The Evaluation of Three Approaches to Implementing an OGC Web Map Service Client Application

---

A thesis
submitted in partial fulfilment
of the requirements for the Degree
of
Master of Science
at the
University of Canterbury
by
Hao Ding

University of Canterbury
2003

Examining Committee

Supervisor	Dr Richard Pascoe
	Department of Computer Science
	The University of Canterbury
Associate Supervisor	Dr Neville Churcher
	Department of Computer Science
	The University of Canterbury
External Examiner	Dr XinFeng Ye
	Department of Computer Science
	The University of Auckland

To my parents and my girlfriend, Xiaohong,
for their ongoing support and love

Abstract

A multitude of technologies are presently available for the development of Web applications, each having its strengths and weaknesses. Three of them that are used on the Java 2 platform, Enterprise Edition (J2EE) are introduced in this thesis. They are JavaServer Pages (JSP), JavaServer Pages Standard Tag Library (JSTL), and eXtensible Markup Language Compiler (XMLC).

The functionality that is representative of a generic OpenGIS Consortium (OGC) Web Map Service (WMS) client has been implemented using three approaches that are based on the above technologies—JSP with embedded Java, JSP with JSTL tags, and XMLC. The functionality includes producing a custom map with layers retrieved from different WMS servers; manipulating views of the map; querying information about features of a location selected on the map by the user, and so on.

In this thesis we evaluate and compare the three approaches from the perspective of application architecture, development, and maintenance, based on our implementation experience. We also present the design and setting up of a local Web mapping system on which the WMS client being implemented has been running.

Acknowledgments

I would like to thank my supervisors Dr. Richard Pascoe and Dr. Neville Churcher for their patience, guidance, and support. Also thank Michael Herman for proof reading the draft of this thesis.

I also would like to acknowledge the support of Dr. Greg Ewing, who set up a WMS-compliant map server that was interacted with in the experiment.

Finally, thank all my colleagues and friends who have given me help, support, and inspiration.

Contents

List of Figures

Chapter 1

Introduction

With the rise of the Internet, the World Wide Web[1] (WWW) has become an important medium for information distribution because of its accessibility and popularity. In the area of Geographic Information System (GIS), Web technologies are widely used in distributed applications, such as Web mapping systems, for geographic data gathering, accessing, and processing. Mapping on the Web includes the visual presentation of geospatial data, as well as more interactive operations, such as creating thematic maps, and querying and analyzing spatial information.

1.1 OGC Web Map Service

The Open GIS Consortium (OGC) Web Map Service (WMS) is part of the Web Mapping Testbed (WMT) project. It intends to advance interoperable Web mapping technology. The OGC WMS specification 1.1.0[2] [25] offers a standard client-server interaction protocol that each map server implements as a common interface for accepting requests and returning responses. The effect of the standard is to ensure that a client is able to access all the available OGC Web map servers over the Internet.

In the absence of the OGC WMS specification, a client that successfully interacts with one map server would in most cases be unable to interact with another as map servers from different venders would probably be implementing different functions.

The interoperable Web mapping structure is illustrated in Figure 1.1, where each

[1] Through out the thesis we will abbreviate the term WWW to the Web

[2] WMS 1.1.0 is the specification we referenced in our research. The latest version is WMS 1.1.1

map server is accessed by the client through the common interface and connects to a unique database containing geo-feature data for a specified spatial domain. In a distributed OGC WMS, a WMS server could also be a cascading map server [25] that is able to aggregate the capabilities of other WMS servers by interacting with them. The WMS client and WMS servers communicate with each other using the Hypertext Transfer Protocol (HTTP).

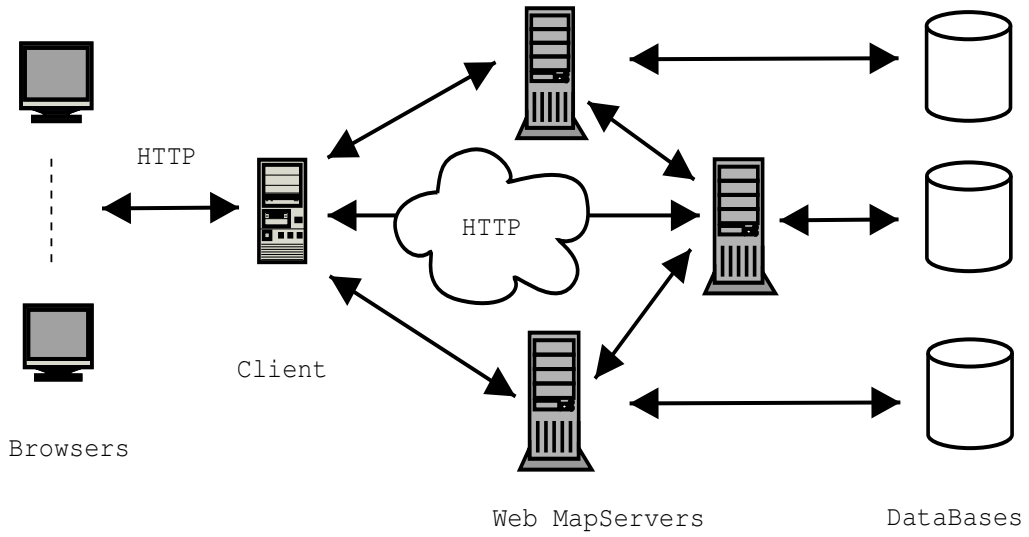

Figure 1.1: Interoperable Web mapping structure

1.1.1 OGC WMS Server

An OGC WMS server implements three standard operations: GetCapabilities, GetMap, and GetFeatureInfo. The first two operations are mandatory for each WMS server:

GetCapabilities: The GetCapabilities operation provides the client with a WMS server's service metadata. This metadata is provided as an eXtensible Markup Language (XML) [40] document that specifies the WMS server's capabilities, including geographic areas covered, supported image formats, coordinate reference systems, available map layers, and so on (see Appendix A for a sample capabilities XML document).

GetMap: The GetMap operation enables the client to request a map. The map is generally rendered in a pictorial format such as Graphics Interchange Format

(GIF) [6] and Portable Network Graphics (PNG) [37], which can be viewed directly in a graphical Web browser or other pictorial software.

GetFeatureInfo: The GetFeatureInfo operation is optional for a WMS server. Using this operation a client requests more information about features at a specific location in the map. The WMS server will respond with either a Geography Markup Language (GML) [24] document, or a plain text file, or a Hypertext Markup Language (HTML) file containing this information. GML is a type of XML encoding for specifying the geographic feature information (see Appendix B for a sample GML document).

1.1.2 OGC WMS Client

An OGC WMS client is a Web application that communicates with OGC WMS servers using the three standard operations. It also dynamically generates HTML pages for displaying WMS server capabilities, maps and features in the Web browser.

In a typical OGC WMS client-server interaction (see Figure 1.2), the client first requests a capabilities document from the WMS server in order to determine what functions the map server implements and what maps can be provided. The client then uses the GetMap operation to get a map. Finally, the client may use the GetFeatureInfo operation with a specific point on the map to retrieve more information.

The WMS client asks a WMS server to perform these operations by submitting HTTP requests in the form of Universal Resource Locators (URLs). All URLs include: the protocol; hostname; path; question mark '?'; specification version number; and service type. An ampersand '&' is inserted between each parameter name/value pair. URLs containing these parameters look like this:

```
http://www.servername/cgi-bin/mapserv?VERSION=1.1.0&SERVICE=WMS&
```

Additional request parameters are appended depending on what operation is requested. When requesting the GetCapabilities operation, a request type parameter as shown below should be appended to above string:

```
REQUEST=GetCapabilities
```

The GetMap request usually includes several parameters that are needed to produce a map by the WMS server. These include: layers; styles; bounding box; Spatial

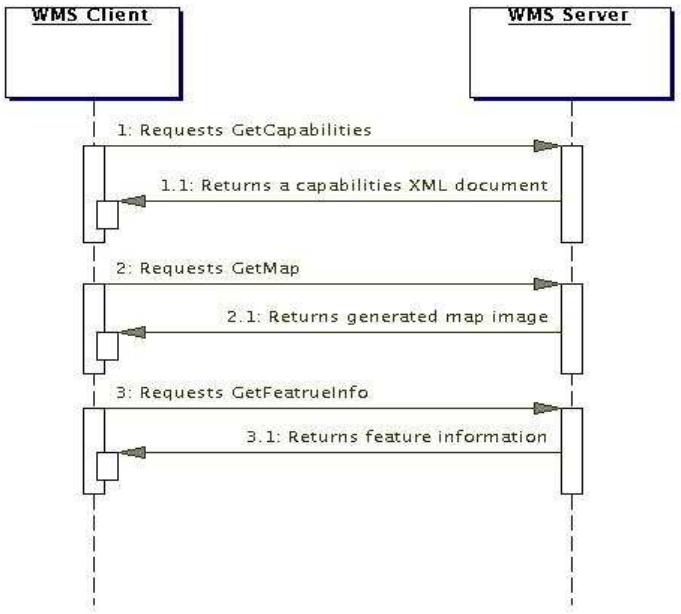

Figure 1.2: A sequence diagram depicting a typical interaction between a WMS client and server

