2002

A framework for completely separating the presentation and logic of Web-based applications.

Shuling. Nie
University of Windsor

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

A Framework for Completely Separating the Presentation and Logic of Web-based Applications

By

Shuling Nie

A Thesis

Submitted to the Faculty of Graduate Studies and Research

through the School of Computer Science in Partial

Fulfillment of the Requirements for the Degree

of Master of Science at the

University of Windsor

Windsor, Ontario, Canada

2002

ABSTRACT

The use of the Internet and the World Wide Web (WWW) has grown at a phenomenal rate. The Web has become the solution for information exchange, user interaction and business processing. However, building dynamic, interactive and maintainable Web-based applications is not an easy task. Technologies currently used for Web presentation generation do not separate the presentation and logic. This requires people of two different skill sets to be involved at the same time, therefore applications are difficult to develop and maintain. In recent years, several frameworks have been developed to remedy the problem. But they all have some limitations and still cannot completely separate the application presentation from its logic.

The current study attempts to provide an approach of completely separating the presentation and logic of Web-based applications by extending the traditional Model-View-Controller architecture. The research includes two parts. The first part provides a template language for representing Web pages containing dynamic contents without including application logic. The second part provides a framework that integrates the dynamically generated information with the predefined Web templates to generate Web page on the fly.

Key words:

Model-View-Controller, Abstract Model, Abstract Model View, Concrete Model, Presentation, Logic, HTML, XML, Java Servlet, Web-based application, web presentation generation

ACKNOWLEDGEMENTS

I am grateful to my supervisor Dr. Indra Tjandra for giving me an opportunity to work in the exciting field of Web-based application. He is always a source of inspiration to me, and provides tremendous support, assistance and encouragement throughout my studies at the University of Windsor.

I would like to thank the members of my committee, Dr. Purna Kaloni, Dr. Xiaojun Chen, and Dr. Imran Ahmad for their valuable suggestions, supports, comments, and for reading and evaluating my thesis.

I would like to acknowledge The Natural Sciences and Engineering Research Council and The Ministry of Education of Ontario for providing financial support during my graduate study. I would also like to thank the administrative staff from our graduate study and our graduate secretary Mary Mardegan for their kindly help and support.

I extend my gratitude to my family. I would like to thank my husband, Qin Liu for his constant support, care and love. I would like to thank my parents, who have always helped me, encouraged me throughout my studies. I would also like to thank my lovely daughter, Angela Liu, who has been the inspiration and happiness for me.

Finally, I would like to take this opportunity to thank all my friends for their helps, encouragement, and company throughout my studies at the University of Windsor.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1 Introduction

During the past two decades, advances in computer technologies combined with telecommunication technologies have led to the development of the Internet and its most popular application, the World Wide Web (WWW). The use of the Internet and WWW has grown at a phenomenal speed. While originally intended as a medium for distributing information in a document-centric form, WWW has become much more than that. With the introduction of dynamism, both on the client-side and the server-side, the Web has emerged as a new platform for software applications serving a variety of purposes. The multimedia capable, interactive, hardware and software neutral, global reaching, cost effective, and around the clock presence nature of the Web makes it the perfect solution for information exchange, user interaction and business processing.

1.1 Motivation and Objectives

It has become a trend that enterprises use Web to accomplish their day-to-day businesses. However, building a dynamic, interactive and maintainable Web-based application is not an easy task. Interactive business transactions as well as Web sites that are personalized on an individual basis require applications to process user requests and generate corresponding Web pages on the fly. Hyper Text Markup Language (HTML), the publishing mother tongue used on the Web does not handle dynamic content. Using current dynamic Web presentation generation technologies (CGI, Servelts and CGI like technologies), the Web presentation is embedded in the application logic. The application programmer must write

code to explicitly generate HTML code that constitutes the Web page. This makes the development and maintenance very difficult because people of two different skill sets must be involved at the same time. In recent years, several templating frameworks ([ASFa][ASFb][FSF][Sem][Sunb]) have been developed to remedy this problem, where page designers use a template language to develop the presentation. Upon request, the framework interprets the template, integrates the dynamically generated information and produces the Web page. Because all the template languages are HTML based and allow regular HTML being used to develop templates (instead of having application programs to generate them), the logic and presentation are separated in some degree, but they are not completely separated. They also have other limitations. Some frameworks are not very easy to use. Some frameworks use completely new technologies so the page designers often have to put into a great deal of effort in learning the new technologies, instead of relying on technologies and languages that they are used to.

The goal of the current research is to develop a new framework that can effectively and completely separate the presentation and logic of a Web-based application. It includes a very simple template language that allows the page designer to represent Web pages containing dynamic contents totally independent of application logic, but still employs the technologies and software that the designer is familiar with. It also includes a framework that first allows the presentation and the logic to be developed independently. It then allows them to be combined to dynamically construct a Web page that presents to Web users upon request.

The main features of the current research include:

- Extend the traditional Model-View-Controller architecture and define the concepts of Abstract Model and Concrete Model.

- Define the concepts of Abstract Model View (AMV), thus allowing all the presentation related issues to be encapsulated in the abstraction of AMV.
- Introduce a simple template language, consisting of a small number of XML tags, for describing dynamic contents of a Web page, so that the presentation of a Web-based application can be developed totally independent of the application logic, and it allows the page designer to use the technologies and software that he is familiar with (no need for learning brand new technologies).
- Design the framework based on the extended Model-View-Controller architecture. Identify the components of the framework and their collaborations. Develop and test the framework.
- Provide a tool that takes an AMV template file as input and generates a Java skeleton file of the corresponding Abstract Model.
- Provide a tool combining the dynamically generated information and the predefined AMV templates to generate Web pages that are sent to Web clients.

1.2 Organization of the Thesis

This thesis is organized into several chapters.

Chapter 1 gives an introduction of how this research is motivated and the objectives of this research.

Chapter 2 gives an overview of technologies currently used for the Web presentation generation, and frameworks that have been developed in recent years for separating presentation and logics of Web based applications.

Chapter 3 first reviews the Model-View-Controller technology, then extends the traditional MVC architecture, and defines the concepts of Abstract Model, Concrete Model.

Chapter 4 introduces the template language and describes how to develop the AMV template using the template language.

Chapter 5 describes the framework developed, its components, collaborations and designing method.

Chapter 6 focuses on the experiment of using the proposed method, its background and results.

Chapter 7 provides conclusions drawn from the work and recommendations for future directions.

CHAPTER 2 Current Web Presentation Technologies

In this chapter I first review the concept of the Web and Web presentation generation, followed by exploring some widely used, as well as newly emerged Web presentation technologies, their concepts, basic capabilities and constraints. Finally, I conclude with the need for effectively and completely separating presentation from logic in Web-based application.

2.1 World Wide Web Concept

The World-Wide Web (WWW), invented by Tim Berners-Lee in 1989, was initially a practical project intended to bring about a global information sharing ([LCG92]). The Web uses a client -server architecture, with a Web browser sitting on the client side and a Web server sitting on the server side (Fig. 2.1). The Web browser is an application that interacts with both the user and the Web server. Its primary jobs are to transform the user requests into HTTP requests, and transform HTTP responses into graphic presentations that are displayed on the screen. The Web server is an application whose primary jobs are to handle or delegate HTTP requests, and to generate or route HTTP responses. Web clients and Web servers share a set of standards for communication, including addressing scheme, common network access protocols, data formats, which enable any browser to communicate to any Web server on the Internet and make the share of global information become possible.

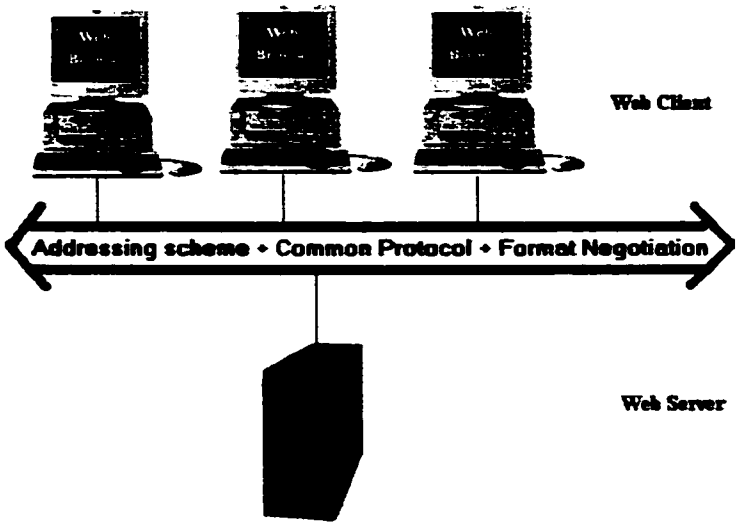

Fig. 2.1 World Wide Web Concept

2.1.1 Static Web Page

The Web was a static medium at the beginning. It served mainly as a document repository with an improved means of navigation. All the Web pages were developed before hand and stored locally. When a Web server receives requests for such web pages, it reads a local file based on the requested URL, and simply streams back the file to the client without any modification (Fig 2.2).

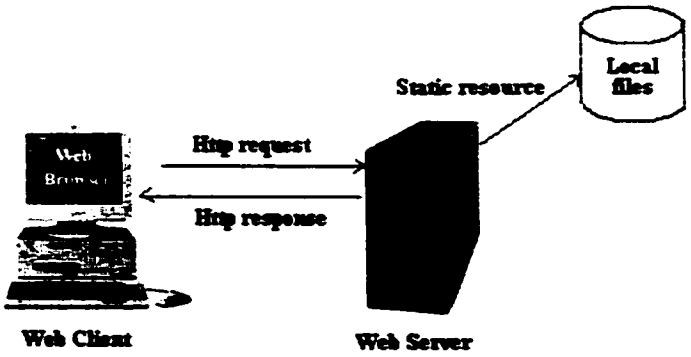

Fig. 2.2 Handle requests to static Web Pages

2.1.2 Dynamically Generated Web Page

With the advent of Common Gateway Interface (CGI) in the National Center for Super Computer Application ([NCSA]), the static nature of the Web changed significantly. Instead of simply retrieving stored documents, a Web page is generated "on the fly" in response to users' interactions. So Web pages can be personalized on individual basis. A modern Web server usually supports various APIs that allow the Web server to delegate HTTP requests to external programs and route Web page generated from external programs to the user (Fig.2.3). The external programs communicate with back-end applications, access and /or update databases that are dynamically generated Web pages in an application-specific manner. A Web server's functionality is extended unlimitedly in this way.

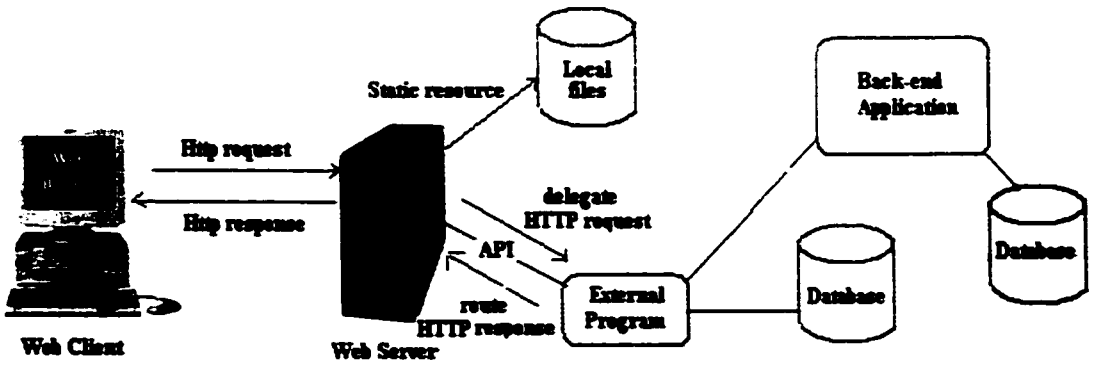

Fig. 2.3 Handle Requests to Dynamically Generated Web Pages

The Web comes to life when it is interactive. Up to now, the ability to deliver dynamic contents has been a prime factor in the growth of the Web and an essential requirement of Web-based application ([Dew98]).

2.2 The Hyper Text Markup Language (HTML)

To accomplish global sharing, information must be represented using a language that all computers can understand. The Hyper Text Markup Language (HTML) is the publishing mother tongue used on the Web ([W3Ca]). HTML is a markup language that is used for typesetting, hypertexting Web page content, extracting, and submitting data from a user to a Web server. It tells Web browsers how information and certain user-interface controls are to be displayed and handled. All Web pages, no matter statically retrieved or dynamically generated, are finally represented in a HTML format. HTML is the basis for all types of Web programming ([EG99]).

HTML uses a set of tags to accomplish its tasks. HTML tags are enclosed within angle brackets, < and >. Most tags have a start tag and an end tag. The start tag marks the beginning of a particular HTML element, while the end tag marks its finish. Enclosed are text or other elements in the Web page. End tags are differentiated from start tags by using a forward slash (/). The tags may have attributes embedded within the beginning tag, in the form of name-value pairs. A HTML tag with attributes looks like this:

<Tag attribname1= value1 attribname2=value2>

Text and/or other tags

</Tag>

Some HTML tags are empty tags, (without any content). These tags are in the form <Tag/>.

Existing HTML tags aim to accomplish two types of functions. The first is to control the structure and display of the Web page. Tags dedicated to this function allow Web browsers to identify components such as

formatted text, tables, images, hyperlinks and frames, and tell the browser how to display various components. For example:

Hello indicates that "Hello" should appear in font Times New Roman with red color.

<P> This is a paragraph </P> indicates that the text should be displayed as one paragraph.

 The university of Windsor indicates that this is a link to the home page of university of Windsor.

The other function provided by the existing HTML tags is that they extract data from a user input and send it back to a server for processing. Two tags used mostly for this function are <form> and <input>. The <form> tag specifies how and where information is to be sent. The <input> tag, which is enclosed inside <form> tag, describes user interface (UI) input components. The UI components include text-box, password, checkbox, radio button etc. A <form> tag looks like this:

```
<form method = "post" action= "http://www.shuling.com/mainServlet">
        <input type = "text" name = "userid" size = "50"/>
        <input type = "password" name = "password"/>
        <input type = "submit" name = "send" value = "submit"/>
</form>
```

This simple form will display two GUI text-box and a button. One of the text-box is for entering a user ID, the other is for entering a password. The "submit" button is used for triggering a HTTP request. When the button is pressed, the HTTP request will be sent using method "post" to the URL specified in "action". The "post" method passes user input from the HTTP

request body. The other method "get" passes user inputs by attaching them in the URL.

The existing HTML tags themselves are sufficient to handle the structure, display of document and the user input extraction. However, HTML is static, it does not offer tags that deal with the dynamic content a Web page. This is because that, firstly, HTML cannot denote a piece of information whose content can only be decided at request time. For example, using HTML, one cannot have the CURRENT time to be displayed because the time to be displayed is decided only when a user makes a request to this page. Secondly, HTML does not support selection. For example, there is no way to express the following scenario: if one wants FRAGMENT1 to be displayed if CONDITION is true, otherwise FRAGMENT2 is displayed. Thirdly, HTML does not support iteration. For example, using HTML one cannot express "display FRAGMENT *n* times".

This research works around the limitations of HTML by introducing a very simple template language. This language is composed of a set of XML tags that are capable of representing the dynamic content of a Web page. The language is discussed in Chapter 4, the development of the AMV template.

2.3 The Common Gateway Interface (CGI)

The Common Gateway Interface (CGI) ([NCSA][TGH96]) is the earliest solution for connecting a Web server and a back-end system, as well as dynamically generating Web presentations. CGI is a standard for interfacing external applications with information servers, such as HTTP or Web servers. This interface allows a Web server to spawn an external process, and pass off a HTTP request for processing. It lets the process to

generate HTML codes that constitute the Web page returned to the client. When a CGI compliant Web server realizes that a HTTP request points to a program instead of a static file, it sets a number of environment variables representing the current state of the server, spawning a new process and passing over the variables and standard input. The process then interprets the input, communicates with the back-end system, and generates the response page. The page is sent back directly to the client or is returned to the Web server for redirection.

CGI has been a popular technique used for extending the server's capabilities and is still widely used today. Its success comes from its ability to communicate with back-end applications written in any language, and its ability to dynamically generate Web presentations. But CGI based applications have some drawbacks. Performance and scalability become big problems due to the need of spawning a new process on the Web server for every client request. A CGI program (and probably also an extensive runtime system or interpreter) needs to be loaded and started for each request. Another problem is security. A CGI script can use the command shell to execute operating system commands with user-supplied data. A bad CGI script can thus crash the whole server. The third problem is that CGI applications are platform-specific. It is very difficult to switch to other servers without modifications. One more problem is that CGI has no direct means of persisting state across application invocations. The programmer has to put significant hand-coding effort into keeping the state of the application and some expensive operations have to be executed repeatedly.

2.4 Java Servlet

The need for Web server to create dynamic Web pages has resulted in the emergence of new technologies for dynamically generating Web presentations. Java Servlet is one of those that are making headlines these days.

Servlet is compiled Java class that can be loaded dynamically into and run by a Web server ([Suna][Hal00][PC00]). It extends the Web server's functionality in a way similar to CGI script. Fig 2.4 shows the physical architecture of Java Servlet. When a Web server realizes that the requested resource is a Servlet, it informs a Servlet container (a container in which a Servlet runs). The servlet container will load the requested Servlet if it has not been loaded yet. The servlet container then creates an instance of the Servlet as a thread, initializes the Servlet and funnels the HTTP request to it. The servlet, which is capable of communicating with back-end system, processes the request and generates a Web page that is send back to the client.

