Developing Java Web Applications with Jakarta Struts Framework

Liviu Gabriel CRETU
"Al. I. Cuza" University, Faculty of Economics and Business Administration, Iasi

Although one can design a fully functional application with either of the two Java technologies, the combination of both with Model-View-Controller(MVC) design pattern seems to be the right (and recommended) choice in order to develop highly flexible, long term maintainable web applications. This paper is split in two sections. The first section introduces some essential aspects of the Servlets and JSP technologies together with some architectural issues about Java web application design. The second section describes the basic architecture of Jakarta Struts framework. A simple web application is built step by step to help readers better understand this well known Java web application framework.
***Key words**: Servlet, JSP, Model-View-Controller architecture, Struts framework.*

Two J2EE technologies for web applications. An Introduction

A server in the Web tier processes HTTP requests. In a J2EE application, the Web tier usually manages the interaction between Web clients and the application's business logic. The Web tier typically produces HTML or XML content, though the Web tier can generate and serve any content type. While business logic is often implemented as enterprise beans, it may also be implemented entirely within the Web tier.

The Web tier typically performs the following functions in a J2EE application: manages interaction between Web clients and application business logic; generates dynamic content; presents data and collects input; controls screen flow (which page to display next); maintains state over the lifetime of a user session.

- *Java Servlets*. A Java Servlet is a Java class that extends a J2EE-compatible Web server. Each servlet class produces dynamic content in response to service requests to one or more URLs. Such a class must implement at least two methods `doGet()` and `doPost()` to handle HTTP GET and POST requests from the browser client.

Servlets offer some important benefits over earlier dynamic content generation technologies. Servlets are compiled Java classes, so they are generally faster than CGI programs or server-side scripts. Servlets are portable both at the source-code level between all Web containers that implement Java Servlet specification and at the binary level (because of the portability of Java bytecode). Servlets also provide a richer set of standard services than any other widely adopted server extension technology. In addition to producing content, servlets have several features that support application structure. A developer can create classes that respond to events in a servlet's lifecycle by implementing listener interfaces. A servlet can also be extended by one or more servlet filters, which are reusable classes that wrap calls to a servlet's `service` method, transforming the request or the response.

- *JavaServer Pages (JSP)*. Most Web applications produce primarily dynamic HTML pages that, when served, change only in data values and not in basic structure. For example, all of the catalog pages in an online store may have identical structure and differ only in the items they display. JSP technology exists for producing such content. A JSP page is a document containing fixed template text, plus special markup for executing embedded logic writen in pure Java language or including other resources. The fixed template text is always served to the requester just as it appears in the page, like traditional HTML. The special markup can take one of three forms: directives, scripting elements, or custom tags (also known as "custom actions"). Directives are instructions that control the behavior of the JSP page compiler and therefore are

evaluated at page compilation time. Scripting elements are blocks of Java code embedded in the JSP page between the delimiters <% and %>. Custom tags are programmer-defined markup tags that generate dynamic content when the page is served. The JavaServer Pages specification defines a set of standard tags that are available in all platform implementations. Custom tags and scripting elements generate dynamic content that is included in a response when a page is being served.

JSP pages differ from servlets in their programming model. A JSP page is primarily a document that specifies dynamic content, rather than a program that produces content. JSP page technology provides a "document-centric" alternative to "programmatic" servlets for creating dynamic, structured data. Although a JSP page looks to its author like a document, most J2EE implementations translate a JSP page into a servlet class when it is deployed.

- *Two design paterns*. When Java servlets were first invented, many programmers quickly realized that they were a good thing. They were faster and more powerful that standard CGI, portable, and infinitely extensible. But writing HTML to send to the browser in endless `println()` statements was a very problematic and error predisposed task. The answer to that was JavaServer Pages, which turned servlet writing inside-out. Now developers could easily mix HTML with Java code, and have all the advantages of servlets. As a consequence, Java web applications quickly became "JSP-centric". This in-and-of itself was not a bad thing, but it did little to resolve flow control issues and other problems of web applications (jakarta.apache.org/struts/).

Many developers realized that JavaServer Pages and servlets could be used together to deploy web applications. The servlets could help with the control-flow, and the JSPs could focus on writing HTML. As a fact, using JSPs and servlets together became known as Model 2 (where using JSPs alone was Model 1).