Reference System (SRS); map output width and height; image format; and so on. This information must be aligned with the information provided in the capabilities XML document. The following are example parameters that may be appended to construct a GetMap request:

```
REQUEST=GetMap&BBOX=171.8889,-43.8908,173.1171,-43.2767&
SRS=EPSG:4326&WIDTH=600&HEIGHT=300&FORMAT=image/png&
LAYERS=airport_poly,road_cl&STYLES=default,default&TRANSPARENT=TRUE
```

Layers specify the information to be shown on the map (e.g. roads, rivers, towns). A list of styles, one or more for each layer, defines how to depict layers. For example, lines represent roads, circles represent towns. If the style of a layer is not claimed in the WMS server's capabilities document, that style is known as the "default" style [25]. The bounding box is a set of four coordinate values (minX, minY, maxX, maxY) indicating a rectangular area on the earth to be mapped using a specified SRS. The SRS names a projected reference system code, which includes a namespace prefix, a

colon and the identifier. Two namespaces are defined in the WMS specification: EPSG and AUTO (see [25] for detailed description)

Most of the GetMap request parameters that generate the original map are repeated in a GetFeatureInfo request. Additional parameters of the GetFeatureInfo request define the format of the file to be returned, the layer to be queried, the coordinates indicated on the map, and the maximum number of features about which to return information. The following illustrates the parameters that may be appended in a GetFeatureInfo request URL:

```
REQUEST=GetFeatureInfo&BBOX=171.8889,-43.8908,173.1171,-43.2767&
SRS=EPSG:4326&WIDTH=600&HEIGHT=300&FORMAT=image/png&
LAYERS=airport_poly,road_cl&STYLES=default,default&TRANSPARENT=TRUE&
QUERY_LAYERS=airport_poly,road_cl&X=280&Y=130&FEATURE_COUNT=5&
INFO_FORMAT=application/vnd.ogc.gml
```

Generic Functionality

Lots of institutions have already developed their OGC WMS-compliant client applications, such as CubeView of CubeWerx Inc [7], WMS Viewer of InterGraph [16] and Digital Earth Web Map Viewer of NASA [10]. Although each WMS client has its own features, some common functionality, such as zooming and panning image maps, adjusting display order of map layers, and producing custom maps with layers retrieved from different WMS servers, is implemented by all of them. We identify this functionality as being representative of a generic WMS client implementation for this research, and implement them using different approaches within the Java 2 Platform, Enterprise Edition (J2EE) framework [36].

1.2 Three Approaches to Implementing an OGC WMS Client

A Web application is a collection of Web components and configuration information running on a Web server, which manages the interaction between Web clients and the application business logic. Web technologies are usually utilised in a Web application

implementation to produce dynamic Web content on demand. Of the many technologies in J2EE, two are of particular interest for this research: Java Servlets [34] and JavaServer Pages (JSP) [33].

A Java Servlet is a Java class implementing the standard interface *javax.servlet*. It produces dynamic content in response to requests (in an *HttpServletRequest* object) from the Web server and outputs HTML by populating an *HttpServletResponse* object.

The JSP is another way to write Java Servlets. Java code fragments are inserted directly into a static HTML page to control dynamic Web content. A JSP document is translated into a Java Servlet, usually at runtime.

The JSP Standard Tag Library (JSTL) [35] is a new technology based on JSP. This library provides a set of standard functional tags and a kind of Expression Language (EL) that can be used in a JSP document instead of Java code to manipulate dynamic Web content.

Enhydra XMLC [12] is an alternative to JSP. It applies the Java Servlet technology and provides an object-oriented mechanism for generating dynamic Web pages from static template HTML pages.

JSP with embedded Java, JSP with JSTL tags, and XMLC are the three approaches under investigation in this research. A detailed introduction of JSP, JSTL, and XMLC is provided in Chapter 2.

1.3 Objective

The objective of the research is to evaluate and compare the efficacy of three Java-based approaches—JSP with embedded Java, JSP with JSTL tags, and XMLC—to implementing a client that interacts with servers supporting the OGC WMS within J2EE framework. This research shows the advantages and disadvantages of each of these three approaches and thus helps developers select the appropriate technology in Web application architecture design, as well as in the development and maintenance of Web pages.

1.4 Methodology

We identify a number of functions that are intended to be representative of a generic WMS client implementation (Section 3.2). These functions are separated into three levels, with the complexity increasing from one level to the next. The WMS client prototype is executed in a locally built Web mapping system. The design of the local system and the generic WMS client are described in Chapter 3.

The generic WMS client has been implemented using the three approaches under investigation. The implementation details are presented in Chapter 4.

With the experience of the implementation, each approach is evaluated and compared with the other two using a set of evaluation criteria (Section 5.1) defined from the perspective of Web application architecture, development and maintenance. They are:

- Separating content from presentation
  - Separating markup from code
  - Separating presentation-oriented tasks from data-oriented ones
- Ease of development
  - Method calling
  - Data type conversion
  - Conditional Web page output
  - Inserting content into HTML tables
  - Dealing with XML/GML
  - Overlapping multiple maps
  - Error Localisation
- Ease of change
  - Changing the page appearance
  - Changing the content
  - Rebuilding updated pages

A detailed description of the evaluation and comparison is discussed in Chapters 5 and 6 respectively. Finally, the conclusion is offered in Chapter 7, and future work is specified in Chapter 8.

Chapter 2

JSP, JSTL and XMLC

JSP, JSTL and XMLC are all Java-based Web technologies for dynamically generating Web content. In this chapter, each technology is introduced and an example is presented to show how they differ one from the other.

2.1 Introduction to the Technologies

Three technologies are introduced in this section: Java Servlets; JavaBeans; and Document Object Model (DOM). They are the essential elements of the JSP, JSTL and XMLC approaches, and will be applied on our implementation.

2.1.1 Java Servlets

A Servlet is a compiled Java class that is managed by a Servlet Container. The Servlet technology allows one to develop Java applications that generate Web content.

An HTTP Servlet must extend *javax.servlet.http.HttpServlet*. When the Web browser requests a page that is produced with a Servlet, the request is processed by a Servlet container in the Web server. The Servlet container creates an instance of the appropriate Servlet and the *init()* method in the Servlet is invoked. After that, the Servlet's *service()* method is called with request and response objects as parameters. The Servlet then executes either the *doGet()* or *doPost()* method and creates the HTML output. Finally, the Servlet container passes the output to the Web browser through the response object.

A Servlet can instantiate other Java classes and perform actions. It can also connect to another Servlets, JSP documents, or HTML pages. The state of the information in an Web application can be managed with scope, which defines the length of time over which the information is maintained and available to be accessed. There are four levels of scope defined in Servlet and JSP: page; request; application; and session.

Page Scope: Page scope is only available to a single Servlet or JSP file. The information stored in the page scope cannot be accessed from outside that page.

Request Scope: The information stored in the request scope will be available across pages and Servlets.

Application Scope Application scope is the broadest level. The information stored in the application scope can be accessed by all of the Servlets and JSP documents that are part of a Web application.

Session Scope Session scope is not tied to particular pages or Servlets, it relates to a particular user or task.

2.1.2 JavaBeans

JavaBeans [32] are portable Java components. They can be used to build visual components, perform the business logic of an application, maintain data elements, and so on. Properties, methods, and events are the three most important features of a JavaBean. Properties are public attributes that can be accessed through *get* and *set* methods. Methods are normal Java methods that can be called from other components or a scripting environment. Events are used to notify other components of the change of states. There are some common conventions that all JavaBeans follow:

- A JavaBean must have a constructor without arguments.
- A JavaBean property consists of a member variable and a get method and an optional set method for returning and/or setting the variable.
- The name of the get and set methods contains the property name, such as *getPropertyName()* and *setPropertyName()*.
- The get method does not contain any input arguments.

2.1.3 Document Object Model (DOM)

The DOM [5] is an Application Programming Interface (API) that allows you to access and manipulate the contents of HTML and XML documents. It provides a set of standard objects for representing individual elements and content in an HTML or XML document, with methods for accessing, changing, deleting, or adding those objects.

The DOM represents a document as a hierarchy of nodes, each representing an element, a comment, text, or some other object. This results in a tree-like structure as illustrated in Figure 2.1. Each object implements the *Node* interface. The root of the tree is a document object that also implements the *Document* interface. The document object contains only one child element, which is the root element of the tree representing the document. Starting from the root element, we can traverse the tree to access individual node within the tree.