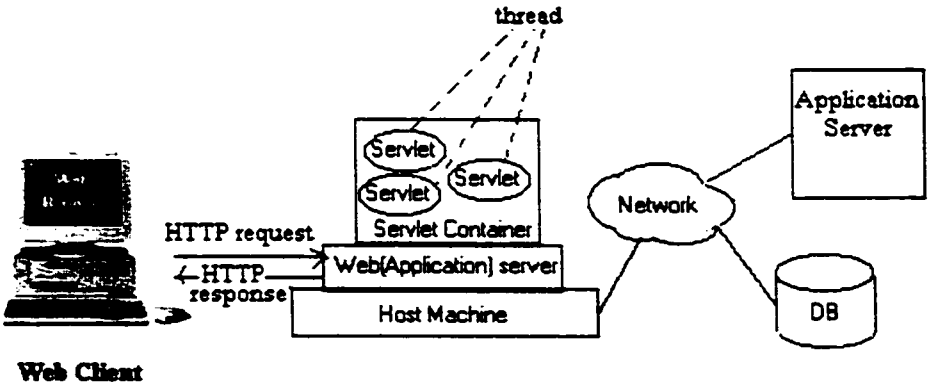

Fig. 2.4 Physical Structure of Java Servlet

There are a lot of advantages of using Servlet than CGI ([KK00]). Servlet has better performances because Servlet runs inside a light-weighted thread

and stays in memory between requests. Fig. 2.5 shows the life cycle of a Servlet. After being created and initialized, a Servlet waits for a request in the memory (in *ready* state). It returns to the *ready* state after finishing processing one request, and is ready to process further requests again without the need of reloading and restarting (like the CGI does). It stays in *ready* state until its *destroy* method is called (usually when the Web server and Servlet container are shut down). This removes the overhead of creating a new process for each request. Servlet also improves security, because Servlet is run in the Java security Sandbox so it can be insulated from disrupting the operating system or breaching security. Moreover, Servlet contains all Java features such as platform independency. It is portable and can be run unchanged on almost every major Web servers. Servlet API supports for session management, which makes written stateful application much easier.

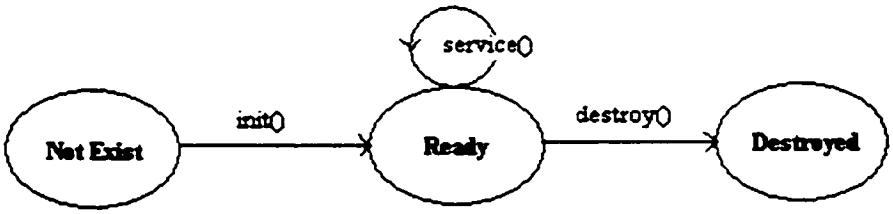

Fig. 2.5 The lifecycle of a Servlet

2.5 Problem with Traditional Servlet, CGI and CGI Like Technologies

As illustrated in Fig. 2.1, a Web server counts on external programs to dynamically produce HTML codes that constitute a Web page. This means that the external programs have two responsibilities. The first responsibility is to process user data and generate the dynamic content of the Web page (the logic part). The second responsibility of the external programs is to

generate HTML codes that constitute the Web page (often need to call *out.println ("<HTMLtag>")* explicitly). In another word, the external program needs to take care of the look and feel (the presentation) of the Web page. Fig. 2.6 is a segment of an external program that dynamically generates a Web page displaying the average income of the current members of a group. The segment includes application logics, which access the database of backend system and calculate the average income of current members. The segment includes presentation. It produces HTML codes that constitute the web page. As we can see, the generation of dynamic Web pages is a mixed process of generating HTML code (the Presentation), accessing, and updating system state (the logic). This makes the development and maintenance of external programs very difficult because programmers of two different skill sets must be involved at the same time. Moreover, it is very difficult to change the appearance of the Web page later.

```
...
{
  ...
  // logic
  int averageIncome = getAverage();
  response.setContentType ("text/html");
  PrintWriter out = response.getWriter();
  // presentation
  out.println ("<HTML>\n" +
               "<HEAD><TITLE>Average Income</TITEL></HEAD>\n" +
               "<BODY>\n" +
               "The average income is:" +  averageIncome +
               "</BODY></HTML>");
}
private int getAverage(){
   // access database and calculate the average
   ...
}
```

Fig. 2.6 Code Segment of an External Program

2.6 Templating Frameworks

Because CGI-like technologies do not separate presentation from logics, there are many solutions developed in recent years attempting to solve this problem. Most of them are Servlet based templating frameworks, where page designers use a template language to develop the presentation. Usually a template is a combination of HTML and one or more of the followings: scripts, standards or user defined tags, and other template language elements. Application logics are embedded in HTML code or are encapsulated using function calls or user-defined tags that are defined outside the template. Upon request the framework interprets the template, integrates the dynamically generated information, and produces the Web page. These frameworks all allow regular HTML being used for presentation and separate the logic and presentation in some degree. But they also have some limitations, as I will discuss in this section shortly.

2.6.1 Tag Based Templating Frameworks

Depending on how the template language is defined, existing templating frameworks can be roughly divided into two categories: tag based or script based. If the template language consists mainly of tags or if it uses mostly user defined tags to encapsulate logic, the templating framework is considered to be tag based. Java Server Page (JSP)([Sunb][AAB00][Hal00][PC00]) and Apache Cocoon ([ASFa]) are examples of tag based templating frameworks.

Extensible Server Page (XSP) is the template language of Cocoon. JSP and XSP use custom tags to encapsulate application logic. The definition of each user defined JSP tag is given in a separate Java tag handler

class. The definition of each user defined XSP tag is given in a separate XSLT (Xml Stylesheet Language for Transformations) stylesheet. Templating frameworks that mostly use custom tags to encapsulate logic have the following drawbacks:

- They are completely new technologies, the page designer will have to give up the old technologies that they familiar with and start from scratch.
- The presentation and logic are still mixed. Because no tags are dedicated to the logic control of HTML code. So HTML fragment (presentation) that needs logic controls (such as selection, repetition control) will have to be embedded in custom tags (logic). So for example, JSP tag handler class still generates HTML code; the custom XSP tags defined in separated XSLT stylesheet still includes HTML fragment. So the maintenance of the application is still very difficult.
- Most WYSIWG (what you see is what you get) HTML editors do not support the development of presentation using existing tag based template language, which makes developing attractive, lively Web presentations much more difficult.

2.6.2 Script Based Templating Frameworks

If the template language consists mainly of scripts, the templating framework is considered to be script based. Velocity ([ASFb]) and Webmacro ([Sem]) are examples of script based templating frameworks. Velocity and Webmacro can separate the presentation and logic a great deal, because their template languages provide selection and repetition logic controls, which can be applied to HTML code in the template, the application programmer no longer need to generate HTML code from

application program. But Velocity and Webmacro have the following limitations:

- The template language of Velocity and Webmacro embed scripts in HTML. They use special characters like $ or # to indicate that an identifier is a template language keyword or a reference. Using this approach, errors can occur easily. For example, if a page designer misses a character accidentally, the keyword or the reference will be interpreted as the content of the Web page. Or if the page designer forgets to escape the special character when he should do, the framework may report errors because it cannot interpret the identifier.
- Using the template languages of Velocity and Webmacro, the dynamic information of the Web page is represented using variables and references to properties and/or methods of Java class. The development of the template still depends on the application logic in some degree. For example, the interface of the Java classes must be developed before hand; the page designer has to be familiar with all the interfaces of the Java class that he will be using, and if the interface changes, the template will have to be rewritten. In this sense, the presentation and logic is not completely separated.

Freemarker ([FSF]) is another templating framework aiming to separate the presentation and logic of Web based application. Its concept is similar to Velocity and Webmacro, having its template language containing selection, repetition and other control logics that can be applied to HTML code. Freemarker is different from Velocity and Webmacro by having its controls denoted using tags instead of scripts. But all the limitations of Velocity and Webmacro still apply to Freemarker, because Freemarker uses script for

reference, and its template still has access to properties or methods of Java class.

2.7 The Need for Effectively Separating Presentation and Logic

As mentioned above, existing solutions do not provide effective ways for splitting presentation and logic for dynamic generation of Web presentations. This makes application development and maintenance difficult and causes poor performance and portability of the application. An approach that can completely and effectively separate the presentation from the logic and relieve page designers of logic burden and application programmers of presentation burden is therefore very needed. The main goal of this thesis is to provide such an approach.

CHAPTER 3 The Extended Model-View-Controller Architecture

This thesis is based on the classic Model-View-Controller (MVC) architecture ([Kas00][KP88]), a three-way separation of functionality among application components. It is first used in Smalltalk for implementing graphical user interface in late 1970's and has been reused and adopted in various GUI class library and application frameworks since.

3.1 The MVC Concepts

The fundamental idea of MVC is to separate the underlying information of the application domain (the Model) from the way that the information is presented (the View), and the way a user interacts with the Model and the View (the Controller). As illustrated in Fig. 3.1, the Controller is the interface between the Model and the View. It is responsible for accepting the user input, choosing the corresponding View, interpreting user gestures, and mapping them to actions that should be performed by the Model. The Model represents the data or knowledge about the underlying application domain. It is responsible for providing data access to the View, self-updating when data changes, and notifying the View and Controller about its state of change. The View is the presentation of the Model. It is responsible for displaying itself, forwarding user gestures to the Controller, and updating itself when the Model changes. By isolating functional units from each other, MVC allows application developers to develop and maintain each particular unit independently, making each unit reusable and pluggable.

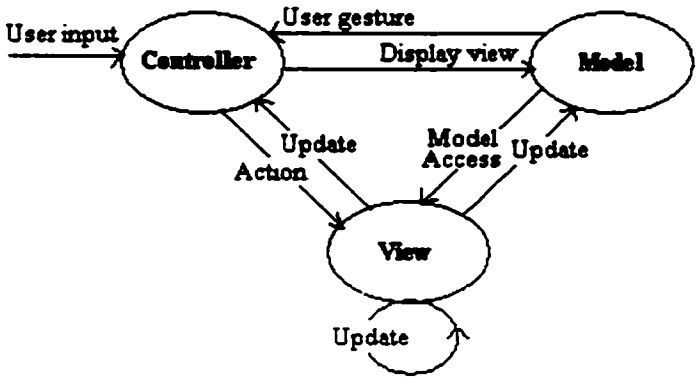

Fig. 3.1 The traditional MVC architecture

A simple example: Let's say that we have a program that can transform a set of data into a diagram for statistic purpose when a user clicks a button or hit a key strike combination. The data here is the model, the diagram is the view, and the key strike combination or mouse click is the controller.

3.2 The Extended MVC Architecture

Using MVC methodology one can easily figure out the reason that traditional Web presentation generation technologies cannot completely separate the presentation and logic. The presentation of Web-based application can be thought of as the View in the MVC architecture. The logic can be thought of as the Model in the MVC architecture. The presentation and the logic are not completely separated because the function units View and Model are not isolated from each other, but both are included in the external program. While trying to apply the MVC architecture in the domain of Web-based application, I found that it was necessary to extend the traditional MVC architecture because the View of Web-based application

and its corresponding Model are different (see below) from the View and Model of traditional usage. The extended MVC (EMVC) architecture is described in Fig. 3.2.

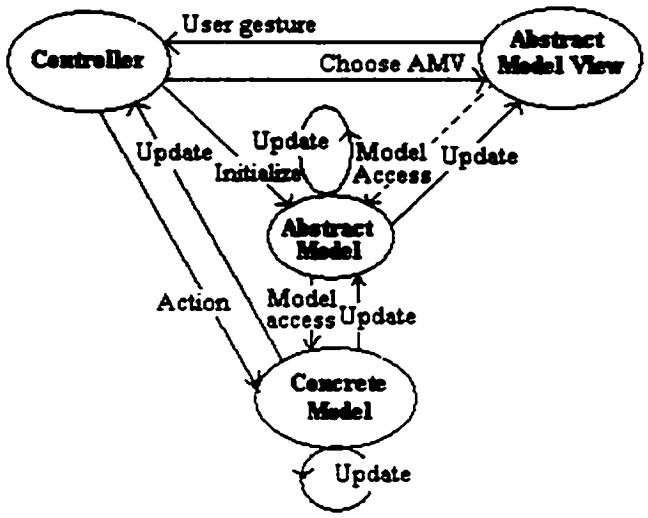

Fig. 3.2 The Extended MVC architecture

3.2.1 Concrete Model and Abstract Model

In the extended MVC architecture, the Concept Model is divided one-step farther into a Concrete Model and an Abstract Model.

A Concrete Model models information that is persistent in the application domain. The information is usually physically stored data, such as customers' accounts, customers' orders etc. The model in a conventional MVC architecture usage in many respects equals to the Concrete Model defined here. The development of the Concrete Model has been studied in depth by many researchers. It is usually done by identifying and implementing the business objects and business process objects of the

domain. Enterprise JavaBeans (EJB) architecture is an excellent standard architecture for the development of a Concrete Model.

An Abstract Model sits on top of the Concrete Model. It models a set of arbitrarily grouped information including references to the Concrete Model, and /or information generated by processing the reference to the Concrete Model. For example, the dynamic information of a Web template of a Web-based application is an Abstract Model. It may include references to the Concrete Model, such as a specific product. It may also include information obtained by processing the Concrete Model (e.g. all the products that are currently in stock). The Abstract Model does not have an actual correspondence in the physical storage. It only exists for a certain time. However the Abstract Model does reflect information inherent in the application domain (at a certain point) and should be separated from its representation.

Because Abstract Model information that only exists temporarily is often arbitrarily grouped, people usually do not realize the fact that the group of information as a whole is also a Model and need to be separated from its view. The Abstract Model is instantiated by the Controller when its View is requested (for example, a user requests a Web page). After that, the Abstract Model updates itself by accessing data of the Concrete Model. It is destroyed when its View no longer needs to be updated.

The reason that existing Web page generation techniques do not completely separate presentation from logic is that they all fail to separate the Abstract Model from its View, where the Abstract Model encapsulates the logic and the View encapsulates the presentation. The main purpose of this thesis is to provide an approach that allows the separation of the Abstract Model from its View in a Web-based application.

If there are well-designed and implemented Concrete Models in hand, development of Abstract Model is easy. The information that constitutes an Abstract Model can be identified by observing the corresponding template. Then the Abstract Model is modeled by implementing methods that produce desired information through accessing information of the Concrete Model.

3.2.2 Abstract Model View

An Abstract Model View (AMV) is the presentation and visualization of an Abstract Model. In Web-based applications, each Web page is an AMV. An AMV can be in one of two states: *template* state or *show* state. An AMV in template state reflects the initial or null state of the Abstract Model. An AMV in show state reflects the updated Abstract Model. When a user requests a dynamically generated Web page (an AMV), the following happen:

- The Controller maps the user request into actions that should be performed by the Concrete Model.
- The Concrete Model performs the actions and updates itself.
- The Controller instantiates an Abstract Model.
- The Abstract Model updates itself by retrieving and /or processing information from the Concrete Model.
- The Controller chooses an AMV (in template state).
- The Controller update AMV by integrating information in the Abstract Model.
- The AMV (in show state) is presented.

3.2.3 Differences Between EMVC and MVC

The major difference between the EMVC and the traditional MVC is that the view presented to the user does not interact directly with the underlying system (the Concrete Model). The interaction between AMV and Concrete Model is encapsulated in Abstract Model. So the AMV (the presentation) and the application logic (the Concrete Model and the Abstract Model) are completely independent of each other and can be developed and maintained separately.

Using EMVC architecture, the development of Web based application can be easily divided into four modules: AMVs that encapsulates presentation; Concrete Model that encapsulates the logic of underlying system; Abstract Model that encapsulates the interaction between views and underlying system; Controller that handles the interaction between AMV, Concrete Model and Abstract Model. By separating the function units from each other, the development and maintenance of presentation and logic of Web-based application can be completely separated. Chapter 4 of the thesis provides a template language for representing AMV that is in template state. Chapter 5 of the thesis provides an application framework developed based on the EMVC. The framework includes the abstractions of the Abstract Model, AMV, the Concrete Model and the Controller, and interactions between them.

3.2.4 The Controller

One of the main responsibilities of a Controller is to interpret a user input according to the control context of an application, and to trigger an action that is to be performed by the Model. In the traditional MVC, one View can have one Model but several Controllers, one for each type of user

interaction. In a Web-based application, all the user gestures are in the form of HTTP request. So for each View (Web page) and Model (Abstract Model) pair there is only one Controller needed. Specifically, views that have the same control context can share the same Controller.

CHAPTER 4 The Development of AMV Template

The EMVC architecture divides Web-based applications into four ways: the Concrete Model, the Abstract Model, the AMV, and the Controller. Due to the availability of more advanced techniques and methodologies, development of the Concrete Model has become easier. Development of the Abstract Model has also become straightforward. However, the template languages of existing templating frameworks do not provide an easy solution that allows the logic to be excluded completely from template. The goal of this chapter is to develop a simple template language that allows page designers to develop AMV templates without including any application logic, and without giving up the technologies and software that they have used for years for Web page designing.

4.1 HTML For Pure Presentation

The content of a dynamically generated page (the AMV) can be divided into two categories. The first type is related to pure presentation such as the page layout, multimedia usage etc. This is the format of the Web page. The other type is the information that should be generated dynamically. It is the direct reflection of an Abstract Model. The whole Web page can be thought of as information presented in a format specified.