A J2EE application's Web tier serves HTTP requests. At the highest level, the Web tier does four basic things in a specific order: interprets client requests, dispatches those requests to business logic, selects the next view for display, and generates and delivers the next view.

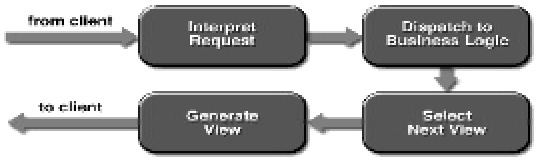

Fig. 1. The Web-Tier Service Cycle (source: Allamaraju, S. (2001))

The Web-tier controller receives each incoming HTTP request and invokes the requested business logic operation in the application model. Based on the results of the operation and state of the model, the controller then selects the next view to display. Finally, the controller generates the selected view and transmits it to the client for presentation.

The two architectural designs that we specified earlier (Model 1 and Model 2) first appeared in the early drafts of the JSP specifications (Budi Kurniawan(2001)).

In Model 1 architecture, the application is page-centric. The client browser navigates through a series of JSP pages in which any JSP page can employ a JavaBean that performs business operations. However, the highlight of this architecture is that each JSP page either processes its own input or processes some session parameters and decides which Web resource to display next.

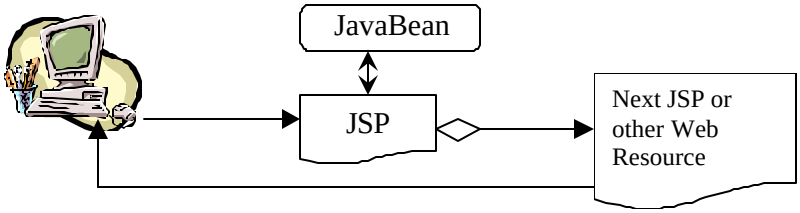

Fig. 2. Model 1: JSP page processes the business logic and acts like a controller

This architectural design is suitable only for small projects, but when we talk about real applications' design it is thought rather as a bad practice because it implies the following major disadvantages: hard to develop by an heterogeneous team where every member has its own, specialized, skills; not flexible; hard to maintain.

Model-View-Controller ("MVC") is the Sun's recommended architectural design pattern for interactive applications (java.sun.com/j2ee/). Following MVC approach, we can organize an interactive application into three separate modules: one for the application model with its data representation and business logic, the second for views that provide data presentation and user input, and the third for a controller to dispatch requests and control flow.

The Model 2 architecture is basically a MVC architecture that separates business logic, content generation and content presentation. The core of this model consists in the presence of a controller servlet between the client browser and the JSP pages or other servlets that generate and present the content. The controller servlet may perform some business logic or delegate this task to a JavaBean object and selects the corresponding presentation based on the request URL, input parameters, and application state. In this model, presentation parts (JSP pages) are isolated from each other.

Model 2 applications are more flexible and easier to maintain, and to extend, because views do not reference each other directly. The Model 2 controller servlet provides a single point of control for security and logging, and often encapsulates incoming data into a form usable by the back-end MVC model.

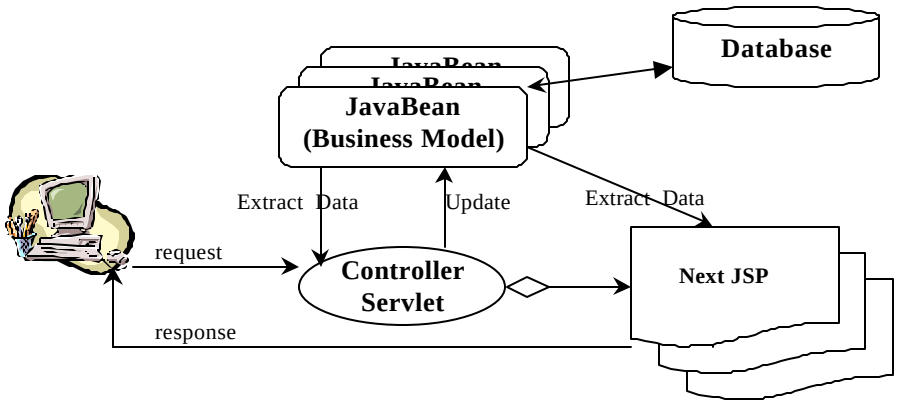

Fig. 3. Model 2: JSP pages provide only the presentation. Business logic and request dispatching are separated.