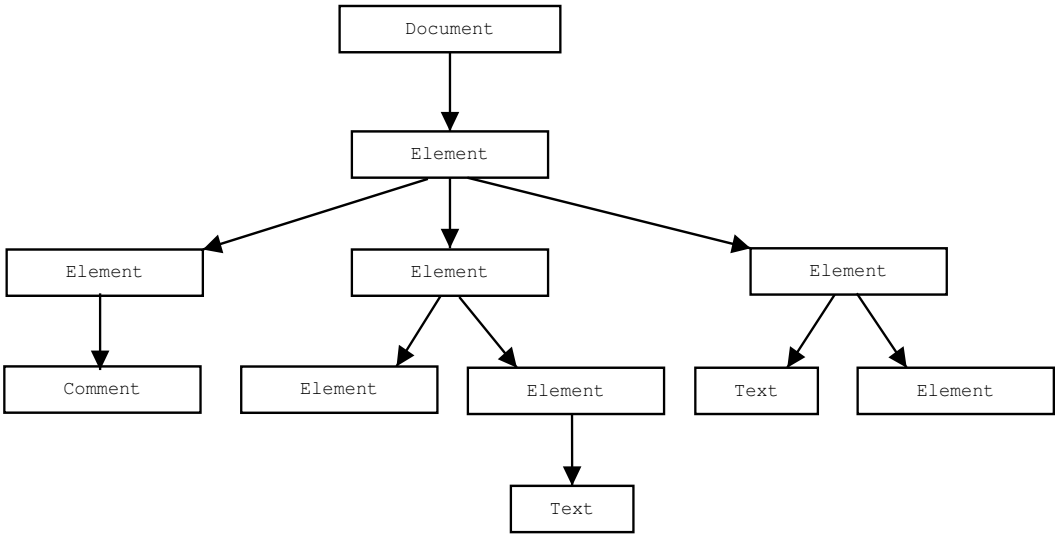

Figure 2.1: A sample DOM tree representing a document

2.2 JavaServer Pages (JSP)

Sun Microsystem's JSP [33] is based on Java Servlets technology. A JSP document is basically an HTML file with embedded fragments of Java code, which generates dynamic content. A JSP file is processed by a JSP container at run time as illustrated

in Figure 2.2. When the Web browser requests a JSP file, the JSP container in the Web server first converts the JSP file to a Servlet and compiles it with more business logic Java code (i.e. JavaBeans). The container then executes the compiled Servlet to generate an HTML page, which is subsequently sent to the Web browser.

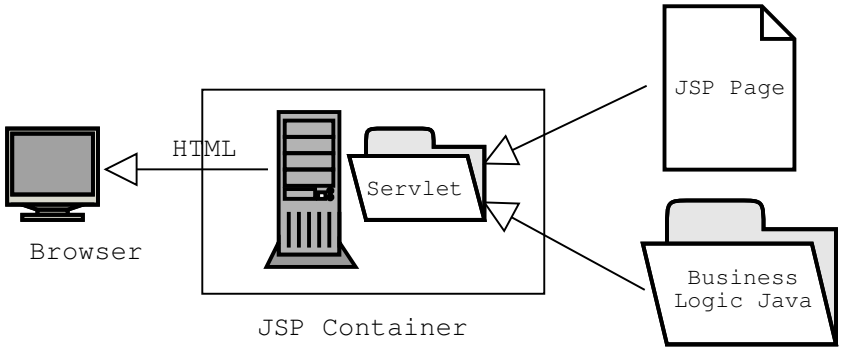

Figure 2.2: JSP processing phases

Aside from HTML, JSP defines three main types of JSP elements you can embed in a page: directives; scripting elements; and action elements:

Directives provide compiler information and let you include class libraries and import tag libraries. There are three directives: page; include; and taglib. They have a syntax of the form ⟨%@ directive...%⟩.

Scripting elements let you specify Java code. There are three types: declarations with the syntax ⟨%!...%⟩; scriptlets with the syntax ⟨%...%⟩; and expressions with the syntax ⟨%=...%⟩.

Action elements let you control the behaviour based on the specific request received by the JSP file, including "include" and "forward" flow control, JavaBeans accessing, and applet plug-ins. They have the syntax ⟨jsp:...⟩ with the tag prefix "jsp".

JSP tags may be extended by custom tags, which are user-generated markup tags. They link to back-end Java code and implement specific user-defined actions. There are many JSP container vendors and each provides their own tag libraries such as Jakarta Taglibs and JRun's library, as well as custom tags written by Java developers

for specific projects. There is no standard that ties them together, and none of them can be used as a standard in all JSP containers [30]. To aggregate and standardize tag libraries, the Java Community Press (JCP) released a standard tag library in mid-2002.

2.3 JSP Standard Tag Library (JSTL)

As JSTL is based on JSP technology, a JSTL page is also a JSP document. JSTL includes a set of standardized actions for almost all common tasks that are needed to produce dynamic content in typical JSP documents in an application. These actions include looping over data; performing conditional operations; importing and processing data from other Web pages; simple manipulating of XML; accessing database; and text formatting.

The availability of standard tags means that not only are JSP developers no longer required to create custom tags for common tasks but also that once they are familiar with JSTL they are able to use the appropriate tags in any compliant JSP container. The Apache Jakarta project has already implemented JSTL, which can be used with Java Servlet 2.3 and JSP 1.2.

JSTL specifies four separate tag libraries, each containing actions targeting a specific functional area:

Core library contains tags with prefix "c". It contains general actions such as flow control actions (conditionals, iterators), evaluating expressions and outputting the results, manipulating scoped variables, and accessing URL-based resources.

XML processing library contains tags with prefix "x". It addresses the basic XML manipulation actions including parsing an XML document, accessing a parsed XML document, looping over elements, conditional processing based on node values, and XSLT (eXtensible Stylesheet Language for Transformations) [39] transformations. XML actions use the XML Path Language (XPath) [38] to specify and select parts of an XML document.

I18N capable formatting library contains tags with prefix "fmt". It supports internationalization and general formatting actions.

Database access library contains tags with prefix "sql". It provides basic capabilities to interact with relational databases.

JSTL also defines an Expression Language (EL), which is supported directly by the JSP 1.2 specification and will be formally defined within the next generation of JSP (JSP 2.0), for accessing and manipulating resource data without using programming languages such as Java. A JSTL expression look like:

```
"${expression}"
```

For example, instead of the Java/JSP expression, such as

```
<%= session.getAttribute("layer").getTitle() %>
```

JSTL can access the data using JSTL expressions:

```
<c:out value="${sessionScope.layer.title}"/>
```

The JSTL tries to kick Java code out of the JSP document, but flow control logic like iteration and conditional operation may still confuse Web designers who are using an HTML editor to design Web pages. This problem is addressed by XMLC.

2.4 Extensible Markup Language Compiler (XMLC)

Enhydra XMLC [12] was developed by Lutris Technologies and is integrated with the open source Enhydra application server. It can also be used separately[1].

XMLC provides a very different method from JSP and JSTL to generate dynamic pages. Compared with the "Pull model" implemented by JSP, which embeds programming languages inside a markup page, XMLC uses a "Push model" that allows the developer to manipulate a template page programmatically [9].

As illustrated in Figure 2.3, a static HTML template page with unique "id" attributes, which override the "id" attribute of the Cascading Style Sheet(CSS), and mock-up content is created first. Using XMLC, the template HTML page is converted into a Java class, where the HTML page is represented using the DOM. The XMLC creates accessor method *getElementXXX()* for every standard HTML tag that includes

[1]We used standalone XMLC for our WMS client implementation.

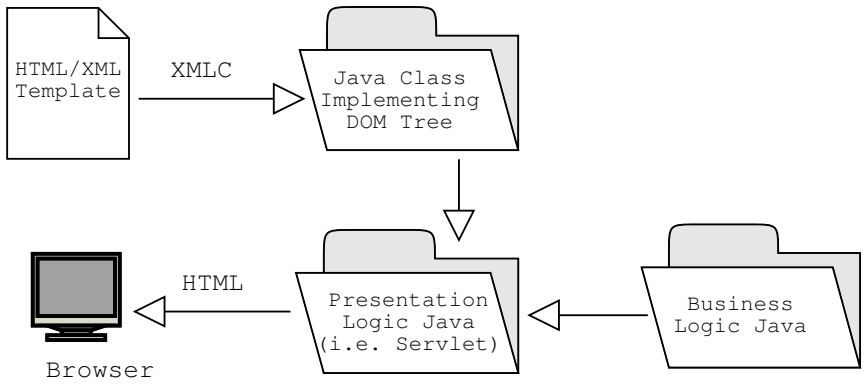

Figure 2.3: XMLC processing phases

an attribute id="XXX" and *setTextYYY(String)* for the tag with text content and an attribute id="YYY". With these methods and other standard DOM APIs, the attributes, content, and nested tags of the elements with "id" attributes can be replaced and removed in a manipulation class. The manipulation class can be implemented as a Servlet. In the Servlet, the page class representing the template page is instantiated, the HTML elements are accessed using the above-mentioned methods, the content is retrieved from the business logic and inserted in the template page as required, and finally the generated page is output to the browser.

2.5 A Simple Example

In this example, a very simple Web page is generated using three approaches: JSP with embedded Java, JSP with JSTL tags, and XMLC. The page gets a parameter named "result" from a request. If the value of the parameter is "won", the page prints out "It is good news! We won the game" and the word "good" will be displayed in red. If the parameter value is "lost", a sentence "It is bad news! We lose the game" is printed out and the word "bad" will be displayed in blue.