This thesis chooses to use HTML for pure presentation due to its several advantages:

- HTML is simple

  HTML is a very simple language, with a limited set of tags. It is easy to learn and many Web developers have mastered it.

- HTML is efficient for presentation

  HTML has been used from the time the Web was invented. It has a set of well-designed tags proven to be sufficient to handle the presentation of a Web page.

- HTML is widely supported

  There are now a lot of software available for HTML based Web site design and development. Most of them are very powerful and easy to use. The Macromedia Dreamweaver, for example, has a WYSIWYG editor that allows the developer to rapidly develop the look and feel of a Web page without worrying about HTML code. It also provides an easy JavaScript integration, CSS definition and Web site management.

Here is how my method works. The Web developer first develops a static sample Web page using whatever useful tools (for example, a WYSISWG HTML editor) in HTML. Then he makes the page XML compliance using XML editor (such as XML spy). The Web developer then replaces the information that should be generated later using the template language given in this thesis. He also needs to provide a summary of all logic related information of the template (using a <declaration> tag), so that the skeleton of the Abstract Model can be easily constructed using a tool (PageHandler Skeleton Generator or PHSG) provided in the framework. The resulting template is an AMV in template state. The application programmers are responsible for developing Concrete Model and Abstract Model. The given framework takes care of the underlying logic of how to trigger the update of the state of the system as well as trigger updating the AMV from template state to show state.

4.2 XML Compliance

Extensible Markup Language (XML)([Mar99][Sunc][W3Cb]) is a markup language that is fast becoming the standard for describing and exchanging data. XML and HTML are both markup languages. The major difference between them is that HTML is specific, whereas XML is general. HTML uses a restricted set of tags that tell the browser how to display the data marked, while XML allows users to develop their own tags to indicate the nature of the data. This thesis uses XML to define a set of new tags that are used to represent the dynamic content of a Web page in AMV templates.

Another merit of XML is that there is a series of standards, APIs (Application Programming Interface) and software for processing XML documents. By making the AMV template XML compliance, I can extract information from an AMV template by using a XML parser without having to interpret XML syntax. I can also modify the AMV template easily with existing APIs. The task of integrating dynamically generated information and the AMV template thus becomes very simple.

The following are standards that this thesis comes across:

- DOM (Document Object Model)

  DOM API is used to convert a XML document into an object-oriented hierarchical representation (a DOM tree). It allows the data to be removed, modified or inserted randomly.

- SAX (Simple API for XML)

  SAX API uses an event-driven approach whereby the developer registers handlers with a SAX parser. The parser invokes the callback methods whenever it encounters a new XML tag. This API is used for serial accessing of XML documents.

- DTD (Document Type Definition)

  A DTD specifies a set of elements that allow to be shown in the XML document and the relations between the elements.
- JAXP (Java API for XML parsing)

  JAXP provides a common interface for creating and using the standard SAX, DOM, and XSLT (Extensible Stylesheet Language for Transformation) APIs in Java. JAXP includes four packages: *javax.xml.parsers* provides a common interface for different vendor's parsers; *org.w3.dom* defines classes for components of a DOM; *org.xml.sax* defines the basic SAX APIs; *javax.xml.transform* defines APIs that transform XML into other forms.

4.3 The Template Language

The template language is XML based. The idea is to represent pure presentation using XHTML (extensible HTML or HTML that comply with XML syntax), and represent dynamic content with a small set of predefined XML tags. The sample HTML pages are first edited using XML editor like XML Spy to make them XML compliance. Then the new tags introduced are used to replace the dynamic content in the template. Unlike other tag based template languages (JSP, XSP) that use custom tags to encapsulate application logic, all the tags introduced are predefined, not application specific, and can be used just like other HTML tags. The tags introduced here can be divided into four categories: structure tags, declaration tags, information tags and logic control tags.

4.3.1 Structure Tags

The structure tags are used to identify characteristics of a document. It indicates which elements are included and how to process them. Two structure tags defined here:

- **<template> Tag**

A <template> tag identifies that the document is an AMV template. It is the root element of every AMV template. It contains two elements: the <declaration> tag and the <html> tag. The <declaration> tag contains information that will be used to construct an Abstract Model (section 5.5). An example of <template> tag:

```
<template>
      <declaration>
        …
      </declartion>
      <html>
        …
      </html>
</template>
```

- **<declaration> Tag**

A <declaration> tag is used to identify a part of a document that contains information that is needed to construct an Abstract Model. This tag acts as a contract between the page designer and the application programmer. The page designer uses this tag to communicate with the application programmer about the dynamic information of the template. The skeleton of an Abstract Model can be generated by using tool PHSG that is capable of interpreting this tag. A <declaration> tag can include declaration tags (section 4.3.2) <attrib>, <param>, <var> and <loop>. A <declaration> tag looks like:

```
<declaration>
      <attrib …/>
      <param…/>
```

```
    <var …/>
    <loop…/>
</declaration>
```

4.3.2 Declaration Tags

Declaration tags are tags that can be included inside <declaration> tag. They are used to describe information about the AMV template. They can be one of the following tags.

- **<attrib> Tag**

A <attrib> tag is used to encapsulate information that is inherent to the AMV, but should not be seen by Web users. It has three attributes, *name*, the name of the attribute, *value*, the value of the attribute, and *description*, which is used to add comments on the attribute. For example:

```
<attrib name = "requireLogin" value = "true" description= " whether the
use need to be logged in to see this page/>
```

- **<param> Tag**

A <param> tag is used to indicate a parameter whose value can be retrieved from a HTTP request. It usually represents form data that is used in the Web page. It has two attributes: *name*, the name of the parameter, and *description*, which gives the comment. It looks like this:

```
<param name = "fair_or_not" description= "the value of a checkbox
indicates whether the visitor thinks the grade is fair"/>
```

- **<var> Tag**

A <var> tag is used as a declaration for a piece of unknown information (a variable) in the AMV template. It has four attributes: *name*, *loopid*, *loopvarid* and *description*. The *name* attribute denotes an identifier that represents the unknown information. The *loopid* attribute denotes the position of the loop in which the unknown information is defined. Loops in an AMV template are numbered according to the order they appear, starting

from 0. The value of *loopid* is such a number. If the variable is outside any loop, the value of *loopid* is -1. The *loopvarid* attribute denotes the position of the unknown information within the loop, in which the unknown information is defined. All the unknown information that is defined in a loop is numbered according to the order they appear, starting from 0. The value of *loopvarid* is such a number. The value is -1 if the variable is outside any loop. Please see section 5.4 for detailed explanation of how this works. An example:

```
<var name= "studentGA" loopid= "-1" loopvarid="-1" description= "all
the student who have grade A"/>
```

- **<loop> Tag**

A <loop> tag is used to represent an iteration used in the AMV template. For each loop used inside the <html> tag, there must also be a <loop> element in the <declaration> tag. The <loop> tag has four attributes. The *subject* attribute denotes the subject of the loop. It refers to a list-variable defined earlier. For example, the subject for loop "for all the students with grade A" is "students with grade A". The value of the subject is the name of an already defined variable (e.g. studentGA). The *varNum* indicates the number of variables in this loop. The *id* attribute is used to denote the ID of the loop, and the *comment* attribute is used for making comments about the loop. For example:

```
<loop subject= "studentGA" varNum= "2" loopid = "0" comment ="This loop
go through all the student with grade A"/>
```

4.3.3 Information Tags

Information tags are place-holders for unknown information used in <html> tag. They represent the initial or null state of an Abstract Model. Same information tags used inside <html> share one <var> entry in the

<declaration> tag. There are several information tags defined, all of them have four attributes: *name*, *loopid*, *loopvarid* and *description*. The definitions of these attributes are the same as those in <var> element of <declaration> tag.

- **<conditionVar> tag**

A <conditionVar> tag is used inside of a <if> tag. It represents a variable whose value is used to decide the flow of control. If it is finally evaluated to be true, then the <then> fragment will become a part of the final AMV to be shown. Otherwise the <else> fragment will be displayed. For example:

```
<if>
      <conditionVar name= "female" loopid= "-1" loopvarid= "-1"/>
      <then> Ms.</then>
      <else> Mr. </else>
</if>
```

- **<attribVar> Tag**

Sometimes, the value of an attribute in a HTML tag can also be dynamic. For example, the URL of a product in stock. A <attribVar> tag is used to represent such a value. A <attribVar> element is used between the beginning tag and the ending tag of a HTML tag whose attribute value is undecided. <attribVar> tag has one more attribute *attrib*. The value of *attrib* is the name of the attribute (whose value is undecided) in the parent tag of <attribVar>. For example:

```
 <a href="?"><attribVar name= "itemurl" attrib="href" loopid="-1"
loopvarid="-1"/></a>
```

- **<var> Tag**

A <var> tag inside <html> tag reserves spaces for general variables.

Same information tag can be used more than once inside a <html> tag, if it denotes the same information. For example:

```
<b>
 Cheer up,
<var name ="studentName", loopid = "-1", loopvarid = "-1">!
 Good luck,
<var name ="studentName", loopid = "-1", loopvarid = "-1">!
</b>
```

4.3.4 Logic Control Tags

Logic control tags are tags that used inside the <html> tag that provide logic control information when updating an AMV from template state to show state. They are mainly for two types of logics: the selection logic and the repetition logic.

- **<if> Tag**

A <if> tag encapsulates selection logic. It contains three elements: a <conditionVar> element defined earlier, a <then> element, and a <else> element. If <conditionVar> is evaluated to be true later, the <then> segment will become a part of the final view of an AMV, otherwise the <else> segment will. For example:

```
<if>
      <conditionVar name= "female" loopid= "-1" loopvarid= "-1"/>
      <then> Ms.</then>
      <else> Mr. </else>
</if>
```

- **<then> Tag**

A <then> tag contains a segment that is a part of the final view if the condition is true. The segment used here and in the following text may include one or more logic tags, information tags and other html tags defined by W3C (World Wide Web Consortium).

- **<else> Tag**

A <else> tag contains a segment that is a part of the final view if the condition is false.

- **<foreach> Tag**

A <foreach> tag encapsulates repetition logic. It contains segments that may appear more than once in the final view. It has one attribute *controlVar*, whose value is the name of an already defined list variable. This attribute is used to denote which subject that the loop is working on. Here is the example:

```
<H2> The following students have a grade of A:</H2>
   <table>
      <foreach controlVar= "studentGA">
         <tr>
            <td><var name="studentName" loopid="0" loopvarid="0"/>
            </td>
            <td><var name="studentGrade" loopid-"0" loopvarid="1"/>
            </td>
        </tr>
     </foreach>
   </table>
```

The <html> tag in the AMV templates can contain two types of elements, one type of tags are those defined by W3 Consortium, the other type of tags are those that are just defined.

4.4 A Sample AMV Template

The following is an AMV template that is defined using HTML and the newly introduced tags.

sample.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<template>
   <declaration>
      <attrib name="requireLogin" value="true"/>
      <param name="fair_or_not" description="the value of the checkbox
indicate whether the visitor thinks the grade is fair"/>
      <var name="date" description="date of today" loopid="-1"
loopvarid="-1"/>
      <var name="requesterIP" description="where is this request come
from" loopid="-1" loopvarid="-1"/>
      <var name="studentGA" loopid="-1" loopvarid="-1" description="all
the student who have grade A"/>
```

```
      <var name="female" loopid="0" loopvarid="0" description="whether
the student is female"/>
      <var name="studentname" loopid="0" loopvarid="1" description="the
name of the student"/>
      <var name="studentgrade" loopid="0" loopvarid="2"
description="the grade of the student"/>
      <var name="studenthomepage" loopid="0" loopvarid="3"
description="the homepage of the student"/>
     <loop subject="studentGA" varNum="4" id="0" comment=" this loop go
through all the students with grade A"/>
   </declaration>
   <html>
      <head>
         <title>My Homepage</title>
      </head>
      <body>
         <H1> Hello! There</H1>Today is :
         <b>
            <var name="date" loopid="-1" loopvarid="-1"/>
         </b>
         <p> This request is from IP : <var name="requesterIP"
loopid="-1" loopvarid="-1"/>
      </p>
      <p>Students with grade A in this class are:</p>
      <table>
        <foreach controlVar="studentGA">
         <if>
            <conditionVar name="female" loopid="0" loopvarid="0"/>
            <then>
               <tr bgcolor="#CC33CC">
                  <td>Ms. <var name="studentname" loopid="0"
loopvarid="1"/>
                  </td>
                  <td><var name="studentgrade" loopid="0"
loopvarid="2"/>
                  </td>
                 <td>
                    <a href="?">
                          <attribVar name="studenthomepage"
attrib="href" loopid="0" loopvarid="3"/>Her home page
                           </a>
                  </td>
               </tr>
             </then>
             <else>
                <tr bgcolor="#3333FF">
                   <td>Mr. <var name="studentname" loopid="0"
loopvarid="1"/>
                   </td>
                   <td>
                      <var name="studentgrade" loopid="0"
loopvarid="2"/>
                   </td>
                   <td>
                      <a href="?">
                          <attribVar name="studenthomepage"
attrib="href" loopid="0" loopvarid="3"/>His home page
```