A simple web application with Jakarta Struts Framework

Most Web-tier application frameworks use some variation of the MVC design pattern. The Apache Jakarta Project (jakarta.apache.org/struts) has developed one that has been well adopted by Java community developers.

Struts is comprised of a controller servlet, beans and other Java classes, configuration files, and tag libraries. This means that when you have downloaded Struts you have available: (1) a controller for your application (the Struts servlet acts as a common controller for the whole application); (2) a collection of tag

libraries that will be used in application's JSP-pages.
To glue these things together Struts uses a set of configuration files.
The simplest way to install Struts framework is to download the archive from (jakarta.apache.org/struts), unpack the "struts-blank.war" which contains a blank apllic a-tion and copy the unpacked content into the applications folder of the web server (if Tomcat is that server, the applications folder is "/webapps"). Then rename the "/struts-blank" folder to, let's say /WebTest. This folder contains a standard Java web application directory structure and the files needed to develop a Struts application (figure 4).

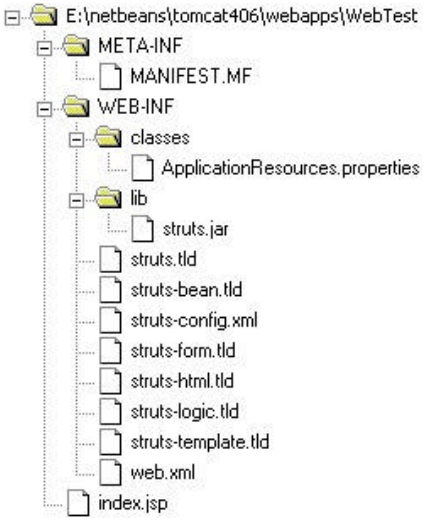

Fig. 4. Struts' directory structure

Files in blank-application have the following meanings: (1) the "*.tld" files are custom tag libraries that we use instead of standard HTML tags in our JSP pages; (2) the "struts-config.xml" is the core of the Struts applic a-tion, because it maps user actions (such as submitting a form) to Java beans that perform the actual action process; (3) "Application-Resources.properties" file contains pears of named parameters and textual values that we use in JSP Pages for writing a string that may change form one situation to another (very useful for multiple-language applications and for error messages) ; (4) the "struts.jar" contains Java class libraries of the Struts framework.
When you develop a Struts web application you basically follow six steps over and over until the application is finished: (1) design a JSP page with Struts' custom tags instead of standard HTML tags; (2) code a Java bean that extends `ActionForm` and defines properties that maps input elements of the HTML form defined earlier; (3) optionally implement the `validate()` method of the `ActionForm` class to validate user input (4) code a Java class that extends `Action` class and provide an implementation for the `perform()` method. (5) build the response documents for that request (6) glue all pieces created in the above steps by writing an `<action>` tag in the "struts-config.xml" file.
Figure 5 pictures a simple Struts Application that validates login information pr ovided by users in a HTML form defined in Login.jsp.

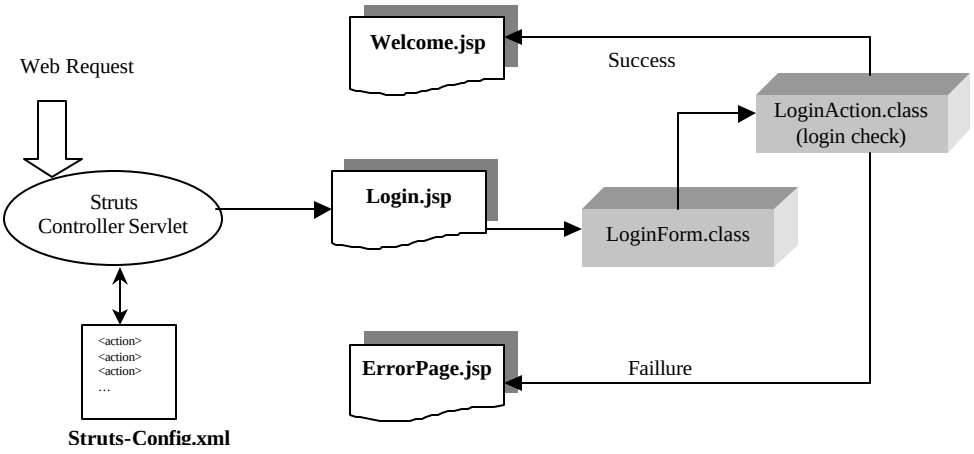

Fig. 5. A simple login application with Struts framework

To build this application we will follow the six phases discussed above. Additional information, necessary to fully understand the framework, will be provided as we step through different stages of application creation.