We use the JSP with embedded Java approach first. In *game1.jsp* (see Figure 2.4), small pieces of Java code are inserted into the HTML to receive the request parameter and control the conditionals. The Java code is embedded with the scriptlet $\langle\%\ldots\%\rangle$, and the expression scripting element $\langle\%=\ldots\%\rangle$ on line 14 is used to evaluate and print out the value of the variable.

```
 1  <!-- game1.jsp with Java embeded -->
 2
 3  <html>
 4  <head><title>game</title></head>
 5  <body>
 6  It is
 7  <% String result = request.getParameter("result");
 8     if(result.equals("won")) {
 9  %>
10  <font color="red">good</font>
11  <% } if(result.equals("lost")) { %>
12  <font color="blue">bad</font>
13  <% } %>
14  news! We <%= result %> the game.
15  </body>
16  </html>
```

Figure 2.4: JSP example: game1.jsp

In *game2.jsp* (see Figure 2.5), we replace the scripting elements with JSTL tags to control the dynamic content. The JSTL library must first be declared on line 5 using a taglib directive to make it available in the page. The request parameter is accessed using EL rather than Java and set to a variable using the tag ⟨c:set⟩. Conditionals are controlled using tag ⟨c:if⟩, and tag ⟨c:out⟩ performs a similar function to the ⟨%=...%⟩ (line 14 of Figure 2.4) to evaluate and print out the variable value.

When generating the Web page using XMLC, we follow this process: create a template page; compile the template page to a Java class; and manipulate the page class to generate final page.

Step one, an HTML template file *game.html* (see Figure 2.6) with mock-up contents is created. The "id" attributes are added to every tag whose attributes and/or text content might be changed. We add attribute id="state" to tag ⟨font⟩ and id="result" to tag ⟨span⟩. We can use the XMLC command with the "-dump" option to produce a DOM hierarchy structure (see Figure 2.7) that represents the HTML template file. Each tag in the HTML file has an element implementation type corresponding to it. The XMLC command looks like:

```
 1   <!-- game2.jsp with JSTL -->
 2
 3   <html>
 4   <head><title>game</title>
 5   <%@ tablib uri="http://java.sun.com/jstl/core" prefix="c" %>
 6   </head>
 7   <body>
 8   It is
 9   <c:set var="result" value="${param.result}"/>
10   <c:if test="${result=='won'}"> <font color="red">good</font> </c:if>
11   <c:if test="${result=='lost'}"> <font color="blue">bad</font> </c:if>
12   news! We <c:out value="${result}"/> the game.
13   </body>
14   </html>
```

Figure 2.5: JSTL example: game2.jsp

```
<!-- Template html file game.html -->

<html>
<head><title>game</title></head>
<body>
It is <font id="state" color="red">good or bad</font> news!
We <span id="result">won or lost</span> the game.

</body>
</html>
```

Figure 2.6: XMLC example: template page game.html

```
$xmlc -dump game.html
```

Step two, we use another XMLC command with the "-keep" option to compile the game.html:

```
$xmlc -keep game.html
```

This will create a compiled Java class *game.class* and a source code file *game.java*, which traverse the above DOM tree with access to each element with an "id" attribute.

```
The DOM tree hierarchy representing the game.html:

    LazyHTMLDocument%[T]:
        LazyComment:  Template html file game.html
        HTMLHtmlElementImpl: HTML
            HTMLHeadElementImpl: HEAD
                HTMLTitleElementImpl: TITLE
                    LazyText: game
            HTMLBodyElementImpl: BODY
                LazyText: It is
                HTMLFontElementImpl: FONT: color="red" id="state"
                    LazyText: good or bad
                LazyText:  news! We
                LazyHTMLElement: SPAN: id="result"
                    LazyText: won or lost
                LazyText:  the game.
```

Figure 2.7: XMLC example: The DOM of the template page

In the readable source Java file *game.java* (see appendix D), there are two *getElementXXX()* and two *setTextYYY(String)* methods for the tags ⟨font⟩ with id="state" and ⟨span⟩ with id="result" as below:

```
getElementState()
getElementResult()
setTextState(String text)
setTextResult(String text)
```

Step three, once we get a Java class presenting the HTML file and some methods we can call to change the HTML element values, a manipulation Java class *gameMan.java* (see Figure 2.8) is created to use the page class. The manipulation class is actually a Servlet that manipulates the presentation of contents as required and prints out the final page. It first creates an instance of the page class *game.class* representing the template HTML page on line 14, and gets a reference to the element ⟨font⟩ on line 16 by calling the method *getElementState()*. Then within the conditional structure it sets the "color" attribute and the text content of the tag ⟨font⟩, followed by setting

the text of the tag ⟨span⟩ on line 30. Finally, the modified final page is printed out on line 32 through the method *toDocument()*, which will create the HTML that the user will see.

2.6 Summary

In this Chapter we introduced JSP, JSTL, and XMLC, and demonstrated each of these approaches using a simple example. The three technologies are designed to generate dynamic Web content under the Java Servlet architecture. Rather than processing HTML in Java classes with the Servlet, the JSP inserts Java code fragments into a regular HTML page. The JSTL tries to use standard tags and EL instead of the Java code in the JSP file to control the dynamic elements. The XMLC separates the HTML design and the Java programming, and uses a totally object-oriented method to create a dynamic Web page.

Three fundamental technologies—Java Servlets, JavaBeans, and DOM—were also introduced in this Chapter. They are used by JSP, JSTL, and XMLC, and applied in the WMS client implementation that will be presented in Chapter 4.

```
 1  // Manipulation class gameMan.java
 2
 3  import java.io.*;
 4  import javax.servlet.*;
 5  import javax.servlet.http.*;
 6  import org.w3c.dom.html.*;
 7
 8  public class gameMan extends HttpServlet {
 9     public void doGet(HttpServletRequest req, HttpServletResponse res)
10        throws ServletException, IOException {
11        res.setContentType("text/html");
12        PrintWriter out = res.getWriter();
13        // Create an instance of the HTML page object
14        game game = new game();
15        // Get a reference to the <font> element
16        HTMLFontElement state = game.getElementState();
17        // Get request parameter
18        String result = req.getParameter("result");
19        if(result.equals("won")) {
20           // Change the "color" attribute of <fone> element
21           state.setColor("red");
22           // Change text within <font> tags
23           game.setTextState("good");
24        }
25        if(result.equals("lost")) {
26           state.setColor("blue");
27           game.setTextState("bad");
28        }
29        // Change the text within <span> tags
30        game.setTextResult(result);
31        //Print out the modified page
32        out.println(game.toDocument());
33     }
34  }
```

Figure 2.8: XMLC example: The manipulation class gameMan.java

Chapter 3

Design of a Generic WMS Client

In this Chapter we present the design issues of the WMS client that has been implemented. We begin with a description of the local Web mapping system that was used in this research. After that, we describe the functionality that is representative of a generic WMS client implementation. This essential functionality is implemented in almost all of the OGC WMS clients we investigated (see Section 1.1.2). Based on the identified functionality, four modules are identified for the WMS client. Each module is composed of View, Model, and Controller components.

3.1 Designing a Local Web Mapping System

A local Web mapping system was assembled to provide an experimental environment in which to execute the implemented WMS client prototype. The architecture of this system is illustrated in Figure 3.1. In this system:

- The viewer is actually a series of HTML pages running inside a Web browser that can interact directly with a map server via HTTP.
- The WMS client manages the viewer interactions with WMS servers via HTTP, and dynamically generates Web pages that can be viewed in the Web Browser.
- The WMS server is a map server, which implements the OGC WMS specification (version 1.1.0), providing three OGC Web Mapping operations (GetCapabilities, GetMap, and GetFeatureInfo) for its clients. The server accepts requests from

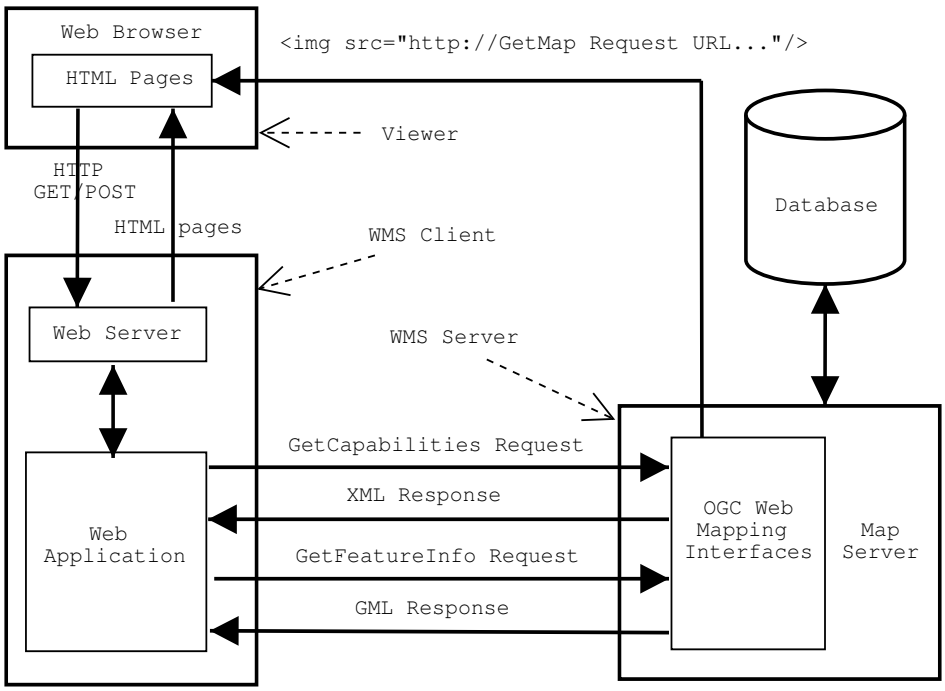

Figure 3.1: Local Web mapping system architecture

the WMS client and the viewer in the form of HTTP URL strings, and returns results encoded as XML, GIF, GML, and so on.

