CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Bulgarian Digital Mathematics Library at IMI-BAS

HI!MVC: HIERARCHICAL MVC FRAMEWORK FOR WEB APPLICATIONS

Micael Gallego-Carrillo, Iván García-Alcaide, Soto Montalvo-Herranz

***Abstract**: This paper presents Hi!MVC, a framework for developing high interactive web applications with a MVC Architecture. Nowadays, to manage, extend and correct web applications can be difficult due to the navigational paradigm they are based on. Hi!MVC framework helps to make these tasks easier.*

This framework allows building a web based interface, generating each page from the objects that represent its state. Every class to be showed in the interface is associated with two entities: its html representation (view) and its interactions in the view manager (controller). The whole html page is generated by composition of views according to the composition relationship of objects. Interactions between user and application are managed by the controller associated to the view which shows interaction elements (links or forms). Hi!MVC allows building web interface in a hierarchical and distributed way.

There are other frameworks and APIs offering MVC architectures to web applications, but we think that they are not applying exactly the same concepts. While they keep on basing their architectures on the navigational paradigm we are offering a new point of view based on an innovator hierarchical model.

First, we present the main ideas of our proposal. Next, we expose how to implement it using different Java technologies. Finally, we make a first approach to our hierarchical MVC model. We also compare shortly our proposal with the previously cited technologies.

***Keywords**: Web Applications Engineering, Model, View, Controller, MVC, framework, J2EE.*

***ACM Classification Keywords**: H.3.5 Online Information Services: Web-based services; H.5.3 Group and Organization Interfaces: Web-based interaction*

Introduction

When the web was created, its main goal was to provide to the scientific community the chance of sharing information easily. At the beginning, files supporting this information were completely static and it could only be changed modifying the content of the files manually. Nowadays, we find a great evolution at this respect and we can see really complex services offered by web sites. This way, we are able of managing complete applications based on web. This fact is more than interesting for giving the possibility of using an application almost everywhere through a network without installing complicated clients but a web browser.

This is the main reason for enterprises and other organizations to base their development interfaces in web, but this may not be so easy. Patterns and many other design tools look forward to find a way to make software products extensible, manageable, reusable and easy to support. MVC architecture appeared to reach this point for window based interfaces. Now, it seems to be apprehensible to think if it would be possible to take MVC advantages to web based interfaces.

In this paper, we get deeper into a previously proposal [Gallego-Carrillo, 2005] that presented the basic guidelines of Hi!MVC framework. First, we describe the main concepts and organization of our proposed framework. At this point, it can be found that this description would be applicable to any other development tool than Java. Next, we describe in detail the framework implementation. Following, we focus on the hierarchical application of MVC. Finally, related works and conclusions are exposed.

Description of the Hi!MVC architecture

Here, it is presented the base and main concepts of the framework. It shows how to develop web interfaces for applications using this MVC architecture [Krasner, 1998]. This architecture reduces the coupling between classes and increases their cohesion. This fact gets the code to be more independent so it can be easily reused to save time and effort in further developments. MVC has been widely implemented in graphics user interfaces to separate the entity responsible of showing information (view), the one responsible of storing it (model) and the one responsible of receiving user events (controller). The standard UI technology used in java, Swing [Swing, 2005], also uses MVC.

As expected, the design we are writing about has three clearly separated parts, each one of them assume a different responsibility corresponding to MVC model. They are the *model*, the *view* and the *controller*. As follows, we describe them and its *event system*.

Model

This component is the responsible for data maintenance. It has to keep it available for the application in a consistent and safe environment, not allowing any external intervention to affect in any negative aspect. Every single code that implements the model has to be reusable by any other kind of interface without modifying it.

To implement this, we need a set of classes. In order to take advantage of many other technologies, these classes have to preserve JavaBeans specifications [JavaBeans, 2005].

View

View is the responsible for the application interface. In this case, it is the HTML and JavaScript code to send it to the client (or any other format like WML). It has to show the information managed by the application, so it can interact with the user. From here, it has to be given the chance of sending information to the application through events. View has to provide mechanisms like links and buttons to carry out the appropriate requests.

The way to implement this part is through JSP technology [JSP, 2005]. Every JavaBean of the model may have at least one JSP file to be presented. From these files it is used different technologies to access the data to be managed like expression language and tag libraries [Taglibs, 2005].