```
                    </a>
                  </td>
                </tr>
              </else>
            </if>
          </foreach>
        </table>
            <form action="displayvote">
              <p>I think it's
                 <select name="fair_or_not"size="1">
                      <option selected="true" value="fair">Fair
                      </option>
                        <option value="notfair">Not Fair</option>
                     </select>
                 </p>
              <input type="SUBMIT" value="Submit"/>
            </form>
        </body>
    </html>
</template>
```

CHAPTER 5 The Framework

After defining the template language that can be used to develop Web pages containing dynamic contents without including application logic in chapter 4, I concentrate on the framework development in this chapter. The design of the framework is based on the EMVC architecture introduced in chapter 3. The purpose of the framework is to map function units of a Web-based application into components whose development and maintenance can be modularized, and to provide the logic for handling interactions between the components. Comparing with other Servlet based templating frameworks such as Velocity, Freemarker, Webmacro, the significant feature of the framework is that it provides a single entry point for all the requests to Web pages that have the same control context. This means there is no need to write and deploy one Servlet for each Web page. The Controller (the single entry point servlet) will intercept HTTP requests, choose proper AMV template and generate the requested Web page. I will first give an overview of the framework, its components and their collaborations. I will then give detailed implementation of each component.

5.1 The Components and Their Collaboration

The framework is composed of several components (Fig. 5.1): *abstractController*, *concreteController*, *abstractModelUpdator*, *concreteModelUpdator*, *ConcreteModel*, *concretemodel*, *PageHandler*, *concretePageHandler*, *ExceptionMap*, *Composer*, and *PageHandlerSkeletonGenerator*. Among which, the *abstractController* together with the *concreteController* are the direct map of the *Controller* in the EMVC. The *PageHandler* together with the *concretePageHandler* are

the direct map of the *Abstract Model* of EMVC. The *ConcreteModel* together with the *concretemodel* are the direct map of the *Concrete Model* of EMVC. *ConcreteModel* is an interface with no methods or fields and serves only to identify the semantics of being Concrete Model. The *abstractModelUpdator* together with the *concreteModelUpdator* are responsible for updating the Concrete Model. The *ExceptionMap* is used for exception handling. It is used to map a particular exception to a Web page to redirect to the Web client. The *Composer* is used by *AbstractController* to combine information generated from a *concretePageHandler* and the AMV template to produce the final page. The *PageHandlerSkeletonGenerator* is a tool used for constructing skeleton of *concretePageHandler* (Abstract Model) from an AMV template.

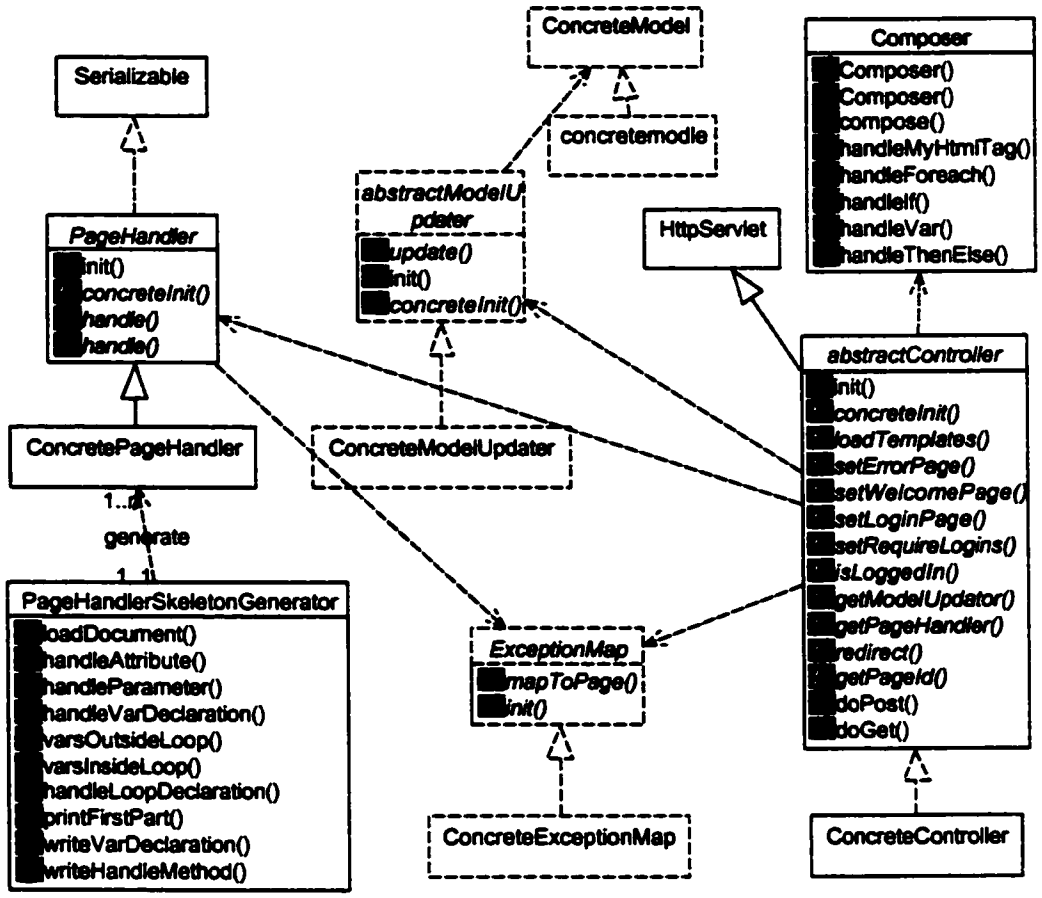

Fig. 5.1 The Components of the Framework

The collaboration of these components is:

- The *concreteController* is a generalization of the *abstractController*, the *concretePageHandler* is a generalization of the *abstractPageHandler*, the *concreteModelUpdator* is a generalization of the *abstractModelUpdator*. The abstract classes whose implementations are given in the framework provide general logics that may be applied to different applications. Operations whose logics are related to a particular application are declared to be *abstract* in

abstract classes. The user of the framework provides concrete implementation of these methods to handle logics that are related to a specific application in concrete classes.

- The *PageHandlerSkeletonGenerator* (see section 5.5) parses all the Web page templates (the AMVs) and generates skeletons of the *concretePageHandlers* for these templates. Application programmers implement the *concretePageHandler* based on the skeleton.

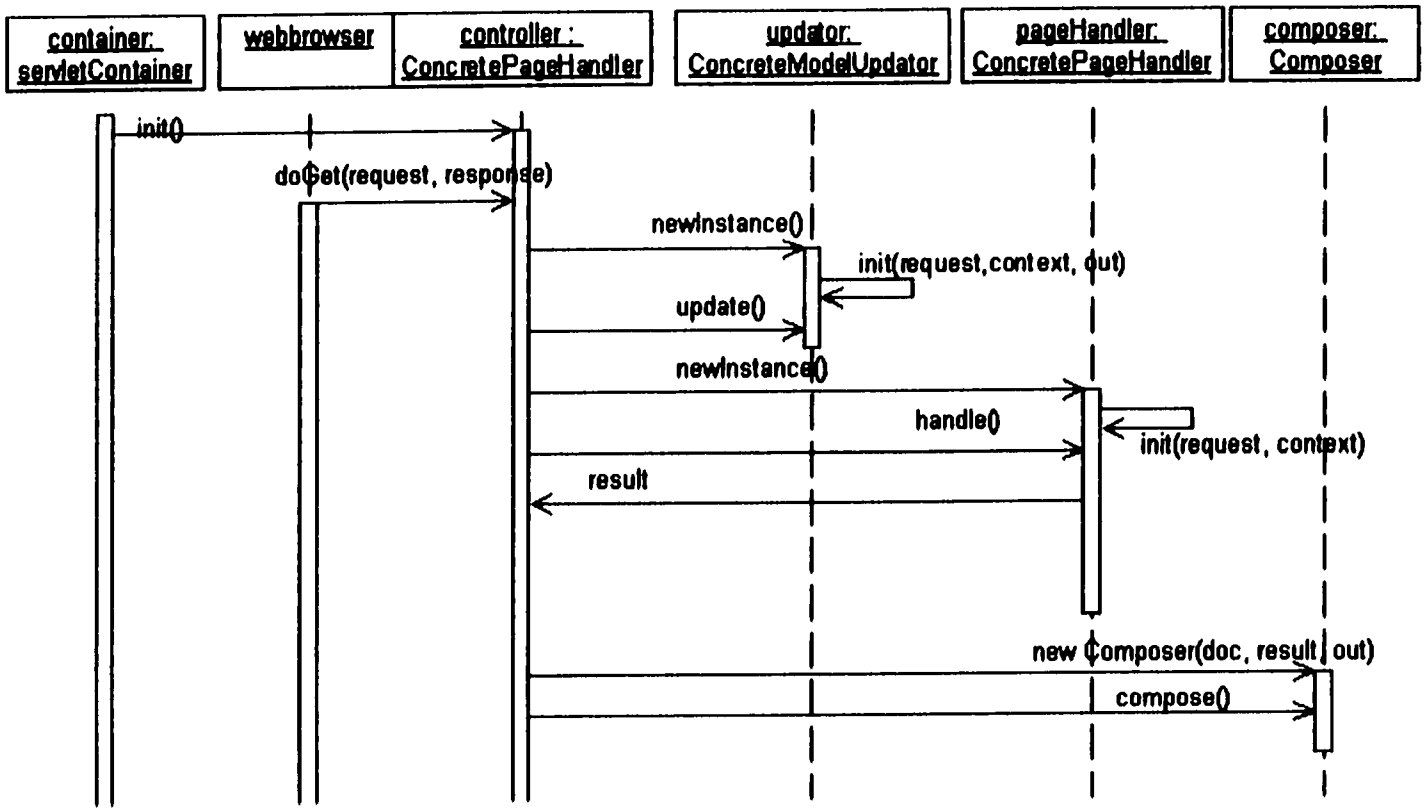

Fig. 5.2 Collaboration of the Components

Fig. 5.2 shows the interaction between the components.

- The Servlet container instantiates and initializes the *concreteController*. The *concreteController* loads all the AMVs into its memory. It optionally instantiates a *concreteExceptionMap*, and stores the *concreteExceptionMap* object in its context.
- The *concreteController* intercepts a Http request. If the page that a user requests has a corresponding model updator, the *concreteController* instantiates a *concreteModelUpdator* and asks it to update corresponding *concretemodels* (by calling its *update* method).
- The *concreteController* instantiates a *concretePageHandler*, initializes it with the HTTP request and Web context, and asks it to update the Abstract Model (by calling its *handle* method). If there are exceptions thrown during the updating of the Abstract Model, the *concretePageHandler* can ask the *ExceptionMap* to map any exception to a Web page to redirect.
- The *concretePageHandler* returns a *Hashtable* that contains all the dynamically generated information from the *handle* method.
- If the *concretePageHandler* has not specified a redirect page, the *concreteController* picks up an AMV template, passes the AMV and the *Hashtable* object returned by the *concretePageHandler* to the *Composer*. The *Composer* integrates the AMV with information in the *Hashtable* object and prints out the Web page (the final view of AMV) to the Web page requester.

- If the *concretePageHandler* has specified a redirect page, the *concreteController* will redirect the user's request to the specified page.

5.2 abstractController and concreteController

The *abstractController* is actually a front end Servlet. It is a single entry point for all the requests to pages that have the same control context. Upon request, it is responsible for triggering the updating of Concrete Model and Abstract Model, choosing appropriate AMV template, transforming the AMV template using information from Abstract Model, and generating Web pages that are sent to the client. Operations related to control context of a particular application are declared to be abstract in *abstractController*. The user of the framework is supposed to provide a *concreteController* and implement all the abstract methods inherited from *abstractController*.

As stated in section 3.2.3, for Web based applications, Views with the same context can share the same controller because the user View interaction and the Abstract Model AMV interaction are the same. Since all gestures of the user appear as Get or Post method of HTTP request, there needs a way of identifying which view the user is requesting. This is done by embedding a *pageid* in each HTTP request. A *pageid* name value string pair needs to be hand coded in each hyperlink, for example http://server/controllerpath?pageid=id. If a form is used, a hidden field *pageid* can be used to carry the *pageid* information. The *pageid* information is used by the *abstractController* to choose model updator, page handler and the AMV template.

Fig. 5.3 is the class diagram of *abstractController*. There are seven fields used in the class. *templates* is a *Hashtable* object. All the AMV

templates are parsed and loaded in memory as XML DOM *Document* objects through method *loadTemplate()*, when the *concreteController* is initialized. *templates* is used to map a page ID string to an *Document* object that represents a AMV. The original *Document* tree obtained from parsing AMV template can be modified so that HTML node becomes the root of the document tree. *requireLogins* is also a *Hashtable* object. It maps a page ID string into a boolean value that indicates weather or not the user should log in to see the requested page. It is initialized through method *setRequireLogin()*. *welcomePage*, *loginPage* and *errorPage* are page ID strings that are used to specify which page to be redirected in a special situation (such as general error, requiring log in). They are initialized through a corresponding *set* method. *exceptionmap* is a *ExceptionMap* object that knows how to map a particular exception to a page to redirect (using method *mapToPage()*). It is an optional component for the framework. If the exception handling logic is simple, the application does not have to have *concreteExceptionMap* implemented. *context* is an *ServletContext* object that encapsulates the Web context of the controller. The seven fields together form the general context of the Web tier of a Web-based application. They are all initialized at the time the Servlet engine loads the *concreteController* (by calling the *concreteController's init* method). The init method does application specific initialization by calling *concreteController's concreteInit* method.

The *getModelUpdator* method is used to create and initialize a *concreteModelupdator* if there is a model updator exists for the requested page (model updator is not needed if a request to a page does not change the underlying system). The *getPageHandler* method is used to create and initialize a *concretePageHandler*. The factory method design pattern

([GoF94]) is used here for the creation of *concreteModelUpdator* *concretePageHandler*. Usually the *concreteController* implementer can store two Hashtable objects in *context* through *concreteInit* method. One is used to store a page id and *concreteModleUpdator* class name string pair. The other is used to store a page id and *concretePageHandler* class name string pair. *concreteController* decides which model updator or page handler to load by mapping the page id to a class name, then initializes it using *getClass().getClassLoader().loadClass(*classNameString*).newInstance();*

doGet() method (Fig. 5.3) controls the flow of the Web-based application. When the user makes a request, the *concreteController* extracts the page ID using method *getPageId*. If the requested page requires log in, the user will be directed to a login page if he has not logged in already. The *concreteController* triggers Models to update themselves by calling *upate* method on the *concreteModelUpdator* object and *handle* method on a *concretePageHandler* object. If the *concretePageHandler* does not specify a page to redirect the user, the *concreteController* will choose an AMV template from the memory. It then asks a *Composer* to update the AMV and prints out the final view to the Web user.

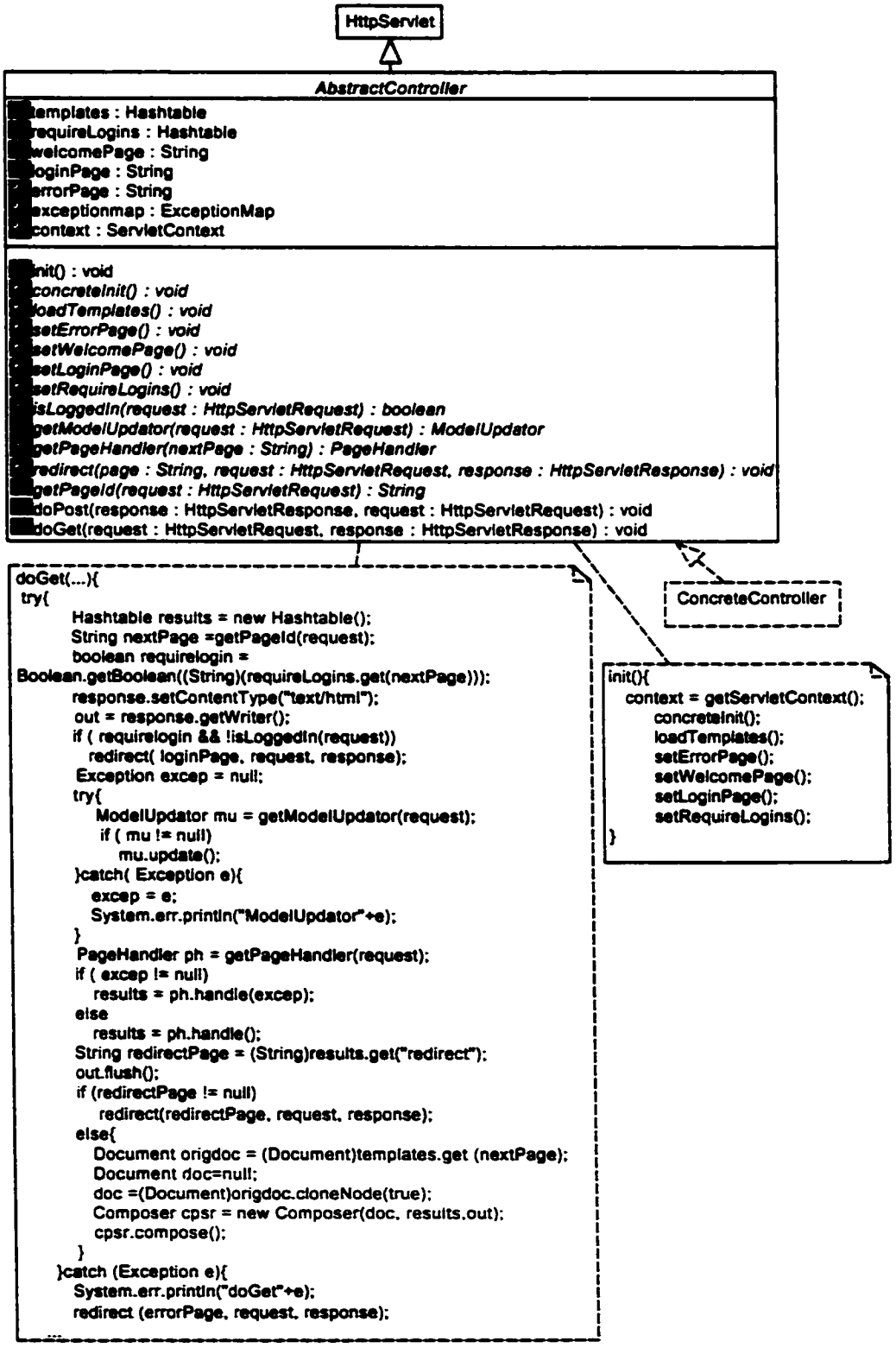

Fig. 5.3 The Class diagram of abstractController

5.3 The Command Design Pattern

The *concreteController* is responsible for mapping the user requests to actions that can be performed by the model. However, it is undesirable for the *concretecontroller* to be aware of details of which operations are to be performed by which models. A Command design pattern is used here to solve this problem.

Design patterns represent proven solutions to specific problems arisen during object oriented software design. The landmark book Design Patterns ([GoF94]) defines design patterns as "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context". Fig. 5.4 is a UML diagram that depicts the participants of a Command pattern.

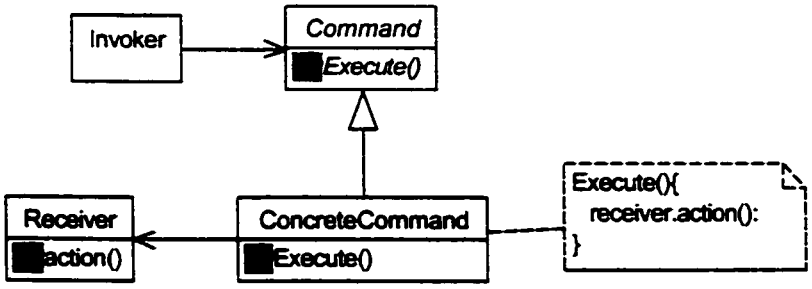

Fig. 5.4 The Command Pattern

In the Command pattern, instead of letting an invoker to invoke actions from a receiver (remember different requests may need to invoke different actions from different receiver), a *Command* interface is used to encapsulate a request to the receiver. The binding of which receiver should

perform which action is defined in *Execute()* method of *ConcreteCommand*, a generalization of Command interface. This enables the invoker to send requests to different receivers using a uniform interface.

ModelUpdator and *PageHandler* are both designed using the command pattern, where *ModelUpdator* and *PageHandler* are interfaces (*Command*) that encapsulate requests. The controller calls the *update()* method on *ModelUpdator* to trigger actions be performed by appropriate Concrete Models. The *concretecontroller* calls *handle()* method to let the Abstract Model invoke appropriate methods to retrieve the dynamic information.

5.4 The ModelUpdator

Model updator (Fig. 5.5) is used to encapsulate the request for updating underlying system. The *concreteModelUpdator* implementer is responsible for providing concrete *update* method, which extracts data from user requests and invoke actions that can be performed by *concretemodels*. Method *concreteInit* method is used to include application specific initialization of *concreteModelUpdator*. The application programmer needs to provide one *concreteModelUpdator* for each template, to which the request will result in changes in the state of the underlying system. If the request to a page does not change the state of the underlying system, Model updator is not needed.

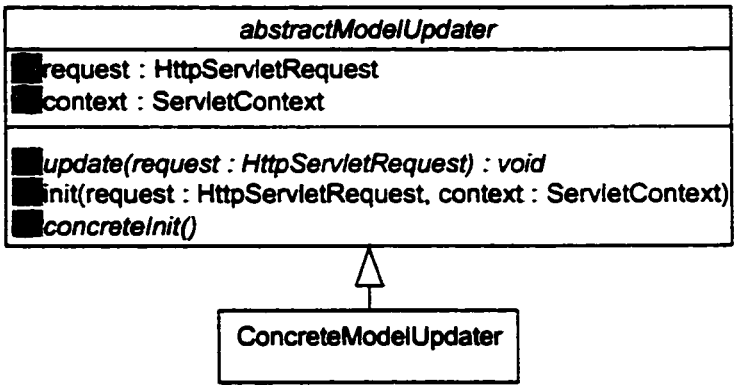

Fig. 5.5 The Class diagram of abstractModelUpdator

5.5 The PageHandler

A *PageHandler* is the direct map of the Abstract Model of an EMVC. It is responsible for modeling the dynamic information of an AMV template. It is also the interface that encapsulates the request for updating the Abstract Model.

The information that a *concretePageHandler* needs to model can be divided into two categories. The first type of information is used inside of a loop (called *varInsideLoop*) in an AMV template. The second type of information is used outside of any loop (called *varOutsideLoop*) in the AMV template. The *concretePageHandler* models these two types of information in such a manner that the information can be directly associated with a particular reserved space in the AMV template without ambiguity.

Consider modeling the dynamic information of AMV template in the following diagram (Fig. 5.6). This template is used to generate a Web page about the information of the Universities in the city that the Web user lives.

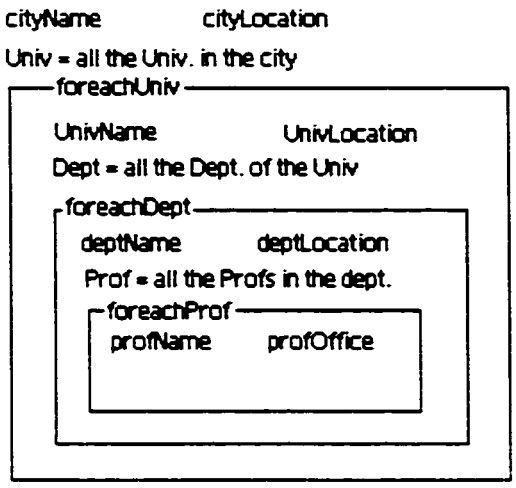

Fig. 5.6 A graphic representation of an AMV template

At the time of a request, all the unknown information in the AMV has been decided. Modeling *varOutsideLoop* is straightforward. A unique name inside the AMV template can identify a piece of unique information without any problem. For example, if I request this page, *cityName* refers only to Akron, *cityLocation* refers only to northeast Ohio. I can use *Univ[0]* to identify the University of Akron, and use *Univ[1]* to refer to the Kent State University etc. But, let us assume that we want to denote the information "the name of a professor in a department at a university in the city where the Web user lives". How can this be accomplished? If we use a Multi-dimensional array, then the question becomes how many dimensions are needed? What if the AMV template has ten or more levels of nested loop?

To denote any unknown information in an AMV template, I use a three-way numbering system. First I number each loop in the AMV template (Fig.5.7), according to the order that they appear in the AMV template, starting from 0.

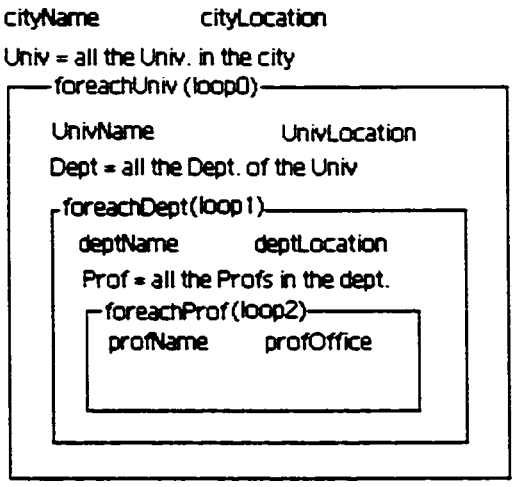

Fig. 5.7 Number Loops

- Second I number all the unknown information defined inside a loop. The information is defined in the loop if its appearance in the loop is also the first time in the AMV template. Information defined inside a loop is numbered according to the order they appear, starting from 0 (Fig. 5.8).

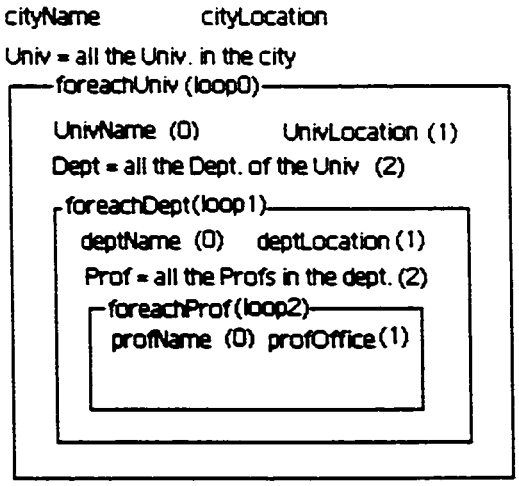

Fig. 5.8 Number Variable inside Loop

By using this method, all *varInsideLoop* of an AMV template can be represented using a three dimensional array: *loopVars[loopPosition][variablePosition][index]*. In the given example, *loopVars[2][0][x]* can be used to denote a professor's name in a department at a university in the city where the Web user resides. The *loopid* attribute of all the information tags and <var> declaration tag gives the position of the loop that the unknown information is defined. The *loopvarid* attribute gives the position of unknown information within that loop.

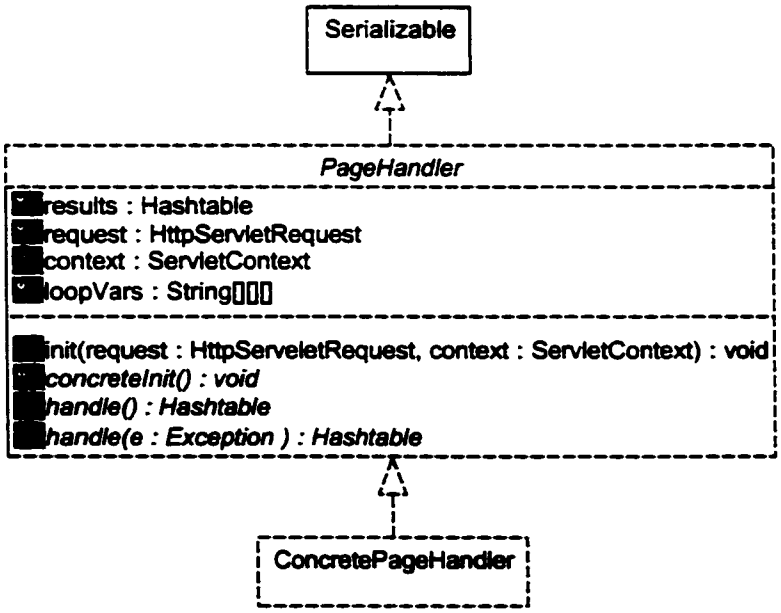

Fig. 5.9 The class diagram of PageHandler

Fig. 5.9 is the class diagram of *PageHandler*. *ConcreteController* calls the *handle* methods of *PageHandler* to obtain the dynamic information of an AMV template. Each *handle* method returns a Hashtable object *results*, which encapsulates all the dynamically generated information. *results* can contain three types of information: *varInsideLoop*, *varOutsideLoop*, and a

page to be redirected. For each *varOutsideLoop*, there will be a name (information identifier) value (the value evaluated for the information) pair entry in *results*. If there are *varInsideLoop*s, there will be one name (*loopVars*) value (a three dimensional array) pair entry in *results*. If the *concretePageHandler* decides that the user should be redirected to another page, there will be one name (*redirect*) value (the page id to redirect to) pair entry in *results*.

There are two *handle* methods defined in *PageHandler*. One of which takes an *Exception* object as an argument. The exception is thrown during the Concrete Model updating. Because the exception is also a gesture that affects the Abstract Model, the Abstract Model can perform proper actions (such as set appropriate page to redirect) based on the nature of the exception.

5.6 The PageHandlerSkeletonGenerator

In section 4.3.1, each AMV template contains a declaration element. This element acts as a contract between the Abstract Model and the AMV. It helps the Abstract Model implementer to identify which information needs to be modeled, and provides information that will affect the modeling, such as the parameters and attributes of the Web page. By doing this, the Abstract Model developer is isolated from the presentation of the Web page. The current study provides a tool *PageHandlerSkeletonGenerator* (PHSG) that extracts information from an AMV template and generates a skeleton for the *concretePageHandler* Java class. The PHSG generates several entries in the skeleton file based on elements in the declaration part of an AMV template.

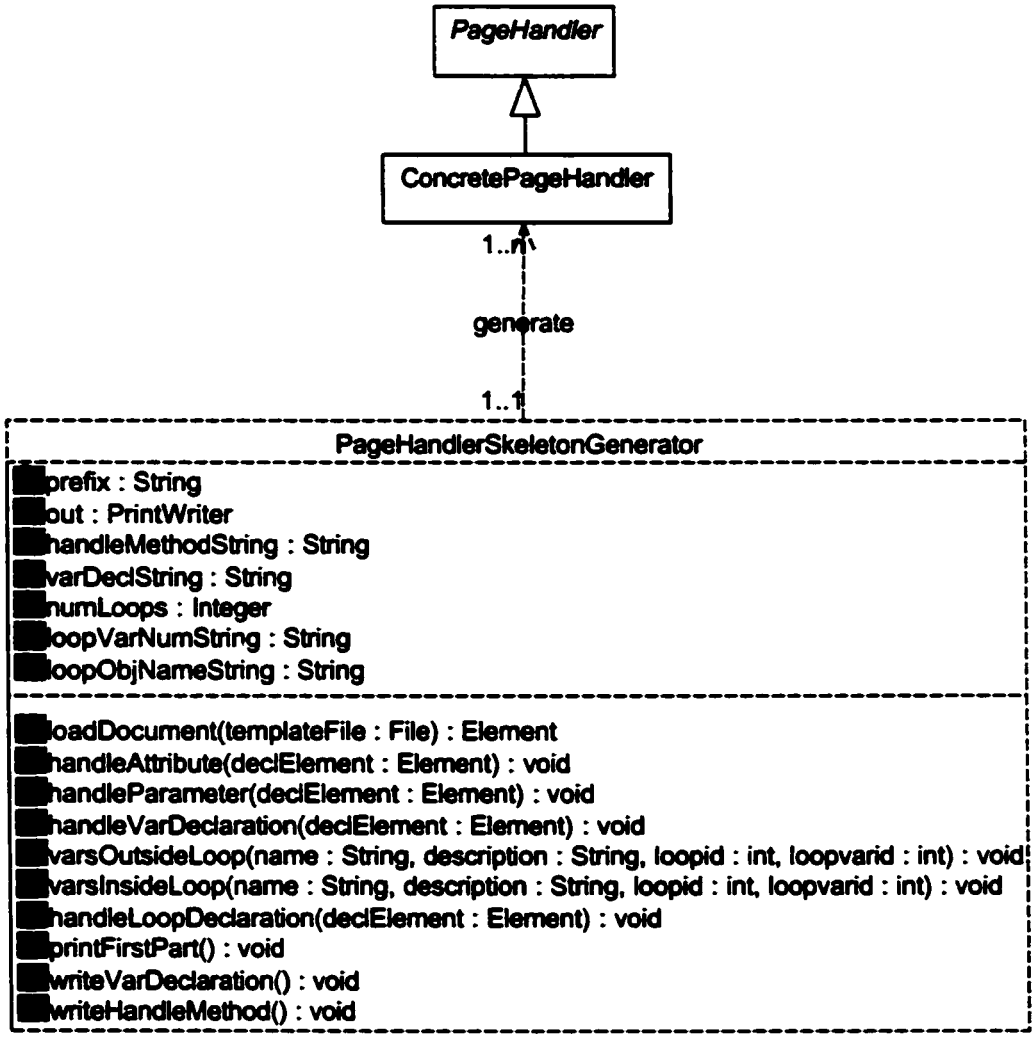

Fig. 5. 10 The class diagram of PHSG

Fig. 5.10 is the class diagram of PHSG:

- Method *varsOutsideLoop()* deals with element