- The database stores geo-feature data that can be accessed and utilised by the WMS server to generate GML documents or to draw maps.

As illustrated in Figure 3.1, the user interacts locally with the viewer and submits HTTP GET/POST requests to the WMS client. The JSP/Servlet container in the Web server intercepts user requests and parses them before forwarding them to the Web application. The Servlets and the other Java components that constitute the Web application then process these requests, and return dynamically generated HTML pages to the Web browser via the JSP/Servlet container and Web server. In practice, GetCapabilities and GetFeatureInfo are requested from the Servlets or other Java components, while a GetMap request is used as an image source embedded in the generated HTML pages in the form of ⟨img src=GetMap request url/⟩. The map retrieved from the WMS server is displayed directly in the Web browser.

3.2 Functionality

The design begins with assessing the functional requirements for the generic WMS client we are implementing. The Use case analysis is applied to identify the actors in the system and the operations they may perform. The Use Case diagram for the WMS client we implemented is shown in Figure 3.2.

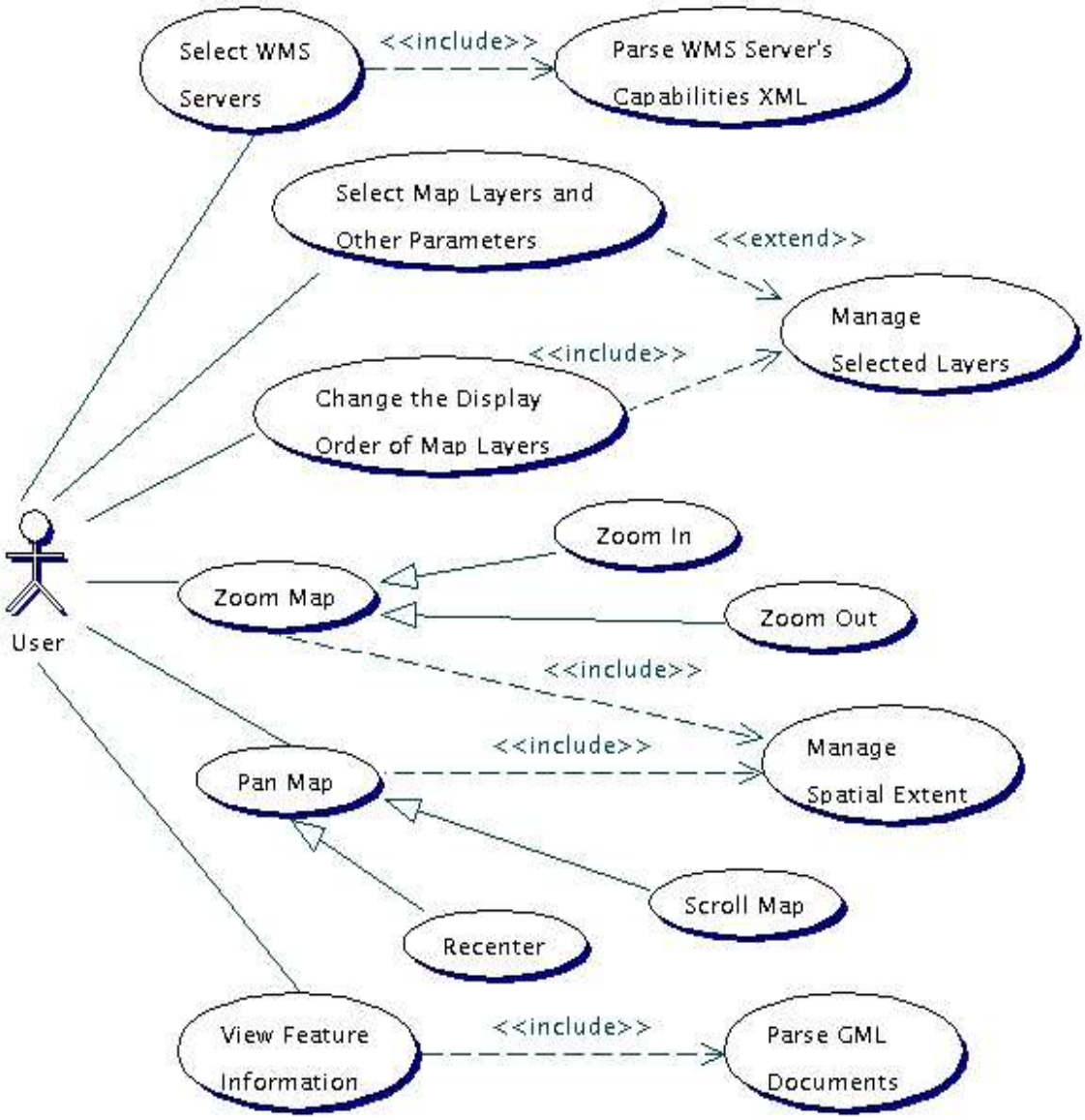

Figure 3.2: WMS client Use Case diagram

Only one actor, the user, interacts with the WMS client through the Web browser. We associate these operations with three levels of functionality, each with complexity increasing from one level to the next: a simple prototype; multiple request enabled; and multiple WMS server interaction.

3.2.1 A Simple Prototype

On the first level, a WMS client prototype with simple operations is implemented. Users may select only one WMS server with which to interact, although several WMS servers could be available.

Map layers but no other capabilities information are presented to the user. Users can select any layers they are interested in and submit them to request a map. Other parameters, such as SRS, initial bounding box and image format, used for generating a map are set with fixed values parsed from the capabilities XML document. They are neither changeable nor selectable.

The WMS server returns a map showing all the requested layers. The display order of the layers is determined by the sequence of the layer names listed in the GetMap request, that is, a WMS server "renders the requested layers by drawing the leftmost in the list bottommost, the next one over that, and so on" [25]. For example, in the GetMap request, the Layers parameter might look like "...&Layers=rivers,loads, bridges&...". Consequently, in the generated map the user may see some bridges cross the rivers and roads. Once a map is loaded, the layers cannot be reordered by the user.

The requested map can be zoomed and scrolled, but recenter is unavailable. The user can also query features by selecting a location on the map. The features of each layer are associated with the location is presented to the user.

3.2.2 Multiple Request Enabled

On the second level, multiple GetMap requests are sent if the user selects more than one map layer—one layer per map. For example, if the user selects three layers (e.g. roads, lakes, and airports), three separate maps will be requested. Since the client interacts with only one WMS server that usually provides geographic information for a specified spatial extent, the maps that are produced for each layer have the same bounding box, SRS, and output size. Therefore, the WMS client can accurately overlay

these maps to produce a composite map as illustrated in Figure 3.3, and the user is able to adjust the layers' display order by changing the maps overlapping sequence to get a different view. The background of each single-layer map must be transparent[1] so that the lower layers will not be hidden by those on top of them during overlapping.

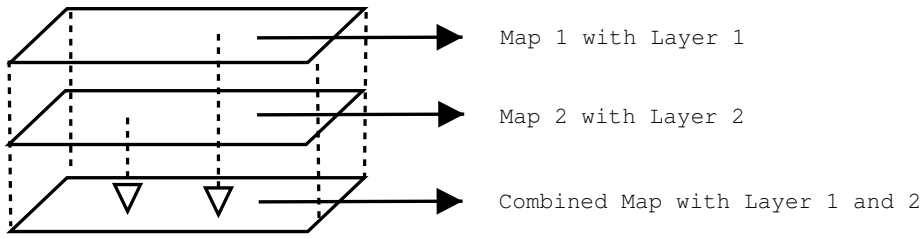

Figure 3.3: Map overlapping

Similarly, multiple GetFeatureInfo requests will also be sent if the user wants to view the feature information—one layer per GML file. As we don't know which map layers are associated with the pointed location, the features of each layer displayed on the map will be requested (zero feature may be contained in a GML file if no layer is associated with the selected location).

Without multiple requests, the WMS client can also adjust the display order of map layers by changing the sequence of layer names in the request, but when a client interacts with several servers concurrently, multiple requests will be necessary. This is implemented in level three.

3.2.3 Multiple WMS Server Interaction

On the third level, WMS servers with different capabilities may be interacted with simultaneously. Here the WMS client must request all user-selected WMS servers for their capabilities information.