Controller

It is the responsible of managing the interactions with the user. Whenever the application receives an interaction or data produced by the user, the controller has to decide what to do next. This is the part that manages the global flow control of the application.

From a web browser, there is only one possibility of sending information to the server. This is done sending a request to a web resource. When requesting, the user can attach more information (for example from a form) than the reference to the new resource. There should be one controller by each JSP file. The controller will be a Java common class with methods that receive data about what happened, interpret it and execute its code consequently to continue with the application.

The event system

In order to follow the MVC architecture, the user interacts with the application raising events. An event is the state change notification of view or model. We are only considering in this paper, those events produced by the view. But the problem is that web pages have not the possibility of raising events directly. What the server is able to receive is just the requests from the client, so events are based on them.

When the user clicks a link, it is produced a request to the server. We call to the reception of this request, *real event*, because it is produced at the same moment that the user clicks.

On the other hand, when the user fills the set of components contained on a form, their information is sent all together into the same request when submit button is pressed. The set of changes in the data of the components from what it was shown is what we call *deferred events pack*. We say they are deferred because they are not sent at the same time they are produced. So, it is important to remark that the order or the time in which they are produced can not be considered.

Implementing MVC with J2EE

In this point it is explained how to implement our MVC model and what technologies we are using for it. In Figure 1 we show a scheme of this and in Figure 2 a web application request sequence diagram.

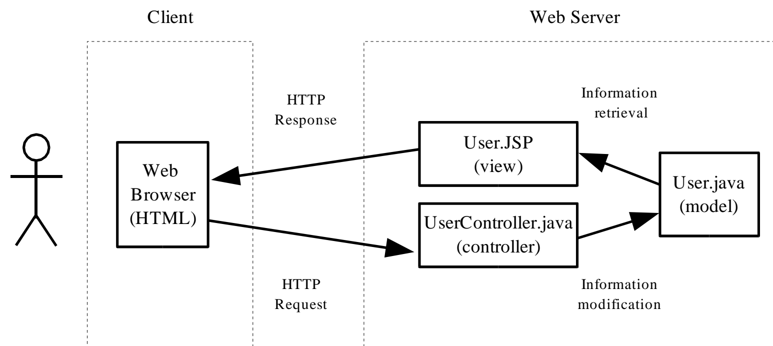

Figure 1. Scheme of design. It shows the collaboration between different layers

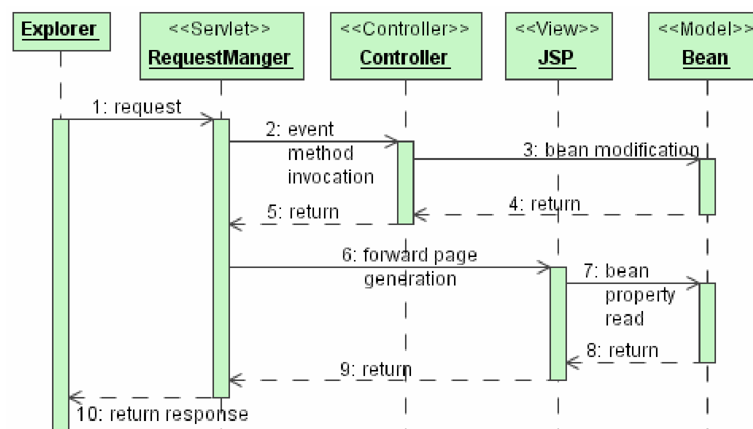

Figure 2. Web application request sequence diagram

Implementing the Model

In the model, we need java classes that support the data to be managed by the application. Fulfilling JavaBeans specification will later help us to be able of using technologies that have JavaBeans as requirement. Information supported by these classes is contained in their attributes, and the type of them can be another class of the same model giving rise a composition hierarchy or an association relation. Hence, basically these classes will have just some attributes and methods to access them. In this component may appear classes related to persistent storage technologies, like databases. In Figure 3 we show a sample JavaBean to implement a bills.