```
<var name= "varname" loopid="-1" loopvarid="-1"/>
```

in the declaration. It generates one method signature:

```
private void set_varname(){
    String key ="varname"
    // get value
    results.put(key, value);
```

```
}
```

This method will be called from the *handle* method to obtain the value of one *VarOutsideLoop*(whose name is *varname*)

- Method *varsInsideLoop()* deals with element

```
<var name= "varname" loopid="x" loopvarid="y"/>,
```

where x not equals to –1, y not equals –1. It generates one method signature:

```
public String get_LoopxVary (Integer index){
   String value=...;
   return value;
}
```

This method will be called from the *handle* method to obtain value of Of one *VarInsideLoop* (which is in loop *x*, position *y* with index *index*)

- Method *handleAttribute()* deals with element

```
<attrib name= "attribname" value="attribvalue"/>
```

in the declaration. It generates one declaration entry:

```
String attribname = attribvalue;
```

- Method *handleParameter()* deals with element `<param name="paramname"/>` in the declaration. It generates a declaration:

```
String paramname;
```

It also generates an entry in method *concreteInit*:

```
Paramname = request.getParameter("paramname");
```

- After processing all the *loop* elements in the declaration using method *handleLoopDeclaration()*, the PHSG writes several declarations in the skeleton, for example:

```
int numLoops =1;
int[] loopVarNum ={2};
//This array stores the control object of each loop
String[] loopControlVarName={"customer"};
```

Where *numLoops* is used to denote the number of loops in the AMV template; *loopVarNum[i]* is used to store the number of variables used inside loop i; *LoopControlVarName[i]* is used to store the name of the variable that controls the iteration of loop *i*.

- The skeleton for *handle()* method is also generated using method *writeHandleMethod()*. It includes two parts:

  - To invoke method to get value from any variables outside any loop and map the name of that variable to its value in *results* (the AMV).

  For example:

```
set_var1();
set_var2();
...
```

  - To invoke a method to get the value of *varInsideLoop* if there are some. This is done by first declaring a three dimensional varying length array *loopVars*, then setting the value of each element in the array through a method call *get_loopxVary* and finally putting the *loopVars* name value pair into *results*. The following is the generated segment for handling loop.

```
//---------------------Array declaration---------------
   String[][][] loopVars = new String[numLoops][][];
     for (int i=0; i<loopVars.length; i++)
       loopVars[i] = new String[loopVarNum[i]][];

     for(int i=0; i<loopVars.length; i++){
       Object[] subjects =((Object[])
                    results.get(loopControlVarName[i]));
       int loopObjLength;
       if ( subjects != null)
         loopObjLength = subjects.length;
       else
```

```
        loopObjLength = 0;
       for (int j=0; j<loopVars[i].length; j++)
        loopVars[i][j]= new String[loopObjLength];
     }

//-------------------set value of array element------------
  for(int i=0; i<loopVars.length; i++)
    for (int j=0; j<loopVars[i].length;j++)
      for (int k=0; k<loopVars[i][j].length; k++){
        Class[] formalArg = { new Integer(k).getClass()};
        String methodName ="get_LoopiVarj;
        Method method =null;
        try{
          method = this.getClass().getMethod(methodName,
                                              formalArg);
          Object[] actualArg ={new Integer(k)};
          if (method != null)
            loopVars[i][j][k]=(String)
                        (method.invoke(this, actualArg));
        }catch(Exception e){
            System.err.println( e);
        }
      }
   results.put(\"loopVars\",loopVars);
```

As can be seen from the above description. The basic logic of *concretePageHandler* has been generated by the PHSG. The following are a list of things that the implementer does while implementing the class:

- Providing method bodies and adding more methods and fields if necessary
- Modifying *handle()* method accordingly (add some data flow logic).

The following is the skeleton that PHSG generated for the sample AMV template mentioned earlier:

========== sample.Java ==========

```
import shuling.thesis.Webappframework.PageHandler;
import Java.lang.reflect.Method;
import Java.util.Hashtable;

public class sample extends PageHandler{
   /* date of today Loopid=-1 loopvarid=-1 */
   private void set_date(){
      String key = "date";
         ...// get value
         ...//results.put( key,value);
   }
```

```
   /* where is this request come from Loopid=-1 loopvarid=-1 */
   private void set_requesterIP(){
      String key = "requesterIP";
         ...// get value
         ...//results.put( key,value);
   }
   /* all the student who have grade A Loopid=-1 loopvarid=-1 */
   private void set_studentGA(){
      String key = "studentGA";
         ...// get value
         ...//results.put( key,value);
   }

   /* whether student is female Loopid=0 loopvarid=0 */
   private String get_Loop0Var0(Integer index){
      String value=...
      return value;
   }

   /* the name of the student Loopid=0 loopvarid=1 */
   private String get_Loop0Var1(Integer index){
      String value=...
      return value;
   }

   /* the grade of the student Loopid=0 loopvarid=2 */
   private String get_Loop0Var2(Integer index){
      String value=...
      return value;
   }

   /* the homepage of the student Loopid=0 loopvarid=3 */
   private String get_Loop0Var3(Integer index){
        String value=...
      return value;
   }

   Hashtable results = new Hashtable();
   String requireLogin = "true";

   //Param to be extracted from request:
    //the value of the checkbox indicate whether the visitor think the
grade is fair
   String fair_or_not;

    int numLoops =1;
   int[] loopVarNum ={4};
   //This array stores the control object of each loop
   String[] loopControlVarName={"studentGA"};

   public void concreteInit(){
      fair_or_not = request.getParameter("fair_or_not");
      ...
   }
```

```
  public Hashtable handle(){
     set_date();
     set_requesterIP();
     set_studentGA();