The user is allowed to set or choose parameters, such as bounding box, SRS, image formats, width and height, and select map layers that may come from different WMS servers. For example, a WMS client could interact with three WMS servers that specify different geographic information of an area: the first server contains atmospheric information; the second presents population distribution; and the third specifies various

[1]Image formats such as GIF and PNG support transparent backgrounds.

context layers such as roads, rivers, boundaries and coastline. The WMS client is able to build customised maps by requesting single-layer maps from different WMS servers and combining them in different display orders. The map created by these requests may be zoomed, scrolled, and recentered.

When querying features, the layers that are associated with the selected location might be requested from different WMS servers. The user will be told which WMS server a certain layer comes from. Similarly, the features of each layer displayed on the map are requested, and each layer's features are returned in an individual GML document

3.3 Functional Modules

Once the functional requirements have been identified, the WMS client application can be divided into modules based on functionality. Four functional modules were identified: capabilities module; legend control module; map control module; and feature module. The relationship among modules is shown in Figure 3.4. Each functional module is structured in Model-View-Controller architecture. We introduce the architecture that we used to implement the WMS client before describing each functional module.

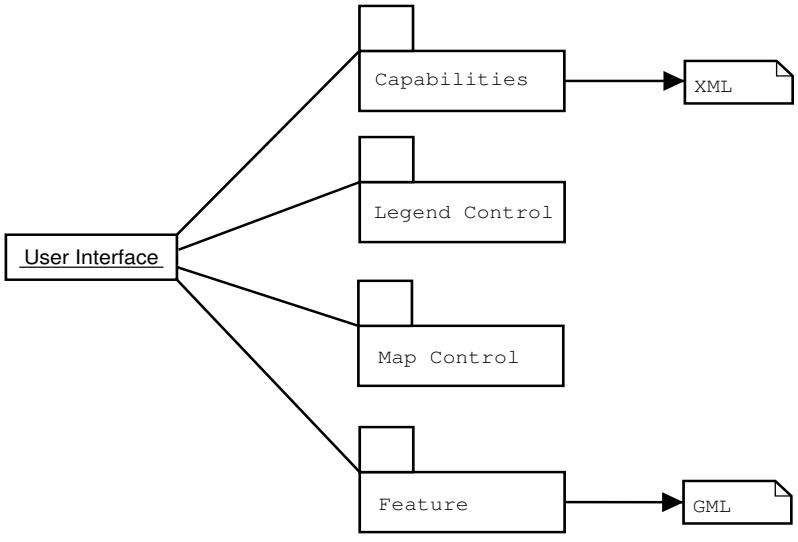

Figure 3.4: WMS client functional modules

3.3.1 The Architecture

In a multitier J2EE Web-enabled application, four tiers are always presented:

- The client tier that is usually provided by the Web browser.
- The Web tier that usually works with the Web server (JSP/Servlet container).
- The Enterprise Java Beans (EJB) tier that usually works with the application server (EJB container).
- The data source tier, which usually includes databases, file systems, or other business services.

Based on these four tiers, various application scenarios are produced [17]. The most suitable scenario is determined by the application's functionality and other requirements such as security and scalability. Considering the functionality identified in Section 3.2, the generic WMS client we implemented was not a large-scale enterprise application with heavy transactional needs. The main duty of the WMS client was to present the application's data—capabilities, maps and features—in response to the user's requests. Consequently, a three-tier Web-centric application scenario [17] was chosen (see Figure 3.5), as follows:

- The Web browser was the client tier.
- The Web tier was the WMS client divided into four functional modules.
- The data source tier was the capabilities XML documents, GML documents and maps retrieved from the WMS server. As the maps are directly displayed on the browser (See Figure 3.1). they are not presented in Figure 3.5.

The EJB tier is not applied as the Web tier is responsible for both presentation and business logic. The software components decomposed from the functional modules are organised following the Model-View-Controller (MVC) architecture [22]. MVC is a widely used architecture suited for interactive applications. It decouples the presentation logic from the business logic in an application.

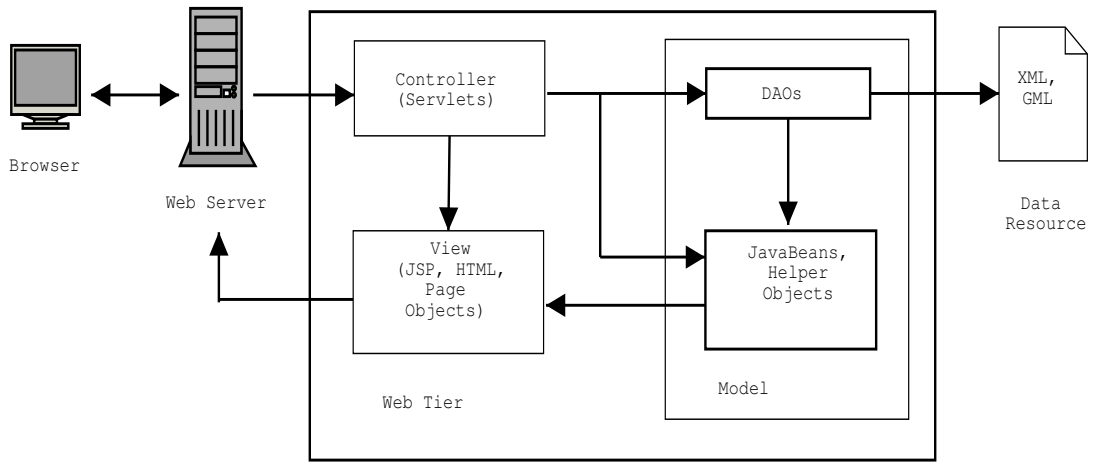

Figure 3.5: WMS client architecture

The View Layer

View handles the presentation logic. It accesses data from the Model and generates the graphical and textual output as required. It also forwards user input to the Controller. In our implementation, JSP and XMLC were used to handle Views. When using JSP, HTML pages and JSP documents were used for presentation. In the case of XMLC, Views would be composed of a set of HTML template pages and Java objects where the HTML templates are presented using DOM.

The Model Layer

Model handles the business logic. It manages the application behavior, encapsulates business data, and modify the data in response to instructions from the Controller. Model is independent of the three approaches under investigation. In our implementation, Model was composed of JavaBeans and some helper objects. The Data Access Object (DAO) pattern [8] was applied to access the data resources—WMS servers' capabilities XML files as well as the GML files. The DAO pattern decouples the data resource access from the business logic.

The Controller Layer

Controller controls the application flow and connects View to Model. It processes user requests, maps them into actions to be performed by Model, and selects which View to display. In the "Model 2 JSP" architecture [14], a Controller is typically implemented as a Servlet. When using XMLC, the manipulation classes can be treated as page controllers and are also implemented as Servlets. In our implementation, each functional module had controllers to manage the behaviors within the module and to interact with other modules.

3.3.2 Capabilities Module

The capabilities module handles the user's selection of WMS servers and maintains the WMS servers' capabilities metadata, which are parsed from capabilities XML documents fetched from WMS servers.

A WMS server catalog view is created. It lets the user view and select available WMS servers. The user's selection will be forwarded to a controller, which uses a data access object to access capabilities XML documents that are retrieved from selected WMS servers. The XML documents are parsed and capabilities data are stored in data objects that are implemented as JavaBeans. The controller will finally forward its control to a capabilities view, where the capabilities data are retrieved from the data objects and then displayed. The capabilities view can be very simple as it contains only one table or selection list when the available map layers of an individual WMS server are listed. To satisfy the third level functionality as described in Section 3.2.3, it is necessary to use a big form containing a number of input elements, where capabilities data in addition to map layers are displayed and can be selected and set by the user.

3.3.3 Legend Control Module

The legend control module manages the user-selected map layers' metadata and their display order. This module can be very simple if the display order of map layers is not changeable.

A controller is used to intercept the request forwarded from the capabilities view. Various actions are performed on this controller, depending on the level of functionality

required. At the lowest level (see Section 3.2.1), only parameters related to the selected map layers are passed in from the capabilities view and there is no requirement to adjust the display order of the map layers. The information of the selected layers is accessed from data objects and directly used to build GetMap and GetFeatureInfo request URLs. At the middle level (see Section 3.2.2), the display order of map layers is changeable. The information about selected layers is maintained in data objects, which are responsible for adjusting the sequence of the map layers' output according to the request from the map view. At the third level (see Section 3.2.1), the request received from the capabilities view includes parameters set by the user in addition to the map layers. The spatial parameters are maintained in a separate data object that will be used in the map control module. Finally, this controller will forward its control to another controller controlling the map output.

3.3.4 Map Control Module

The map control module handles the user's operations on the map and maintains spatial parameters such as the bounding box for generating maps.

A map controller is used to manipulate the operations on the map. It intercepts the request from the map view and forwards the control to various targets as required. When the map is zoomed or panned, the map controller will update the spatial parameters maintained in the data object, and forward the control back to the map view. When the display order of map layers is changed, the map controller will call the data objects that belong to the legend control module to adjust the sequence in which the map's layers are output, as well as forward the control to the map view. When the feature is queried, it will forward the request to the controller that belongs to the feature module.