Bill.java

```
public class Bill {
        private String customerName;
        private String desc;
        private int price;

        public String getCustomerName() { return this.customerName; }
        public String getDescription() { return this.desc;  }
        public int getPrice() { return this.price; }
```

```
        public void setCustomerName(String cn) { this.customerName = cn; }
        public void setDescription(String desc) { this.desc = desc; }
        public void price(int prince) { this.price = price; }
}
```

Figure 3. A sample JavaBean to implement bills

Implementing the View

View implementation is mainly based on JSP files that access to model to show it. As classes in the model are JavaBeans, it will be more comfortable to apply Expression Language and User Defined Tag Libraries. One JSP file is always associated to a class in the model and its responsibility is to present the whole or a part of the information containing on it. One class of the model may be associated to one or more JSP files. It will be convenient to associate more than one JSP file, if it is needed, to represent the information with different formats or parts of the associated class.

Commonly, web applications are developed from a *navigational point of view*, namely, there are an independent pages set associated by links included on them. User interacts with the application navigating through these links. In our proposal we consider the web application as the representation in HTML, WML, etc., of an object. So, interactions with the application are made by changing its state and representing it again. We call this *object representation point of view*. This way there is not a home page to navigate to the rest of application. Page generated by JSP has to give the appropriate links and forms to the client so the object state can be changed. Instead of letting developer to write them directly (what would induce to keep navigational point of view) links and forms are generated dynamically and automatically by User Defined Tags. It is needed to pay special attention to the code generated for this, because it will be the responsible of notifying what events are produced and what controller has to listen to them. Further details about this code will be explained later in this point.

In Figure 4 we show two samples of JSPs corresponding to Bill class shows in figure 3. Bill.jsp is used to display bill's information in HTML. Bill.new.jsp is used to modify the state of a bill.

Bill.jsp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://www.lpsi.eui.upm.es/es/upm/lpsi/himvc" prefix="himvc" %>
<%@ page contentType="text/html;charset=ISO-8859-1" language="java" %>
Bill for <strong>${this.customerName}</strong> <br />
Description: ${this.description} <br />
Total price: ${this.price} <br />
```

Bill.Modify.jsp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://www.lpsi.eui.upm.es/es/upm/lpsi/himvc" prefix="himvc" %>
<%@ page contentType="text/html;charset=ISO-8859-1" language="java" %>
<himvc:form id="modify">
   Customer name:
      <himvc:text textId="customerName" text="${this.customerName}" />
   Description:
      <himvc:text textId="description" text="${this.description}" />
   Price:
      <himvc:text textId="price" text="${this.price}" />
</himvc:form>
```

Figure 4. Bill and Bill.Modify views implemented with JSPs corresponding to Bill class

Implementing the Controller

There will be one Java common class by each JSP file that represent a class in the model so, the controller will be a set of classes that receives information from the requests and modify the state of the model according to it. Because it is not necessary to keep the state of these classes, all their methods will be static. Every controller class has one method associated to each link or form that its JSP file is able to generate in order to attend it. Every method will receive one parameter with the object of the model that the JSP used to represent it. If there is additional information on the request like components of a form or parameters of a link, this information will be received by the method. In Figure 5 we can see the BillModifyController class belongs to Bill.Modify view. There is not a controller for Bill.jsp because it haven't any link or form.

BillModifyController.java

```
public class BillModifyController {
   public static void modifyCustomerName(Bill bill, String text){
      bill.setCustomerName(text);
   }
   public static void modifyDescription(Bill bill, String text){
      bill.setDescription(text);
   }
   public static void modifyPrice(Bill bill, String text){
      bill.setPrice(Integer.parseInt(text));
   }
}
```

Figure 5. Bill.Modify controller

It is needed a Servlet that manages requests and determine the correct controller to take control. This Servlet is called *RequestManager* and it is included in the framework. It expects to be able of determine the controller through the URL of the request previously generated by the User Defined Tag. On the other hand, it also expects to receive the form components data to forward them to the corresponding controller. This way the controller knows what events had been produced. So, it can change the model according to this.

Hierarchical MVC architecture

To build the model with object oriented programming, it is often used composition hierarchies between classes. To develop the user interface, it would be very useful to use the same composition hierarchy. To get it, each class in the model has one or more associated JSP files in the view that will take the responsibility of presenting it. If the class to be presented by its associated JPS file has attributes of other classes then, the JSP file may delegate the representation of them on its associated JSP files regarding the composition hierarchy. This is showed in Figure 6. We provide User Defined Tags to make easier the delegation in page generation.