     String[][][] loopVars = new String[numLoops][][];
     for (int i=0; i<loopVars.length; i++)
        loopVars[i] = new String[loopVarNum[i]][];
     for(int i=0; i<loopVars.length; i++){
        Object[] subjects =((Object[])
                 results.get(loopControlVarName[i]));
        int loopObjLength;
        if ( subjects != null)
           loopObjLength = subjects.length;
        else
           loopObjLength = 0;

        for (int j=0; j<loopVars[i].length; j++)
           loopVars[i][j]= new String[loopObjLength];

     }

     for(int i=0; i<loopVars.length; i++)
        for (int j=0; j<loopVars[i].length;j++)
           for (int k=0; k<loopVars[i][j].length; k++){
              Class[] formalArg = { new
                              Integer(k).getClass()};
              String methodName ="get_Loop"+i+"Var"+j;
              Method method =null;
              try{
                 method =
            this.getClass().getMethod(methodName, formalArg);
                 Object[] actualArg ={new Integer(k)};
                 if (method != null)
                    loopVars[i][j][k] =(String)
                       (method.invoke(this, actualArg));
              }catch(Exception e){
                  System.err.println( e);
              }
           }
        results.put("loopVars",loopVars);
    // set redirect...
    ...
    return results;
    }

     public Hashtable handle(Exception e){
          ...
          return results;
    }
}
```

5.7 The Composer

The AMV template and the dynamically generated information (encapsulated in *result* returned from *handle* method) need to be integrated in order to generate the final view. The *concreteController* does this by calling the *compose()* method of a *Composer* object. The *concreteController* first calls composer's constructor *Composer(Document doc, Hashtable results, PrintWriter out)* to create a *Composer* object. Where *doc* is a DOM object that is constructed from an AMV template, *results* is the Hashtable object returned from the handle method of *concretePageHandler*, *out* is a *printWriter* that can be used to write to clients. The basic idea is that *Composer* modifies the DOM tree by replacing the unknown information with data from *results*, transforms the modified DOM tree to XML format, and sends to the client through *out*.

In the provided framework, all AMV templates are parsed and loaded as XML DOM trees in the memory, where the templates are modified slightly so that the root of each document tree is a HTML element (<html>). The following diagram (Fig. 5.11) shows nodes that may appear in the DOM tree and their potential child nodes (if any):

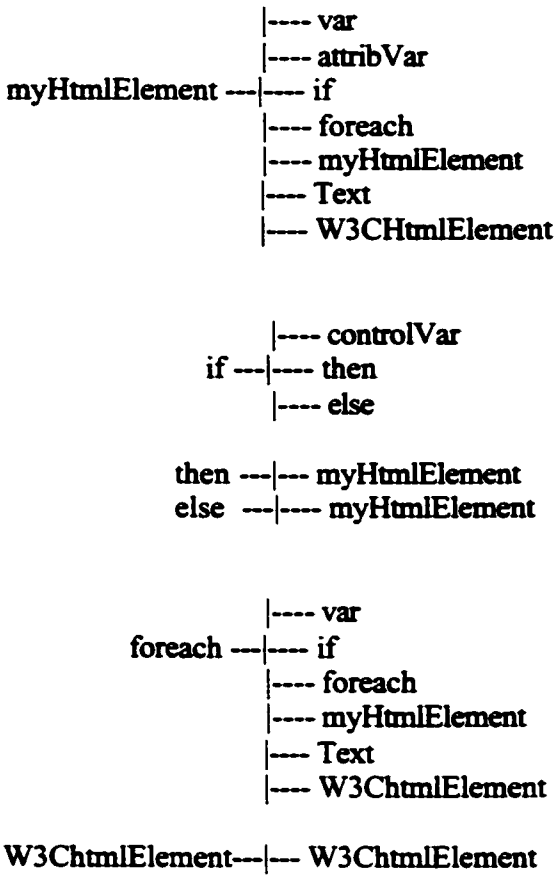

Fig. 5.11 The Nodes and Their Children of an AMV DOM Tree

In the diagram, *W3ChtmlElement* and *myHtmlElement* both refer to tags that can appear in a HTML document according W3C's HTML DTD. The difference is that *W3ChtmlElement* can only have *W3ChtmlElement* or *Text* as child nodes, while *myHtmlElement* can in addition have tags that I introduced in section 4.3 as child nodes.

Here is how the composer accomplishes the integration: starting from the root, the composer recursively modifies each child node until there are only *Text* nodes and tags defined by W3C.

The document itself is a *myHtmlElement.* The composer modifies it by going through all of its child nodes:

- If a child node is a *Text* node or a *W3CHtmlElement*, the *Composer* skips it and begins to exam the next child node (Fig. 5.12).

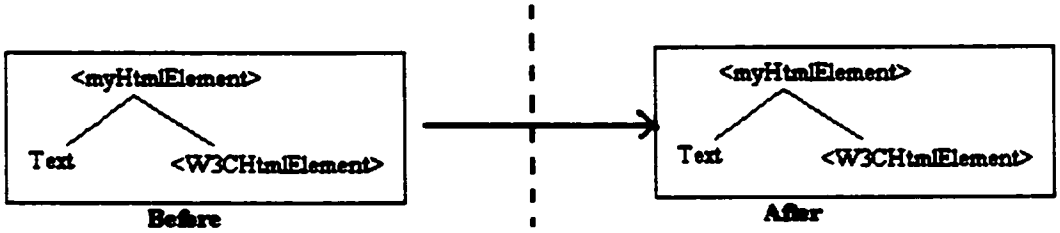

Fig. 5.12 Text and W3ChtmlElement as Child Node

- If the child node is a *var* node, the composer retrieves the value that corresponds to the *var* from *results* (the *Hashtable* object returned by *concretePageHandler*). It constructs a *Text* node based on the value, and it replaces the *var* node with the *Text* node (Fig. 5.13).

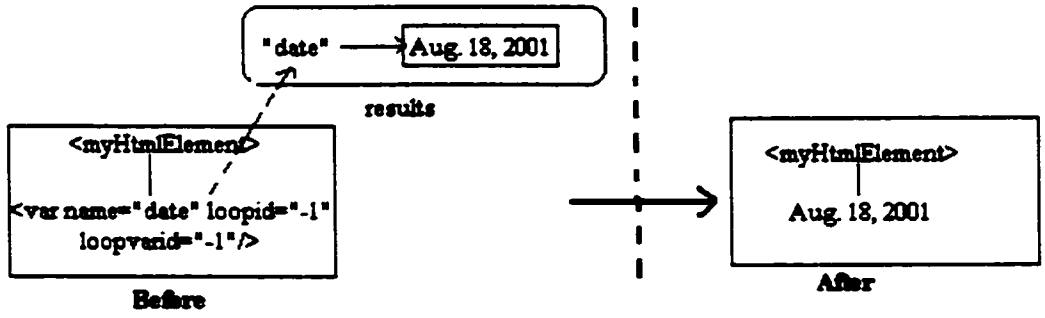

Fig. 5.13 Var as Child Node

- If the child node is an *attribVar* node, the composer gets the value corresponding to the *results*. It sets the value of the corresponding attribute of its parent, it then removes the *attribVar* node from the parent (Fig. 5.14).

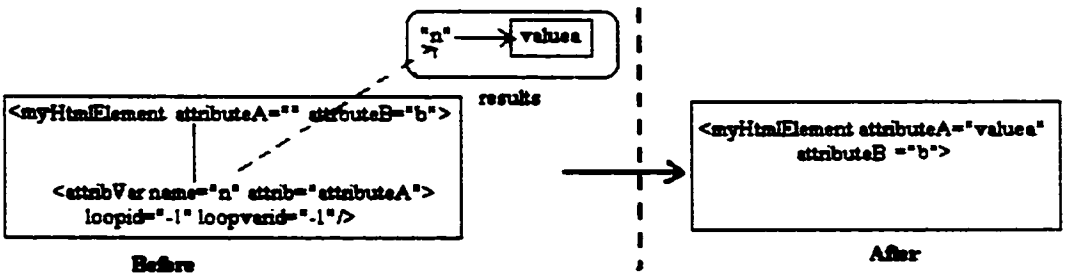

Fig. 5.14 attribVar as Child Node

- If the child node is an *if* node, the composer will retrieve the value corresponding to its *conditionVar* child node. If the value is evaluated to be true, its *then* child node is sent to be modified, otherwise its *else* child node is sent to be modified. The *if* node is then replaced by the modified node returned (Fig.5.15).

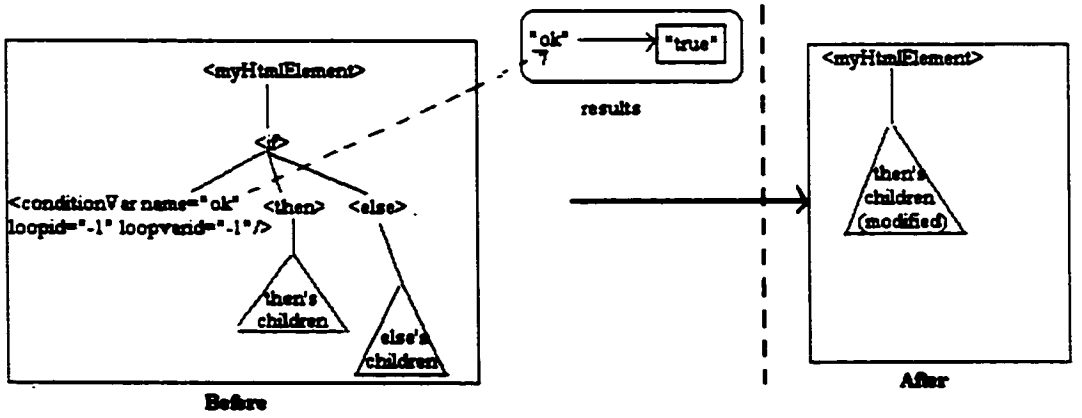

Fig. 5.15 if as Child Node

- If the child node is a *foreach* node, the composer first constructs a new document object. This object acts as a template to hold a new node that will be used to replace the *foreach* node. The composer determines the number of iterations that should take place from the value corresponding to *foreach* node's *controlVar* attribute (which is

an array) from *results*. Then, for each iteration, the composer clones the *foreach* node, modifies all the child nodes of the cloned node and inserts the modified child nodes to the new document. A *DocumentFragment* objecte is then constructed from the new document and is used to replace the *foreach* node (Fig.5.16).

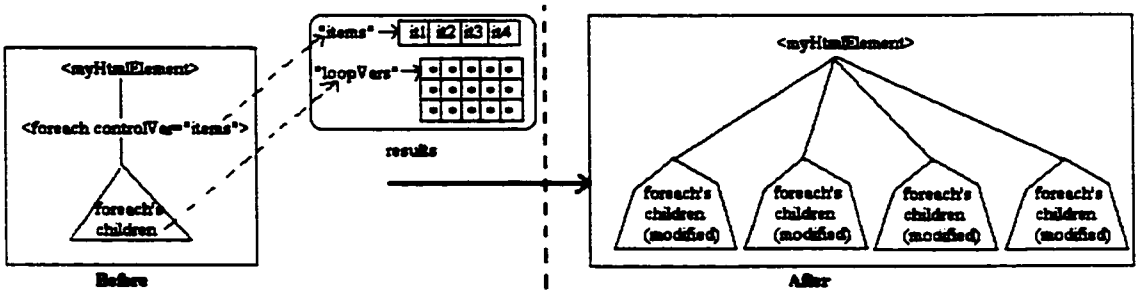

Fig. 5.16 foreach as Child Node

- If it is a *myHtmlElement*, it is replaced by a modified *myHtmlElement* (Fig.5.17).

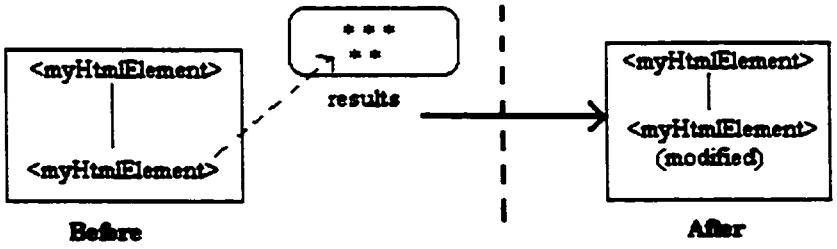

Fig. 5.17 myHtmlElement as Child Node

Fig. 5.18 is the class diagram of the composer. The *concreteController* uses constructor *Composer(Document doc, Hashtable results, PrintWriter out)* to create a *Composer* object. Where *doc* is a clone of a document representing an AMV, so that the original AMV templates do not change from request to request. *results* is the *Hashtable* object returned from *concretePageHandler*. *out* is a *PrintWriter* object obtained by calling

getWriter() method on a *HttpResponse* object. When the *compose* method is called, the composer calls method *handleMyHtmlTag(). handleMyHtmlTag which* in turn calls *handleForeach, handleIf, handleVar* or *handleMyHtmlTag* to recursively modify its child node. Finally the document tree contains only *Text* node and *W3ChtmlElement.* Recall that *Javax.xml.transform* package defines APIs that allow the transformation of XML into other forms. The composer uses this package to transform the modified DOM tree into a XML document and prints it to the Web user. The following is the code segment:

```
import Javax.xml.transform.TransformerFactory;
import Javax.xml.transform.dom.DOMSource;
import Javax.xml.transform.stream.StreamResult;
import Javax.xml.transform.Transformer;