The map view contains three parts: a map and some arrows that are used to scroll the map; some options including zoom in, zoom out, recenter, and query feature; if the maps layers' display order is changeable then a selection list showing the titles of all the available legends will be added. As a result, it is possible to adjust the sequence in which the titles are listed.

3.3.5 Feature Module

The feature module handles the user's querying about feature information and maintains the feature data, which are parsed from GML documents retrieved from WMS servers.

The controller in this module receives the request forwarded from the map controller, and also uses a data access object to access GML documents that are retrieved from selected WMS servers. The GML documents are parsed and the feature information about the pointed location is stored in various data objects that are also implemented as JavaBeans. The controller will forward the control to a feature information view that displays detailed information about features.

When the multiple request function is enabled, the feature information about each layer is encapsulated in an individual GML document (see Section 3.2.2 and Section 3.2.3). Therefore, we can create a feature summary view to show how many features are found for each layer and which WMS server the layer comes from. This is invoked in the controller. Then from within the feature summary view, the feature information view can be invoked to display detailed information about features of an individual layer.

3.4 Summary

In this Chapter we described the design of a local Web mapping system and the functionality and architecture of the WMS client we implemented.

We identified the functionality that was representative of a WMS client in general. The functionality was divided into three levels and the complexity increased from level one to level three: a simple prototype; multiple request enabled; and multiple WMS server interaction.

Four functional modules were identified according to the functionality: capabilities, legend control, map control, and feature. The MVC architecture was applied on the implementation of each module, and the DAO pattern was adopted for accessing the WMS server's capabilities XML file and GML file. Based on the modules, five views were identified: WMS server catalog view, capabilities view, map view, feature summary view, and feature information view.

Considering that the WMS client is a Web application that handles user interactions and presents data as required, without complex transactions and high security and scalability requirements, a Web-centric J2EE application model was chosen.

Chapter 4

Implementation

In this Chapter we first introduce the building of a local Web mapping system and then describe in detail the implementation of the WMS client based on the four functional modules identified in Chapter 3: capabilities; legend control; map control; and feature. Finally, the implementation of the three generic operations used by these modules is presented.

4.1 Building the Local Web Mapping System

Based on the architecture designed in Section 3.1, a local Web mapping system was installed and configured using a combination of open source software and the OGC WMS clients implemented for this research. Figure 4.1 shows the system we used.

The viewer can be provided by any Web browser supporting HTML 4.0, such as Internet Explorer, Netscape Navigator, and Mozilla. We deployed Apache HTTP server 1.3 [2] as the Web server, and Apache Tomcat 4.0.3 [18] was set up as the JSP/Servlet container integrating with the Apache HTTP server.

The WMS server was set up using the University of Minnesota's (UMN) MapServer 3.6 [21], which implements Version 1.1.0 of the OGC WMS specification [25]. MapServer consists of only one executable file named "mapserv", which knows how to handle WMS requests. It is a CGI program and runs in the Apache HTTP Web server. For OGC WMS compliance, MapServer must be compiled together with PROJ.4 [29], which is a cartographic projections library. MapServer also uses GD library [13] to render GIFs and PNGs. Each MapServer needs a mapfile (a control file with the suffix .map), which

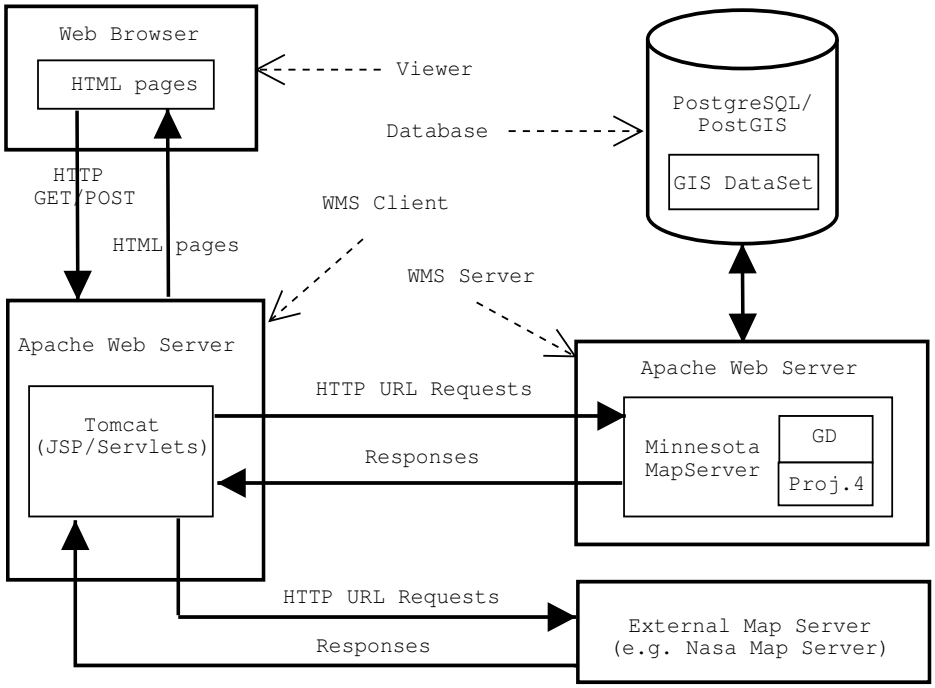

Figure 4.1: Practical Web mapping system

is a configuration file defining display and query parameters and the data source to be used. In Appendix C is shown the mapfile we set up for MapServer with UMN MapServer demo dataset for Itasca County.

We used PostGIS [27] as a vector database to store geo-feature data. PostGIS is an extension to the PostgreSQL [28] object-relational database system that allows GIS objects to be stored in the database. PostGIS supports the "simple features" such as Point, LineString, Polygon, and so on, that are defined by the OGC [23]. The vector format data is a coordinate-based data structure that can be used directly by MapServer to draw maps or generate GML documents.

4.2 Implementation of a Generic WMS Client

As introduced in Section 3.3, four functional modules were identified for the generic WMS client we implemented. In this section, the software components implemented for each module are described.

4.2.1 Capabilities Module

The structure of the capabilities module is shown in Figure 4.2[1]. The index page is an HTML page, from which the controller *ServerHandler* is invoked when the user selects WMS server(s). *ServerHandler* uses a DAO to access capabilities XML documents.

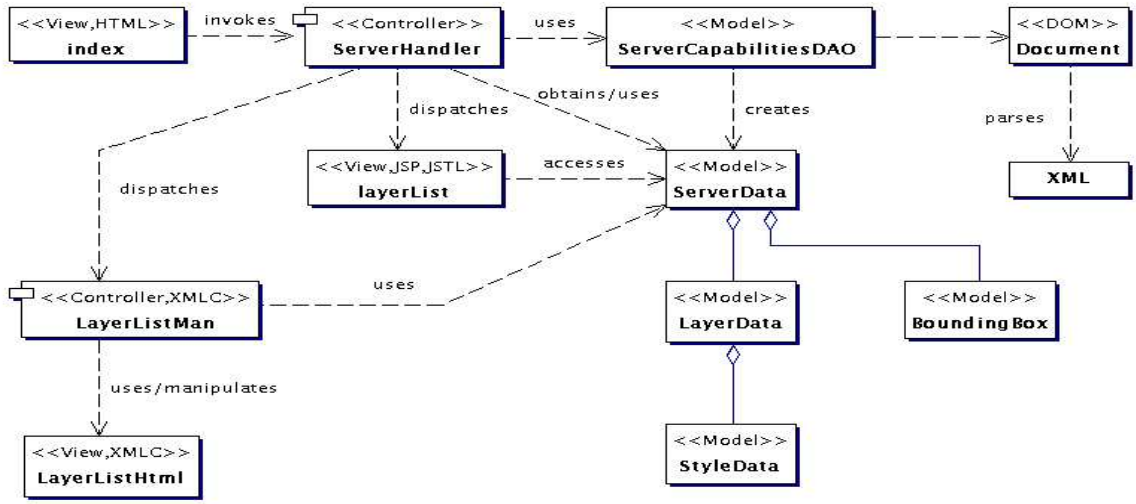

Figure 4.2: The capabilities module class diagram

The index page *index.html* (see Figure 4.3) is the entry page that is representative of the WMS server catalog view, where three available WMS servers are listed.

ServerHandler is a controller Servlet that handles the request from the index page. It uses *ServerCapabilitiesDAO* to load the capabilities of the user-selected WMS server(s), and forwards the request to *layerList.jsp* that implements the *layerList* page when JSP is used. When using XMLC, the control is forwarded to *LayerListMan*.

LayerListMan is a Servlet manipulating the *layerList* page when XMLC is used. It manipulates *LayerListHtml*, which is a page class representing the HTML template *layerList.html*, and prints out the generated *layerList* page.