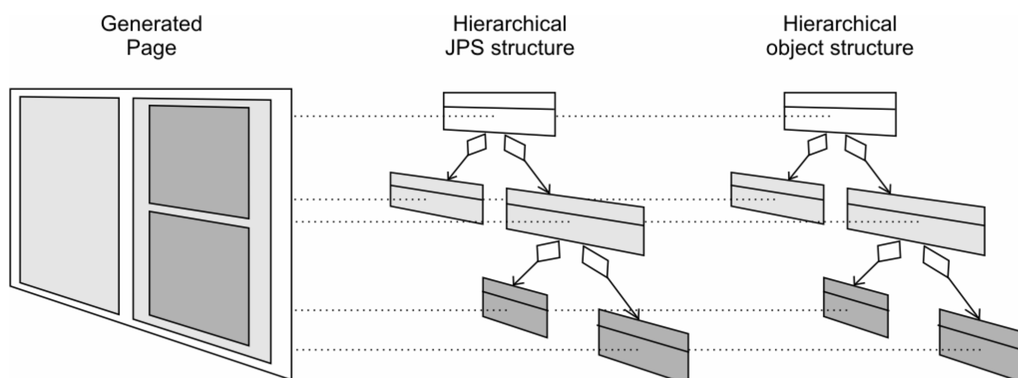

Figure 6. Hierarchical MVC architecture

To show a composition relation between classes in Figure 7 we can see BillsManager model class. This class hold a list of bills and allows to manage the held bills. Figure 8 shows a view for BillsManager. Finally, in Figure 9 we can see the BillsManager controller.

BillsManager.java

```
public class BillsManager {
   List<Bill> bills = new ArrayList<Bill>();
   int changingBill = -1;
   public List<Bill> getBills(){ return bills; }
   public int getChangingBill(){ return changingBill; }
   public void setChangingBill(int cb){ this.changingBill = cb; }
}
```

Figure 7. BillsManager model class

BillsManager.jsp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://www.lpsi.eui.upm.es/es/upm/lpsi/himvc" prefix="himvc" %>

<%@ page contentType="text/html;charset=ISO-8859-1" language="java" %>

<himvc:form id="manager">
  <table>
    <c:forEach var="i" begin="0" end="${this.bills.size()}">
      <td><tr>
        <c:choose>
          <c:when test="${this.changingBill == i}">
              <himvc:include object="${this.bills.get(i)}" type="Modify">
            <br/><himvc:submit id="submit" text="Aceptar"/>
          </c:when>
          <c:otherwise>
              <himvc:include object="${this.bills.get(i)}">
            <br/><himvc:link id="modify" text="Modify" params="${i}"/>
          </c:otherwise>
        </c:choose>
      </td></tr>
    </c:forEach>
  </table>
</himvc:form>
```

Figure 8. BillsManager view

BillsManagerController.java