The *first step* consists of building the Login.jsp page (remember that in Java everything is case-sensitive, even the names of files that contains classes). The source code can be found in listing 1. As you can see, the standard HTML tags have been replaced with Struts tags (<html:text property="userid"> instead of <input type="text" name="userid">). In order for the `ActionForm` bean to work, we **must** use Struts' own tags for creating the HTML form and the GUI controls in it. We will not go into details about these tags - they're clearly modeled after the "real" HTML-tags and can be found, in details, in Struts documentation.

Listing 1. The Login.jsp page

```
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ page contentType="text/html; charset=ISO-8859-1" %>

<html>
<head><title>Login form</title></head>
<body>
<html:errors />
<html:form action="/login">
<table border="0" width="200">
<tr>
 <td>User Name:</td>
 <td><html:text property="userid" maxlength="13"/></td>
</tr>
<tr>
 <td>Password:</td>
 <td><html:password property="password" maxlength="13"/></td>
</tr>
</table>
<br>
<html:submit value="Login"/>
</html:form>

</body>
</html>
```

The `<html:form>` tag defines the attribute `action` which is mandatory. This attribute's value specifies the action to be processed by Struts' Controller Servlet. The value must map to a value of the `path` attribute specified for an `<action>` tag in "struts-config.xml" (see listing 5). In the same fashion, the corresponding tags for standard HTML `<input>` (`<html:text>`, `<html:radio>`, `<html:select>` and so on) define a `property` attribute that is also mandatory. This attribute's value *must map a property (it's name) defined in the Java bean* (subclass of ActionForm class) that will be

used both to extract the HTML form field's value and to print the input element's `value` to the client.

We can find the corresponding JavaBean's source code (*step 2*) in listing 2. The bean provides properties for the Login.jsp form (listing 1). The `validate()` method verifies if the userid and password are not null or empty. This method must return an `ActionErrors` object that is, in fact, a collection. If the object returned is null or empty, the `Action` class is called. However, if the `ActionErrors` collection contains some elements (objects `ActionError`) the original page (in this case Login.jsp) is sent back to the client together with the error messages. Error messages are taken from "ApplicationResource.properties" file, by mapping the character string provided as an argument to `ActionError` constructor with parameters defined in that file. Listing 3 pictures the contents of this file for our example.

Listing 2 The ActionForm class for Login.jsp form

```
package databeans;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionError;
import org.apache.struts.action.ActionMapping;
import javax.servlet.http.HttpServletRequest;

public class LoginExistingUserForm extends ActionForm
{
 String userid;
 String password;
 public String getUserid() { return userid; }

 public void setUserid(String newUserid) { userid = newUserid; }

 public String getPassword() { return password; }

 public void setPassword(String newPassword) { password = newPassword;}

public ActionErrors validate(ActionMapping mapping, HttpServletRequest request){
 ActionErrors errors=new ActionErrors();
 if (this.userid==null || this.userid.equals(""))
 {errors.add("user",new ActionError("error.userid.required"));
 }
 if (this.password==null || this.password.equals(""))
 {errors.add("pass",new ActionError("error.password.required"));
 }
 return errors;
 }
}
```

Listing 3. Defining application messages in ApplicationResources.properties file

```
errors.header=<h3>Erori completare formular</h3>
errors.footer=</ul><hr>
error.userid.required=<li>You Must provide an User Name!!!
error.password.required=<li>You Must provide a Password!!!
```

The Struts framework generally assumes that you have created an `ActionForm` bean (that is, a Java class implementing the `ActionForm` interface) for each input form required in your application. If you define such beans in your `ActionMapping` configuration file (see listing 5), the Struts controller servlet will automatically perform the following services for you, before invoking the appropriate `Action` method:

- Check in the user's session for an instance of a bean of the appropriate class, under the appropriate key.
- If there is no such session scope bean available, a new one is automatically created and added to the user's session.
- For every request parameter (all input elements of a HTML form end up as parameters of the `HttpServletRequest` object) whose name corresponds to the name of a property in the bean, the corresponding setter method will be called. This operates in a manner similar to the standard JSP action `<jsp:setProperty>` when you use the asterisk wildcard to select all properties.