. . .
try{
      TransformerFactory tFactory = TransformerFactory.newInstance();
      Transformer transformer = tFactory.newTransformer();
      DOMSource source = new  DOMSource (doc);
      StreamResult output = new StreamResult (out);
      transformer.transform(source,output);
 }catch(Exception e){
            e.printStackTrace();
   }
...
```

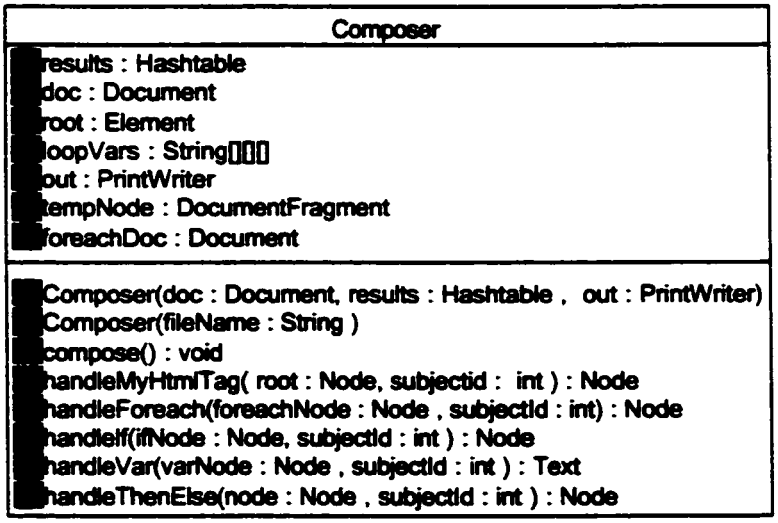

Fig. 5.18 Class diagram of Composer

CHAPTER 6 The Experiment

The Java 2 Platform, Enterprise Edition (J2EE)([Sund]) is a fully featured development framework introduced by Sun Microsystems. It is designed to simplify complex problems with development, deployment, and management of multi-tier enterprise applications. The mission of J2EE is to provide a platform-independent, portable, multi-user, secure, and standard enterprise-class platform for both client-side and server-side deployments written in Java.

The J2EE applications are typically composed of three tiers: the client tier, the middle tier and the enterprise information system (EIS) tier. Fig. 6.1 illustrates such an architecture. The client tier provides user interface and user interaction. The middle tier provides client services and business logic for an application. The middle tier may consist of one or more subtiers. Web tier supports client services through Web containers. EJB tier supports business logic component services through Enterprise JavaBeans (EJB) containers. The EIS tier in the backend provides access to existing information systems. EJB technology is a very good technology for implementing Concrete Model. However, the Web presentation technology JSP is not so wonderful due to its inability to separate the presentation and logic.

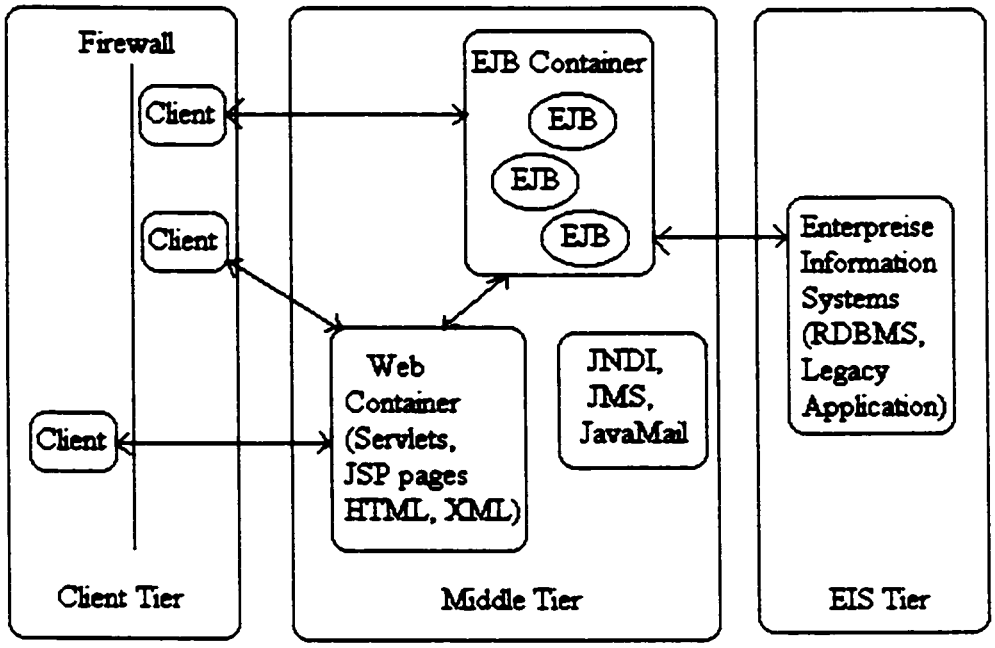

Fig. 6.1 J2EE Architecture ([Kas00])

6.1 Experiment Description

The J2EE Blueprints program provides the official "blueprints" for building enterprise applications ([Kas00][Sune]). It offers very useful practice guidelines and architectural recommendations for developers to build portable, scalable, and robust applications using the latest J2EE technologies. It is no doubt the best sample application that illustrates the design and development of Web-based application using J2EE technologies. J2EE Blueprints contains three applications. The major one is Java Pet Store (JPS). JPS is a sample e-commerce application, presenting users with various views of pets for sale; processing orders; managing user accounts and shopping sessions etc.

Since the goal of this thesis is to provide a framework for separating the presentation and logic of a Web-based application, instead of

implementing an actual application. I feel that it is sufficient to modify an existing non-trivial application that uses JSP as a major Web presentation generation technique, unplug its Web presentation part, rewrite the Web presentation using the template language introduced, and rebuild an equivalent application using the proposed framework. Because JPS is well designed and implemented, it maximizes the modularity and reusability of the components. It allows the replacement of part of the application without affecting the whole application. It uses JSP pages as the major component of Web-tier to handle user requests and dynamically generates Web pages. It is chosen as the test base of the experiment. The goals of the experiment are:

- To re-implement the Web presentation of JPS using the template language introduced in Chapter 4.
- To model the dynamic information of each Web template (implement *concretePageHandlers*).
- To implement a *concreteController* for JPS.
- To use the framework proposed in Chapter 5 to integrate the new Web presentation and the EJB-tier and EIS-tier of JPS.

6.2 The Results of the Experiment:

- AMV templates encapsulate all the presentation.

  First the sample pages are obtained by running the original JPS application, so that they can keep the exactly same Web presentation as the original JPS. Then each page is modified using a XML editor (XML spy 3.5 is used here). The page is made to comply with XML standards. For example, <b> is changed to <b/>. The dynamic part of the page is replaced by tags introduced in section 4.3. It proves that

HTML tags in addition to tags introduced are sufficient to represent any type of Web templates. The implementation of Web presentation (AMV template) is completely separated from the application logic.

- *ConcretePageHandler* encapsulates logic.

  All the templates are processed by the PHSG to generate skeletons for *concretePageHandler*. Since most of the logics have been taken care of in the skeleton. The implementation is done by simply filling in method bodies, and adding some flow controls in *handle* method. It proves that none of the presentation is generated in *concretePageHandler*. All the logics related to dynamic information are contained in *concretePageHandler*s.

- The framework can integrate the presentation and the logic.

  A *concreteController* is implemented, and the proposed framework is used to integrate the components. The resulting application provides exactly the same Web presentation, and functions equivalently.

The extent of presentation and logic separation using Servlet, JSP, and proposed method is given in table 6.1.

Traditional CGI, Servlet and CGI like	Existing Templating Framework	Proposed method
No	Some	100%

Table 6.1 Degree of Presentation and Logic separation

CHAPTER 7 Conclusions and Future work

This thesis provides a new approach for completely separating presentation and logic in a Web-based application. For the first time, it extends the traditional model-view-controller architecture and gives the definition of the Abstract Model and Abstract Model View. It provides a template language that allows the presentation of a Web-based application be developed independently, and provides a framework that integrates the dynamically generated information with the predefined Abstract Model View template for generating Web page that is sent to the client. Unlike the tag based templating frameworks such as JSP and XSP that use totally new Web presentation methodology, the template language provided here is based on widely accepted standards in HTML and XML. This allows Web developers to use existing techniques and tools to develop Web presentations. Also unlike the script based templating frameworks such as Webmaro and Velocity that embed special character identified scripts in HTML code, which is error prone. The current template language is tag based. Most importantly, the template language allows the page designer to develop and maintain Web page template totally independent of application logic.

This thesis is intended to serve as a proof of a concept. There is still a great deal of room for improvement. For example:

- A DTD can be developed and used to check the validation of an AMV template.

- JSP has an "include" directive that allows the content of the resource be included in the specified place of a JPS page. So a similar tag can be introduced to give this function.
- A tool that helps the development of AMV template can be developed. This tool can automatically fix the HTML file generated from HTML editor and make the HTML file comply with XML. The tool can check the consistency between the declaration tags and tags used in <html>.
- The composer uses recursion to combine the dynamically generated information and the AMV template. A non-recursion solution may be used to save space.
- The proposed framework is a general framework. It is more concerned with the identification of components and their interactions of Web based applications than the detailed Web application development. The turbine ([ASFc]) framework, a tool for building web applications, provides a collection of re-usable components that application programmers can use to rapidly build Web applications. These two frameworks are not contradictory, but they compensate each other. Slightly modifications can make these two frameworks work together, which could be an interesting future project.

APPENDIX

The following is the source code for the Java classes that have been developed for the framework. The Java classes used for testing the framework (rewriting JPS) are not included because of the length of code.

Class AbstractController.java

```
package shuling.thesis.webappframework;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.ServletException;
import javax.servlet.ServletContext;

import java.io.PrintWriter;
import java.util.Hashtable;
import org.w3c.dom.Document;

/* This class contain logics that handle interaction between
controller,abstract model, abstract model view and concrete model*/

public abstract class AbstractController extends HttpServlet{
    //map a pageid to a DOM tree (represent an AMV template) in memory
    protected Hashtable templates;
    //page to redirect for general errors
    protected String errorPage;
    //welcome page
    protected  String welcomePage;
    //page to redirect when log in is needed
    protected  String loginPage;
    //map a pageid to a boolean value that indicates whether the
requested page require log in
    protected  Hashtable requireLogins;
    //the context of the servlet
    protected ServletContext context;
    //the printWriter used for generating http response
    protected PrintWriter out;

    // This method initialize the controller
    public void init(){
        context = getServletContext();
        //do any application specific initialization here
        concreteInit();
        //load AMV templates in memory
        loadTemplates();
        setErrorPage();
        setWelcomePage();
```

```
        setLoginPage();
        setRequireLogins();
    }

    // Application specific initialization goes to here
    abstract protected void concreteInit();
    // This method Loads the AMV templates in memory
    abstract protected void loadTemplates();
    // This method sets the value of errorPage
    abstract protected void  setErrorPage();
    // This method sets the value of welcomePage
    abstract protected void  setWelcomePage();
    // This method sets the value of loginPage
    abstract protected void  setLoginPage();
    // This method sets the value of requireLogins
    abstract protected void  setRequireLogins();
    // This method extracts pageId from request
    abstract protected String  getPageId(HttpServletRequest request);
    // This method checks whether the customer has signed in
    abstract protected boolean isLoggedIn(HttpServletRequest request);
    // This method creates and initialize a concrete pageHandler for
the requested AMV and returns it
    abstract protected PageHandler getPageHandler(HttpServletRequest
request);
    // This method creates and initialize a concreteModelupdator for
updating Concrete Model
    abstract protected ModelUpdator getModelUpdator(HttpServletRequest
request);

    public void doGet(HttpServletRequest request, HttpServletResponse
response)throws ServletException{

        try{
           // Every Http request contains an id of the page a client
requests
           String nextPage =getPageId(request);
           // check whether the client need to sign in to see the
requested page
           boolean requirelogin =
Boolean.getBoolean((String)(requireLogins.get(nextPage)));
           // set header
           response.setContentType("text/html");
           //PrintWriter out = response.getWriter();
           out = response.getWriter();

           // redirect the client to the login page if he is required to
login but has not signed in
           if ( requirelogin && !isLoggedIn(request))
              redirect( loginPage, request, response);

           Exception excep = null;
           try{
               // get modelupdator for updating Concrete Model
               ModelUpdator mu = getModelUpdator(request);
               if ( mu != null)
                   mu.update();
           }catch( Exception e){
```

```
            excep = e;
            System.err.println("ModelUpdator "+e);
         }
         // get Abstract Model
         PageHandler ph = getPageHandler(request);
         // hashtable is used to hold all the dynamic generated
information (updated AMV)
         Hashtable result = new Hashtable();

         // handle method updates the Abstract Model
         if ( excep != null)
            results = ph.handle(excep);
         else
            results = ph.handle();

         String redirectPage = (String)results.get("redirect");
         out.flush();
         // redirect the client to page specified in "redirect"
         if (redirectPage != null)
             redirect(redirectPage, request, response);
         else{
            // make a copy of the AMV template
            Document origdoc = (Document)templates.get (nextPage);
            Document doc=null;
            doc =(Document)origdoc.cloneNode(true);
            Composer cpsr = new Composer(doc, results,out);
            // combine the AMV template and the Abstract Model to
generate the web page
            cpsr.compose();
          }
      }catch (Exception e){
         System.err.println("doGet"+e);
         redirect (errorPage, request, response);
      }
   }

   public void doPost(HttpServletRequest request, HttpServletResponse
response)throws ServletException{
        doGet(request, response);
   }
   // Redirect "page" to Web client
   protected abstract void redirect(String page,HttpServletRequest
request, HttpServletResponse response)throws ServletException;


}
```

Class Composer.java

```
package shuling.thesis.webappframework;

import org.w3c.dom.Element;
import org.w3c.dom.Document;
import org.w3c.dom.DocumentFragment;
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;
```

```
import org.w3c.dom.Text;
import org.w3c.dom.CharacterData;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerConfigurationException;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import javax.servlet.http.HttpServletResponse;

import java.util.Hashtable;
import java.io.PrintWriter;
import java.io.File;

/* This class is used to combine the dynamically generated information
of a web page (the abstract model) and the AMV template and generate
the final page that is sent to client */
public class Composer{
    // dynamically generated information
    private Hashtable results;
    // cloned AMV template
    private static Document doc;
    private Element root;
    String[][][] loopVars;
    PrintWriter out;
    DocumentFragment tempNode;
    Document foreachDoc;

   public Composer(Document doc, Hashtable results,PrintWriter out){
        this.doc = doc;
        this.results = results;
        this.out=out;
        root = doc.getDocumentElement();

        loopVars = (String[][][])results.get("loopVars");
    }
    public Composer(String fileName){

        try {
           DocumentBuilderFactory docBuilderFactory =
DocumentBuilderFactory.newInstance();
           DocumentBuilder parser =
docBuilderFactory.newDocumentBuilder();
           Document origdoc = parser.parse(new File(fileName));
           doc =(Document)origdoc.cloneNode(true);
           root = doc.getDocumentElement();
           Node declNode=
root.getElementsByTagName("declaration").item(0);
           root.removeChild(declNode);
           root = (Element)root.getElementsByTagName("html").item(0);
        }catch (Exception err){
            System.err.println(err.toString());
        }

        results = new Hashtable();
```

```
  }

  public void compose(){

       Node htmlNode = doc.getElementsByTagName("html").item(0);
       handleMyHtmlTag( htmlNode,-1);
       Node newhtmlNode = doc.importNode(htmlNode,true);
       Node templateNode =doc.getElementsByTagName("template").item(0);
       doc.removeChild(templateNode);
       // make the updated HTML node the root node
       doc.appendChild (newhtmlNode);

       try{
          TransformerFactory tFactory =
TransformerFactory.newInstance();
          Transformer transformer = tFactory.newTransformer();
          DOMSource source = new  DOMSource (doc);
          StreamResult output = new StreamResult (out);
          /*transform the DOM tree to a XML document and send it to the
client*/
          transformer.transform(source,output);
       }catch(Exception e){
            e.printStackTrace();
       }
    }
    /* This method recursively update each child node of the HTML node
until the children are all W3C allowed elements*/
    private Node handleMyHtmlTag(Node root, int subjectid){
        Node current = root.getFirstChild();

        while ( current != null){
           Node next = current.getNextSibling();
           String currentName= current.getNodeName();

           if (current instanceof Element){
              // if the child node is a <var>
              if ( currentName.equals("var")){
                  Node newNode = handleVar(current,subjectid);
                  //replace <var> with its value
                  root.replaceChild(newNode,current);
                  current = next;
                  continue;
              }
              // if the child node is <attribVar>
              if ( currentName.equals("attribVar")){
                  // get its value
                  Node attribValue = handleVar(current, subjectid);
                  // get the attribute name in the parent tag
                  String attribName =
((Element)current).getAttribute("attrib");
                  // set the value of the attribute in the parent tag
                  ((Element)root).setAttribute(attribName,
((Text)attribValue).getData());
                  // remove <attribVar> tag
                  Node toRemove = current;
                  current = next;
                  root.removeChild(toRemove);
```

```
                continue;
            }
            // if the child node is <if>
            if ( currentName.equals("if")){
                Node newNode = handleIf(current, subjectid);
                root.replaceChild (newNode, current);
                current =next;
                continue;
            }
            // if the child node is <foreach>
            if ( currentName.equals("foreach")){
                  Node newNode=handleForeach(current,subjectid);
                root.replaceChild(newNode,current);
                current =next;
                continue;

            }else{ // if the child node is html tags that may contain
introduced tags and other html elements
                handleMyHtmlTag(current, subjectid);
                current =next;
                continue;
            }
          }else
                current = next;
        }
        return root;
    }
    private Node handleForeach(Node foreachNode,int subjectId){

       DocumentBuilderFactory factory=
DocumentBuilderFactory.newInstance();

       try{
          // create a new document as the temporary holder
          DocumentBuilder builder = factory.newDocumentBuilder();
          foreachDoc = builder.newDocument();
          Element foreachroot = foreachDoc.createElement("foreachDoc");
          foreachDoc.appendChild(foreachroot);

          if( foreachNode instanceof Element){
             Element foreachCondElement =(Element)foreachNode;
             String controlVarName =
foreachCondElement.getAttribute("controlVar");
             // get the array whose elements shall be gone through
             Object[] subjects = (Object[])results.get(controlVarName);
             int length = subjects.length;
             // a DocumentFragment object to hold the updated child
node
             DocumentFragment fragofSubject;

             for(int i=0; i<length; i++){
                fragofSubject = doc.createDocumentFragment();
                // clone the foreachnode
                Node clonedNode = foreachNode.cloneNode(true);

                /* operate on the cloned node so that the original node
does not change*/
```

```
                Node current = clonedNode.getFirstChild();
                while ( current != null){
                    Node next = current.getNextSibling();
                    String currentName= current.getNodeName();
                    if (current instanceof Element){
                        if ( currentName.equals("var")){
                        Node newNode = handleVar(current,i);
                        fragofSubject.appendChild(newNode);
                        current = next;
                        continue;
                        }
                        if ( currentName.equals("if")){
                            Node newNode = handleIf(current, i);
                            fragofSubject.appendChild(newNode);
                            current =next;
                            continue;
                        }
                        if ( currentName.equals("foreach")){
                            Node newNode= handleForeach(current,i);
                            fragofSubject.appendChild(newNode);
                            current =next;
                            continue;
                        }else{
                            Node newNode =handleMyHtmlTag(current, i);
                            fragofSubject.appendChild(newNode);
                            current =next;
                            continue;
                        }
                    }else{
                        fragofSubject.appendChild(current);
                        current = next;
                    }
                }

                fragofSubject.normalize();
                Node toappend=
foreachDoc.importNode(fragofSubject,true);
                // append the document fragment to the new document
                foreachroot.appendChild(toappend);

            }
          }
        }catch(Exception e){
            System.err.println(e);
        }
        /* in the original document, create a document fragment and
copy all the child nodes of the new document to it*/
        tempNode = doc.createDocumentFragment();
        NodeList list =
foreachDoc.getElementsByTagName("foreachDoc").item(0).getChildNodes();
        for (int i=0; i<list.getLength(); i++){
            Node anode =doc.importNode( list.item(i),true);
            tempNode.appendChild(anode);
        }
        tempNode.normalize();
        return tempNode;
```

```
    }

    /* return the updated child nodes of <then> if the <conditionVar>
evaluated to true
       return the updated child nodes of <else> if the <conditionVar>
evaluated to false*/
    private Node handleIf(Node ifNode, int subjectId){

        if(ifNode instanceof Element){
            Element ifElement = (Element)ifNode;

            Node varNode =
ifElement.getElementsByTagName("conditionVar").item(0);
            String value = ((CharacterData)handleVar(varNode,
subjectId)).getData();
            boolean istrue = value.equals("true");
            Node
thenNode=ifElement.getElementsByTagName("then").item(0);

            if (istrue)
               return handleThenElse(thenNode, subjectId);
            else{
               Node elseNode =
thenNode.getNextSibling().getNextSibling();
               return handleThenElse(elseNode, subjectId);
            }

        }
        return null; //throw exception
    }
    // return the value of the <var>
    private Text handleVar(Node varNode, int subjectId){
        Text textNode= doc.createTextNode("");

        int loopvarid =-1;
        if( varNode instanceof Element){
           Element varElement=(Element)varNode;
           int loopid =
Integer.parseInt(varElement.getAttribute("loopid"));
           String value;

           if (loopid != -1){      // var inside loop
              loopvarid =
Integer.parseInt(varElement.getAttribute("loopvarid"));
              value = loopVars[loopid][loopvarid][subjectId];
           }
           else{    //var outside any loop
              String name= varElement.getAttribute("name");
              value = (String)results.get(name);
           }
           textNode = doc.createTextNode(value);
        }
        return textNode;
    }
  // return the updated child nodes of <then> or <else>//
  private Node handleThenElse(Node node, int subjectId){
```