```
public static class BillsManagerController {
   public static void managerSubmit(BillsManager bm){
      bm.setChangingBill(-1);
   }
   public static void managerModify(BillsManager bm, int index){
      bm.setChangingBill(index);
   }
}
```

Figure 9. BillsManager controller

Related Works

Nowadays, there are several technologies that apply MVC architecture for developing web applications. The most representative in J2EE are Struts [Struts, 2005] and Java Server Faces [JSF, 2005]. Both are very similar in the way they apply MVC architecture.

JSF applies it at page level, so there is one JSP file per page. The model is represented by one object per page as well. The controller is implemented by the own technology and it is configured through defined tags in the JSP file. The controller updates JavaBeans properties with the values found in the interface components generated by the JSP file.

Struts also uses JavaBeans in the model and it implements the view with JSP files that generate the output from those JavaBeans. The controller manages the requests received by the web application. It associates the actions to logic names used to build the links and forms in the JSP file. Both technologies apply navigational paradigm and manage links through configuration files.

Main difference between these technologies and our proposal is that they have the navigation and page concepts as their base. In our approach, the page concept is not applied; instead of, we build a view from an object in the web application. The user does not interact with the application navigating through links but interacting with objects which representation is refreshed every time it changes. Due to this approach, there is not one JSP file per page but when an object needs to be presented then, it will use its own JSP file. If the state of the object is based in other objects, it can delegate the presentation to them including other JSP files.

Conclusion

In high interactive web applications it is not possible to apply in a natural way the navigational paradigm because the user needs to see the state of the application on every moment. The architecture proposed here presents a set of classes that shows their content and modifications at a given time, so not navigational point of view is used in this approach. This fact helps a lot to increase the manageability of the project at developing time. On the other hand, this approach takes good care of encapsulation and reusability benefits offered by classes saving time in the develop process.

In the future, incorporating this architecture into an Integrated Development Environment will enable faster developing giving to the programmer the possibility of generating big parts of his code automatically.

Moreover, building the user interface hierarchically, so it can be used the same delegation as in object oriented technology, makes that repercussions in changing code of the application are very limited.

Next to this work we will continue developing tools, code generators and libraries while going deeper into these concepts to form a complete framework that will help in development of web applications with a high interaction.

Acknowledgement

We would like to thank the following people for their help and their support: Fernando Arroyo Montoro, José Ernesto Jiménez Merino and Carmen Luengo Velasco.

Bibliography

[Gallego-Carrillo, 2005] Gallego-Carrillo M., García-Alcaide,I., Montalvo-Herranz, S. Applying Hierarchical MVC Architecture to High Interactive Web Applications. In: Proceedings of the Third International Conference I.TECH, 2005, pp. 110-115.

[J2EE, 2005] Java 2 Enterprise Edition. http://java.sun.com/j2ee/

[Krasner, 1998] Krasner, G. and Pope, S., 1988. A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 system. Journal of Object Oriented Programming, Vol. 1, No. 3, pp 26-49.

[Swing, 2005] Swing. http://java.sun.com/products/jfc/

[JavaBeans, 2005] JavaBeans. http://java.sun.com/products/JavaBeans/

[JSP, 2005] JavaServer Pages. http://java.sun.com/j2ee/JSP/

[Taglibs, 2005] Tag libraries. http://java.sun.com/products/JSP/taglibraries/

[Struts, 2005] Struts. http://jakarta.apache.org/struts/

[JSF, 2005] JavaServer Faces. http://java.sun.com/j2ee/javaserverfaces/

Authors' Information

Micael Gallego-Carrillo – ESCET, Universidad Rey Juan Carlos, C/Tulipan s/n, 28933 – Móstoles (Madrid), Spain ; e-mail : micael.gallego@urjc.es

Iván García-Alcaide – LPSI, Universidad Politécnica de Madrid, Campus Sur, Carretera de Valencia Km. 7, 28031 Madrid, Spain; e-mail : igarcia@eui.upm.es

Soto Montalvo-Herranz – ESCET, Universidad Rey Juan Carlos, C/Tulipan s/n, 28933 – Móstoles (Madrid), Spain ; e-mail : soto.montalvo@urjc.es

A SENSITIVE METRIC OF CLASS COHESION

Luis Fernández, Rosalía Peña

Abstract*: Metrics estimate the quality of different aspects of software. In particular, cohesion indicates how well the parts of a system hold together. A metric to evaluate class cohesion is important in object-oriented programming because it gives an indication of a good design of classes.*

There are several proposals of metrics for class cohesion but they have several problems (for instance, low discrimination). In this paper, a new metric to evaluate class cohesion is proposed, called SCOM, which has several relevant features. It has an intuitive and analytical formulation, what is necessary to apply it to large-size software systems. It is normalized to produce values in the range [0..1], thus yielding meaningful values. It is also more sensitive than those previously reported in the literature. The attributes and methods used to evaluate SCOM are unambiguously stated. SCOM has an analytical threshold, which is a very useful but rare feature in software metrics. We assess the metric with several sample cases, showing that it gives more sensitive values than other well know cohesion metrics.

Keywords*: Object-Oriented Programming, Metrics/Measurement, Quality analysis and Evaluation.*

ACM Classification Keywords*: D.1.5 Object-oriented Programming; D.2.8 Metrics*

Introduction

The capacity to measure a process facilitates its improvement. Software metrics have become essential in software engineering for quality assessment and improvement. Class cohesion is a measure of consistency in the functionality of object-oriented programs. High cohesion implies separation of responsibilities, components' independence and less complexity. Therefore, it augments understandability, effectiveness and adaptability. Actually, these are major factors of the great interest in using object-oriented programming in software engineering.