC for smaller Web sites.

References

Works Cited

1. **Gamma, E., Helm, R., Johnson, R., Vlissides, J.** *Design patterns: elements of reusable object-oriented software.* Reading, Mass. : Addison-Wesley, 1995.
2. **GuangChun, L., WangYanhua, Xianliang, L, and Hanhong.** A Novel Web Application Frame Developed by MVC. *Software Engineering Notes.* 2003, Vol. 28, 2.
3. **Leff, A. and Rayfield, J.T.** Web-Application Development Using the Model/View/Controller Design Pattern. *IEEE XPlore.* 2001.
4. **Sauter, P., Vogler, G., Specht, G., and Flor, T.** A Model-View-Controller extension for pervasive multi-client user interfaces. *Pers Ubiquit Comput.* October, 2004.
5. **The Apache Software Foundation.** *Apache Struts Web Application Framework.* http://jakarta.apache.org/struts.
6. **Maverick Project.** *Source Forge.* http://mav.sourceforge.net.
7. *openMVC: A Non-proprietary Component-based Framework for Web Applications.* **Barrett, R. and Delany, S.J.** New York : ACM, WWW 2004.
8. *A Database and Web Application Based on MVC Architecture.* **Selfa, D.M., Carrillo, M., and Rocio Boone, M.** Puebla, Mexico : IEEE, IEEE Int. Conf. on Electronics, Communications, and Computers (CONIELECOMP 2006).
9. *Domain Driven Web Development With WebJinn.* **Kojarski, S. andLorenz, D.H.** Anaheim, CA : ACM, OOPSLA 2003.
10. *Apache Struts 2 Documentation* http://struts.apache.org/2.0.6/docs/home.html. 2/12/2007