```
        DocumentFragment frag = doc.createDocumentFragment();
        handleMyHtmlTag(node, subjectId);
        NodeList child = node.getChildNodes();
        for (int i=0; i< child.getLength(); i++){
            Node cld= child.item(i);
            frag.appendChild(cld.cloneNode(true));//Note: must clone,
otherwise the node is moved!
        }
        return frag;
    }


    public static void main(String[] args){
        String fileName= args[0];
        Composer cp= new Composer(fileName);
        cp.compose();

    }


}
```

Class ExceptionMap.java

```
package shuling.thesis.webappframework;

import java.util.Hashtable;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.ServletContext;
import java.io.PrintWriter;

/* this class map an exception to the id of a page that the client
should be redirected to */
public abstract class ExceptionMap implements java.io.Serializable{
    private Hashtable map;

    public ExceptionMap(){
        map = getMap();
    }
    abstract protected Hashtable getMap();
    public String mapToPage(Exception e){
        String classname = e.getClass().getName();
        return (String)(map.get(classname));
    }

}
```

Class ConcreteModel.java

```
package shuling.thesis.webappframework;
```

```
/* interface that is used to identify the semantics of being Concrete
Model*/
public interface ConcreteModel{
}
```

Class ModelUpdator.java

```
 package shuling.thesis.webappframework;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.ServletContext;

/* this class is used to map the Http request to actions that should be
performed by proper concrete models*/
public abstract class ModelUpdator implements java.io.Serializable{

    protected HttpServletRequest request;
    protected ServletContext context;

    public ModelUpdator(){

    }
    public abstract void update() throws Exception;

    public void init(HttpServletRequest request, ServletContext
context){
        this.request =request;
        this.context = context;
        concreteInit();
   }
   protected abstract void concreteInit();

}
```

Class PageHandler.java

```
package shuling.thesis.webappframework;

import java.util.Hashtable;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.ServletContext;
import java.io.PrintWriter;

/* This is the base class of all pagehandlers, all the subclass must
provide concrete "handle" method to set the values for all the
dynamically. This class corresponds to the abstract command in the
command pattern */
public abstract class PageHandler implements java.io.Serializable{
    protected Hashtable results;
    protected HttpServletRequest request;
    protected ServletContext context;
    protected PrintWriter out;
```

```
    public PageHandler(){
        results = new Hashtable();
    }
    /* if the concrete model has been updated successfully */
    public abstract Hashtable handle() throws Exception;
    /* if the concrete model has not been updated successfully */
    public  abstract Hashtable handle(Exception e);

    public void init(HttpServletRequest request, ServletContext
context,PrintWriter out){
        this.request =request;
        this.context = context;
        this.out =out;
        concreteInit();
   }
   protected abstract void concreteInit();

}
```

Class PageHandlerSkeletonGenerator.java

```
package shuling.thesis.webappframework;

import java.io.BufferedReader;
import java.io.PrintWriter;
import java.io.FileReader;
import java.io.FileOutputStream;
import java.io.FileInputStream;
import java.io.File;
import org.xml.sax.InputSource;
import org.w3c.dom.Element;
import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;

import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import org.xml.sax.SAXException;

import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import java.io.BufferedReader;
import java.io.FileInputStream;

/* This class takes template file names from the command line and
generate corresponding page handler java skeleton files. The developer
can implement page handler class base on the file generated.The
assumption that the template files follow xml syntax and rules for AMV
template */

public class PageHandlerSkeletonGenerator{
    //prefix of the file name to genetate, it's the same as the prefix
of the template file
    private static String prefix = "";
    //printWriter to write to the file generated
```

```
    private static PrintWriter out;
    //String for method "handle" in generated java file
    private static String handleMethodString;
    private static String concreteInitString;
    //String for variable declaration in generated java file
    private static String varDeclString;
    private static int numLoops;       // number of loops in the
template file declaration
    private static String loopVarNumString="";
    private static String loopObjNameString="";

    //Suppose all the AMV template files are in the same directory of
this file
    public static void main(String[] args){

        for (int i=0; i<args.length; i++){

            handleMethodString = "   public Hashtable handle(){\n";
            varDeclString ="";
            concreteInitString ="   public void concreteInit(){\n";
            numLoops =0;

            int endIndex = args[i].lastIndexOf(".");
            prefix = args[i].substring(0,endIndex);
            System.out.println("prefix"+prefix);

            try{
               String skeletonfileName = prefix +".java";
               System.out.println("SKELE"+skeletonfileName);
               // create the java file with the same prefix of the AMV
template file
               out = new PrintWriter(new FileOutputStream(new
File(skeletonfileName)));
            }catch(Exception e){
               System.err.println(e.getMessage());
            }

            printFirstPart();
            Element root = loadDocument(new File(args[i]));
            root.normalize();
            // declaration node of AMV template has all the information
needed
            Node declarationNode =
root.getElementsByTagName("declaration").item(0);
            Element declElement = (Element)declarationNode;
            handleAttribute(declElement);
            handleParameter(declElement);
            handleVarDeclaration(declElement);
            handleLoopDeclaration(declElement);

            writeVarDeclaration();
            out.println(concreteInitString);
            writeHandleMethod();
        }
    }
    //load and parse the AMV template file
    public static Element loadDocument(File templateFile) {
```

```
        Document doc = null;
        try {
            DocumentBuilderFactory docBuilderFactory =
DocumentBuilderFactory.newInstance();
            DocumentBuilder parser =
docBuilderFactory.newDocumentBuilder();
            doc = parser.parse(templateFile);
            Element root = doc.getDocumentElement();
            root.normalize();
            return root;
        } catch (Exception err) {
            System.err.println(err.toString());
        }
        return null;
    }
    /* This method handles attributes, which are information that
should not be known by the client, but needed for updating Abastract
Model */
    private static void handleAttribute(Element declElement){
        NodeList attribList
=declElement.getElementsByTagName("attrib");
        if(attribList!= null && attribList.getLength() >0){
           Element attribElement;
           String attribName;
           String description;
           String attribDeclString;
           String attribValue;
           for(int i=0; i<attribList.getLength(); i++){
              attribDeclString ="";
              attribElement =(Element)(attribList.item(i));
              attribName = attribElement.getAttribute("name");
              description = attribElement.getAttribute("description");
              attribValue = attribElement.getAttribute("value");

              if (description.length()>0)
                 attribDeclString += "    //"+description +"\n";
              // write a "String attribName = attribValue" entry in the
generated file
              attribDeclString += "    String "+attribName+" = \""+
attribValue + "\";\n";
              varDeclString += attribDeclString;
           }
         }

    }
    /* This method extract names of parameters that should be found in
the request*/
     private static void handleParameter(Element declElement){
        NodeList paramList = declElement.getElementsByTagName("param");
        if(paramList!= null && paramList.getLength() >0){
           Element paramElement;
           String paramName;
           String description;
           String paramDeclString;
           for(int i=0; i<paramList.getLength(); i++){
              paramDeclString ="";
              paramElement =(Element)(paramList.item(i));
```

```
                paramName = paramElement.getAttribute("name");
                description = paramElement.getAttribute("description");
                paramDeclString += "\n    //Param to be extracted from
request:\n";
                if (description.length()>0)
                   paramDeclString += "    //"+description +"\n";
                paramDeclString += "    String "+paramName+";\n";
                varDeclString += paramDeclString;
                concreteInitString += "       "+paramName+" =
request.getParameter(\""+paramName+"\");\n";

            }
          }
          concreteInitString += "       ...\n    }\n";

      }

     /*for each variable,generate a method skeleton which shall be
implemented later to set the value of the variable */
     public static void handleVarDeclaration(Element declElement){
         NodeList varList = declElement.getElementsByTagName ("var");

         if(varList != null && varList.getLength()>0){
            for( int i=0; i< varList.getLength(); i++){
               Element varDeclaElement = (Element)(varList.item(i));
               String name = varDeclaElement.getAttribute("name");
               String description =
varDeclaElement.getAttribute("description");
               String loopidString =
varDeclaElement.getAttribute("loopid");
               String loopvaridString =
varDeclaElement.getAttribute("loopvarid");
               int loopid =
Integer.parseInt(varDeclaElement.getAttribute("loopid"));
               String comment ="    /* " + description +"
Loopid="+loopidString+" loopvarid="+loopvaridString+" */";
               if (loopid == -1)
                  varsOutsideLoop(name, comment);
               else{
                  int loopvarid =
Integer.parseInt(varDeclaElement.getAttribute("loopvarid"));
                  varsInsideLoop(name, comment, loopid, loopvarid);
               }
            }
         }
     }
     /*For variables outside loop, generate a "set_VarName" method
skeleton */
     public static void varsOutsideLoop(String name, String comment){

        out.println(comment);
        out.println("    private void set_"+name +"(){");
        out.println("       String key = \""+name +"\";");
        out.println("          ...// get value ");
        out.println("          ...//results.put( key,value);");
        out.println("    }\n");
        out.flush();
```

```
        handleMethodString +="      set_" + name +"();\n";
    }

    /* For variables inside loop, generate a "get_loopXVarY" method */
    public static void varsInsideLoop(String name, String comment,int
loopid, int loopvarid){
        out.println(comment);
        out.println("   public String get_Loop"+loopid
+"Var"+loopvarid+"(Integer index){");
        out.println("      String value=...");
        out.println("      return value;");
        out.println("   }\n");
        out.flush();
    }

    /**
     * This method generates some variable declartions in the resulting
java file
     * loopNum is the number of loops in the template file
     * loopVarNum[i] is the number of variables for loop i
     * loopSubjName[i] is name of the variable that controls loop i
     * assume that the loopid start from 0, the declaration of the loop
is placed by the order
     * of the loopid
     */
    public static void handleLoopDeclaration(Element declElement) {

        NodeList loopList = null;
        loopList=declElement.getElementsByTagName("loop");



        if(loopList != null && loopList.getLength()>0){

           loopVarNumString = "   int[] loopVarNum ={";
           loopObjNameString="   //This array stores the control object
of each loop\n";

           loopObjNameString +="   String[] loopControlVarName={";
           for(int i=0; i< loopList.getLength(); i++){
              numLoops ++;
              String name = "";
              String subject="";          //var that controls the loop
              int varNum;              //How many variables are declared
inside the loop

              Element element = (Element)(loopList.item(i));
              subject = element.getAttribute("subject");
              varNum = Integer.parseInt(
element.getAttribute("varNum"));
              loopVarNumString += varNum +",";
              loopObjNameString += "\""+subject+"\",";
           }
        }

    }
```

```
    // generate the beginning part of the java file

    public static void printFirstPart(){
        out.println("package
shuling.thesis.myPetstore.pageHandlers;\n");
        out.println("import
shuling.thesis.webappframework.PageHandler;");
        out.println("import java.lang.reflect.Method;");
        out.println("import java.util.Hashtable;\n");

        out.println("public class "+prefix+" extends PageHandler{");
        out.flush();
    }

    // generate the variable declaration of the java file

    public static void writeVarDeclaration(){
        if (numLoops >0)
           varDeclString += "\n   int numLoops =" + numLoops +";\n";

        if ( loopVarNumString.length()>0)
            loopVarNumString=
loopVarNumString.substring(0,loopVarNumString.length()-1)+ "};\n";
        if ( loopObjNameString.length()>0)
            loopObjNameString =
loopObjNameString.substring(0,loopObjNameString.length()-1)+"};\n";
        varDeclString += loopVarNumString;
        varDeclString += loopObjNameString;
        out.println(varDeclString);
        out.flush();
    }

    // generate handle method skeleton of the java file

    public static void writeHandleMethod(){
       out.println(handleMethodString);
       if ( numLoops >0){
          out.println("      String[][][] loopVars = new
String[numLoops][][];");
          out.println("      for (int i=0; i<loopVars.length; i++)");
          out.println("         loopVars[i] = new
String[loopVarNum[i]][];");

          out.println("      for(int i=0; i<loopVars.length; i++){");
          out.println("         Object[] subjects
=((Object[])results.get(loopControlVarName[i]));");
          out.println("         int loopObjLength;");
          out.println("         if ( subjects != null)");
          out.println("            loopObjLength = subjects.length;");
          out.println("         else");
          out.println("            loopObjLength = 0;\n");
          out.println("         for (int j=0; j<loopVars[i].length;
j++)");
          out.println("            loopVars[i][j]= new
String[loopObjLength];\n ");
          out.println("      }\n");
```

```
          out.println("      for(int i=0; i<loopVars.length; i++)");
          out.println("         for (int j=0;
j<loopVars[i].length;j++)");
          out.println("             for (int k=0;
k<loopVars[i][j].length; k++){");
          out.println("                Class[] formalArg = { new
Integer(k).getClass()};");
          out.println("                String methodName
=\"get_Loop\"+i+\"Var\"+j;");
          out.println("                Method method =null;");
          out.println("                try{");
          out.println("                   method =
this.getClass().getMethod(methodName, formalArg);");
          out.println("                   Object[] actualArg ={new
Integer(k)};");
          out.println("                   if (method != null)");
          out.println("
loopVars[i][j][k]=(String)(method.invoke(this, actualArg));");
          out.println("                }catch(Exception e){");
          out.println("                    System.err.println( e);");
          out.println("                }");
          out.println("             }");
          out.println("      results.put(\"loopVars\",loopVars);");

       }

       out.println("   // set redirect...");
       out.println("   ...");
       out.println("   return results;");
       out.println("   }\n");
       out.println("   public Hashtable handle(Exception e){");
       out.println("          ...");
       out.println("          return results;");
       out.println("   }");
       out.println("}\n");
       out.flush();

    }
}
```

REFERENCES

[ASFa] Apache Cocoon Homepage. Apache Software Foundation. Online: http://xml.apache.org/cocoon

[ASFb] Apache Velocity Homepage. Apache Software Foundation. Online: http://jakarta.apache.org/velocity/

[ASFc] Apache Turbine Homepage. Apache Software Foundation. Online: http://jakarta.apache.org/turbine/

[AAB00] Avedal, Karl; Ayers, Danny; Briggs, Timothy etc. Profession JSP. Wrox Press Inc. 2000

[Dew98] Dawna Travis Dewire. Thin Clients. McGraw-Hill publishing 1998

[EG99] Eddelbüttel, Dirk; Goffe, William L. Display and Interactive Languages for the Internet: HTML, PDF, and Java. Computational Economics. Volume: 14, Issue: 1/2, October 1999, pp. 89-107

[FSF] Freemarker Homepage. Free Software Foundation, Inc. Online: http://freemarker.sourceforge.net/

[GoF94] Erich Gamma, Richard Helm,Ralph Johnson, and John Vlissides; Design Patterns: Elements of Reusable Object-Oriented Software; Addison Wesley; October 1994

[Hal00] Marty Hall. Core Servlets and JavaServer Pages. Prentice Hall PTR, Upper Saddle River, NJ 07458, 2000

[Kas00] Nicholas Kassem etc. Designing Enterprise Applications with the Java 2 Platfrom, Enterprise Edition. Addison-Wesley Publishing. 2000

[KK00] zusammen mit S. Kuhlins: "Java Servlets vs. CGI - Implications for Remote Data Analysis" Studies in Classification, Data Analysis, and Knowledge Organization Vol. 16, Springer-verlag, Heigelberg, 2000, pp. 227-236

[KP88] Krasner, Glenn; Pope, Stephen "A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 system." Journal of Object Oriented Programming, 1988. vol. 1, no. 3, pp. 26-49

[LCG92] T. Berners-Lee et al, "World Wide Web: The Information Universe", Electronic Networking: Research, Applications and Policy, 1(2), 1992

[Mar99] Benoit Marchal. XML By Example. QUE 1999

[NCSA] Common Gateway Interface Home Page. National Center for Supercomputing Applications. Online: http://hoohoo.ncsa.uiuc.edu/cgi/intro.html

[PC00] Perrone, Paul; Chaganti, Venkata. Building Java Enterprise Systems with J2EE. SAMS, Macmillan USA 2000

[Sem] WebMacro Homepage. Semiotek Inc. Online: http://www.webmacro.org/

[Suna] Java Servlet Homepage. Sun Microsystems, Inc. Online: http://java.sun.com/products/servlet

[Sunb] Java Server Pages Homepage. Sun Microsystems, Inc. Online: http://java.sun.com/products/jsp/

[Sunc] Java Technology & XML. Sun Microsystems, Inc. Online: http://java.sun.com/xml/index.html

[Sund] Java 2 Enterprise Edition Home Page. Sun Microsystems, Inc. Online: http://java.sun.com/j2ee/

[Sune] J2EE blueprints Homepage. Sun Microsystems, Inc. Online: http://java.sun.com/blueprints/

[TGH96] Tittel, Ed; Gaither, Mark; Hassinger, Sebastain. CGI Bible. Foster City, CA : IDG Books Worldwide, 1996

[W3Ca] HyperText Markup Language Home Page. World Wide Web Consortium. Online http://www.w3.org/MarkUp/

[W3Cb] Extensible Markup Language (XML) World Wide Web Consortium http://www.w3.org/XML/

VITA AUCTORIS

Shuling Nie, female, was born on November 1, 1973 in Sichuan China. She got her associative degree in Accounting from the Southwest Finance and Economics University in Chengdu China in 1995. She got her Bachelor degree in Computer Science from the University of Windsor in 1999. She is expecting to get her Masters degree in Computer Science from University of Windsor in January 2002.