- The updated instance of the ActionForm bean will be passed to the Action Class perform() method when it is called, making these values immediately available.

When we code our ActionForm beans, we have to keep the following principles in mind:

- The ActionForm interface itself requires no specific methods to be implemented. It is used to identify the role these particular beans play in the overall architecture. Typically, an ActionForm bean will have only property getter and property setter methods, with no business logic.
- Generally, there will be very little input validation logic in an ActionForm bean. The primary reason such beans exist is to save the most recent values entered by the user for the associated form -- even if errors are detected – so that the same page can be reproduced, along with a set of error messages, so the user need only correct the fields that are wrong. Validation of user input should be performed within Action classes (if it is simple), or appropriate business logic beans.
- Define a property (with associated getXxx() and setXxx() methods) for each field that is present in the form. The field name and property name must match according to the usual JavaBeans conventions. For example, an input field named userid will cause the getUserid() (for printing input element data) and setUserid()(to populate the bean with data typed by user in the input elements of the same form) methods to be called.

We must be aware that a form bean does not necessarily correspond to a single JSP page in the user interface. It is common in many applications to have a "form" (from the user's perspective) that extends over multiple pages. Think, for example, of the wizard style user interface that is commonly used when installing new applications. Struts encourages you to define a single ActionForm bean that contains properties for all of the fields, no matter which page the field is actually displayed on. Likewise, the various pages of the same form should all be submitted to the same Action class. If we follow these suggestions, the page designers can rearrange the fields among the various pages, with no changes required to the processing logic in most cases.

Step 4 states that we should implement an Action class' perform() method. This method will be called after the user submits the form and the controller servlet populates the corresponding ActionForm's properties. Listing 4 provides the source code for our Action class. As you can see, the perform() method does a very simple task: verifies if the user name and password typed by the user match some character strings (of course in a real-world application we will query a database or an XML file). The user name and password data are extracted from the form bean defined earlier, through the form argument that servlet-controller sends to perform() method. Note that we have to down-cast this reference to the original type (LoginExistingUserForm). The perform() method must return an ActionForward object which will tell to the controller servlet what view (JSP page or other resource) to display next. An object of type ActionForward will be obtained if findForward(String pathValue) method is invoked for the mapping object (also a reference obtained as a method argument). The argument pathValue must map to a value of path attribute of the corresponding <action> tag in "struts-config.xml" (see listing 5).

Listing 4. The ActionClass for Login.jsp form

```
package formactions;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionError;
import org.apache.struts.action.ActionErrors;
import java.io.IOException;
```

```
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import databeans.LoginExistingUserForm;
import java.sql.*;
public class LoginExistingUserAction extends Action
{

 public ActionForward perform(ActionMapping mapping, ActionForm form, HttpServletRequest re-
quest, HttpServletResponse response) throws IOException, ServletException
 {
 LoginExistingUserForm loginData=(LoginExistingUserForm)form;
 if (loginData.getUserid().equals("John")&&loginData.getPassword().equals("Pass"))
 return mapping.findForward("successLogin");
 else
 return mapping.findForward("failureLogin");
 }
}
```

Now, all we have left to do is to make these three pieces of application co-operate. This will be done by adding some specific tags to "struts-config.xml" file (listing 5). This file needs special attention because it is the very engine of the whole application. The process goes as it follows:

- If you look carefully at `Login.jsp` page definition (listing 1) you can see that the action of the form points to "/login" path . In struts-config.xml we must define an `<action>` tag with attribute `path="/login"`. All `<action>` tags are nested into a single `<action-mappings>` tag.
- The value of the `name` attribute of `<action>` tag must be the name of the bean specified in `<form-bean>` tag that will provide/undertake input data from the HTML form
- The `input` attribute value must be the JSP page that will provide input form data
- If `scope="session"` that bean will be included in the `HttpSession` object for the user's session, so it will be possible to extract it later from the user request
- The `name` attribute of the `<forward>` tag must have the same character string value as that specified in `Action` class: see for example `return mapping.findForward("successLogin")` in `Action` class.
- The `path` attribute of the same `<forward>` tag must provide the actual path to the document to be sent back to the client.

Listing 5. Putting it all together in struts-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts Configuration
1.1//EN" "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">
<struts-config>
<!-- ====================beans for HTML form data============================== -->
 <form-beans>
 <form-bean name="loginBean" type="databeans.LoginExistingUserForm"/>
 </form-beans>
<!-- ====================Action mappings ================================ -->
 <action-mappings>
 <action path="/login" type="formactions.LoginExistingUserAction" name="loginBean"
 input="/Login.jsp" scope="session">
 <forward name="successLogin" path="/Welcome.jsp" redirect="true" />
 <forward name="failureLogin" path="/Diverse.jsp" redirect="true" />
 </action>
 </action-mappings>

 <message-resources parameter="ApplicationResources" />
</struts-config>
```

Note. Following the Servlet 2.2 API directory structure restriction, all Java classes must be located in the WEB-INF/classes folder. Also remember that everything is case-sensitive.

The source code for Welcome.jsp ca be found in listing 6. You can see that we use the same bean (`LoginExistingUserFOrm`) to extract the user name for printing to the response document. The bean name (`id="loginBean"`) must match exactly the

name attribute of the <form-bean> tag in struts-config.xml in order to get the same object from the HttpSession object of the client request.

Listing 6. Source code for Welcome.jsp page

```
<%@page contentType="text/html"%>
<jsp:useBean id="loginBean" class="databeans.LoginExistingUserForm" scope="session"/>
<html>
<head><title>JSP Page</title></head>
<body>
<h3> Welcome <%=loginBean.getUserid()%>
<br> You have succesfuly loged in
</h3>
</body>
</html>
```

Figure 6 pictures the functionality of this application while figure 7 demonstrates input data validation with validate() method of the form's associated bean.

Fig. 6. Sample login application in action

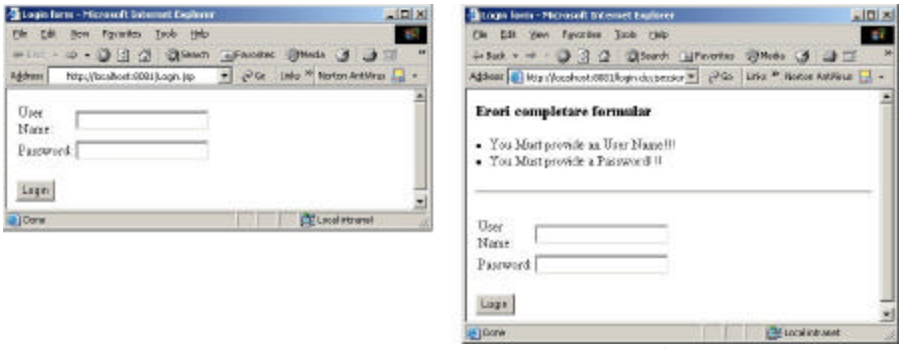

Fig. 7. Validating user input with validate() method of ActionForm bean

Conclusions

The Model-View-Controller design pattern provides a host of design benefits. MVC separates design concerns (data persistence and behavior, presentation, and control), decreasing code duplication, centralizing control, and making the application more easily modifiable. MVC also helps developers with different skill sets to focus on their core skills and collaborate through clearly defined interfaces. For example, a J2EE application project may include developers of custom tags, views, application logic, database functionality, and networking. An MVC design can centralize control of such application facilities as security, logging, and screen flow. New data sources are easy to add to an MVC application by creating code that adapts the new data source to the view API. Similarly, new client types are easy to add by adapting the new client type to operate as an MVC view. MVC clearly defines the responsibili-

ties of participating classes, making bugs easier to track down and eliminate.

Apache's Jakarta Struts project clearly implements the MVC design pattern. The framework is stable, quite simple to use, and helps heterogeneous teams to work together in a very productive environment.

References

Allamaraju, S. (2001). *Proffesional Java Server Programing J2 EE 1.3 Edition*, Wrox Press Ltd.

Budi Kurniawan.(2001). *Java for the Web with Servlets, JSP, and EJB*, New Riders, Indianapolis;

*** http://java.sun.com/j2ee/ . *Java Servlet Specification v2.4, Proposed Final Draft*

*** http://java.sun.com/products/jsp/

*** http://jakarta.apache.org/struts/