[1]In class diagrams, if the class or object is not labeled with ≪JSP≫, ≪JSTL≫, and ≪XMLC≫ stereotypes , that class or object is applicable to all approaches

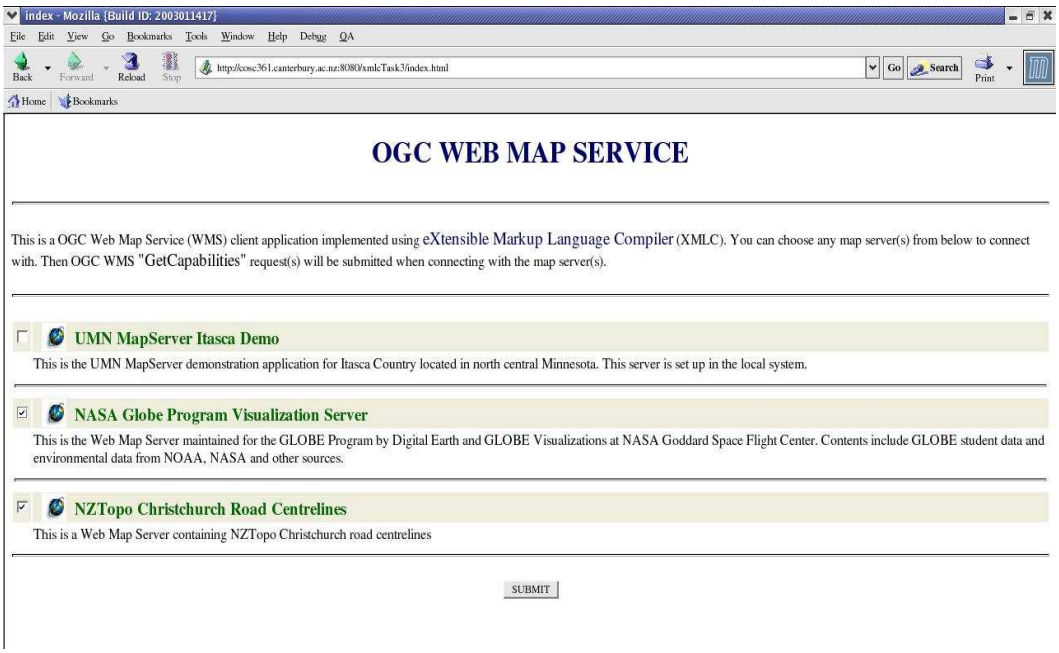

Figure 4.3: The *index* page

ServerCapabilitiesDAO uses DOM APIs to parse the capabilities XML document into a DOM document and to read it. It creates *ServerData* objects where the capabilities information is stored.

ServerData encapsulates WMS servers' capabilities information. There is always one *ServerData* object for each WMS server. It integrates *LayerData* and *BoundingBox* objects.

LayerData encapsulates the metadata of a map layer. It integrates *StyleData* objects.

StyleData encapsulates the metadata of a style.

BoundingBox represents and manipulates rectangular spatial envelopes. It contains a group of four coordinates representing west, south, east, and north respectively.

4.2.2 Legend Control Module

The structure of the legend control module is shown in Figure 4.4. The controller *LayerHandler* is invoked from within the *layerList* page when the user submits the

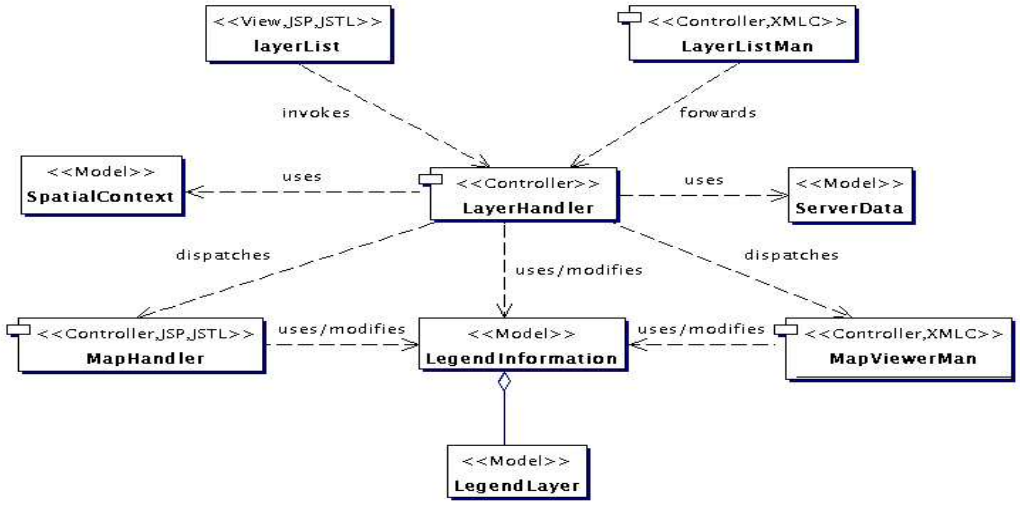

Figure 4.4: The legend control module class diagram

selected map layers.

As mentioned in Section 3.2, only one WMS server is interacted with when implementing level one and level two functionality. When the user selects a WMS server in the index page, a framed page divided into three sections is invoked (See Figure 4.5). The page title appears in the top frame; the left-hand frame contains the *layerList* page listing the map layers (titles) of the selected WMS server; and the right-hand frame (the main area) is a "Welcome" message, which will be replaced by the *mapViewer* Page when the user requests a map (see Figure 4.8 and Figure 4.9). The *layerList* page is representative of the capabilities view. The map layers are presented using either a list of checkboxes or a multiple selection list. Users can either click on the "Submit" button to request a map or reset the selection by clicking the "Reset" button. The buttons are below the layer list.

When we interacted with multiple WMS servers, we implemented a new *layerList* page (see Figure 4.6). All the layer titles of the selected WMS servers are listed in the multiple selection lists—one selection list per WMS server. The user is also allowed to select or set other parameters for map and feature requesting. These parameters include bounding box, SRS, image type, feature type[2], image size, zoom rate, and the maximum number of features to be returned. Users can either select the initial

[2]Only GML is available in this implementation

bounding box from the list or set it manually. Finally, users can click the "Go" button to view the generated map displayed in the *mapViewer* page (see Figure 4.10 on Page 43).

LayerHandler accesses *ServerData* objects for the selected layers' metadata. When implementing level one functionality, the data are directly used for building GetMap and GetFeatureInfo request URLs. In level two, the selected layers' information is encapsulated in *LegendLayer* objects, which are integrated in the *LegendInformation* object. In level three, the information from the selected layers is also collected in *LegendLayer* objects, and the spatial parameters set by the user are stored in a *SpatialContext* object. *LayerHandler* finally forwards the control to *MapHandler* (when using JSP) or *MapViewerMan* (when using XMLC).

LegendInformation integrates *LegendLayer* objects and manipulates the sequence of layers to be shown on the map and listed in the selection list. It is accessed by *MapHandler* or *MapViewerMan* when the order of layers displaying on the map is changed. This object is not necessary for implementing level one functionality.

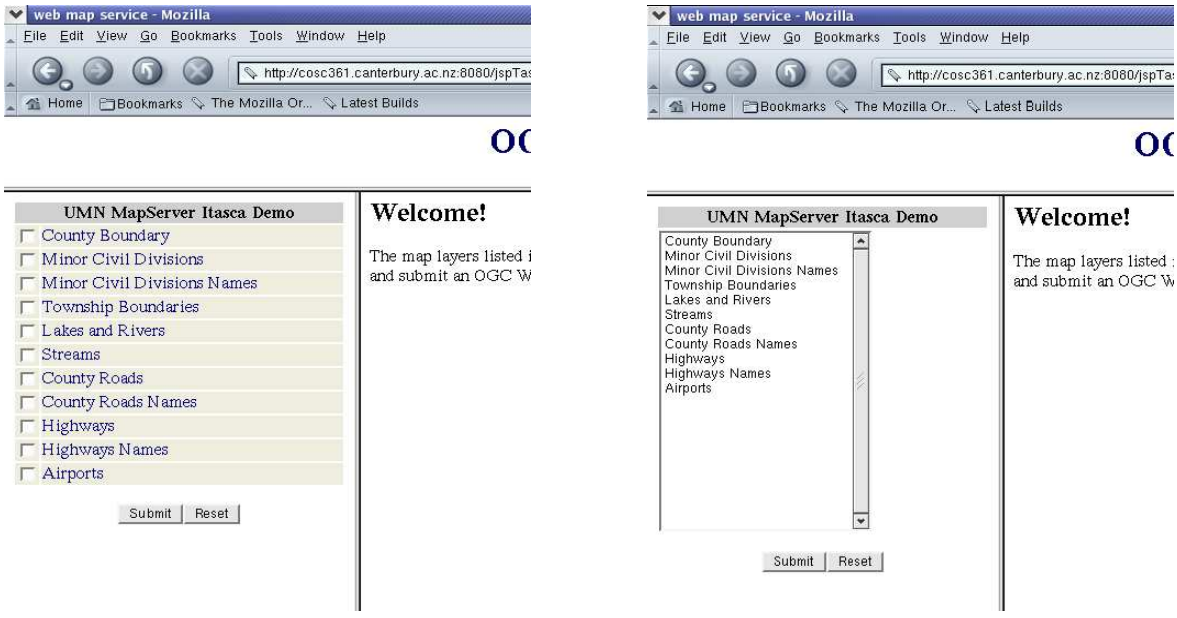

Figure 4.5: A framed pape containging the *layerList* page

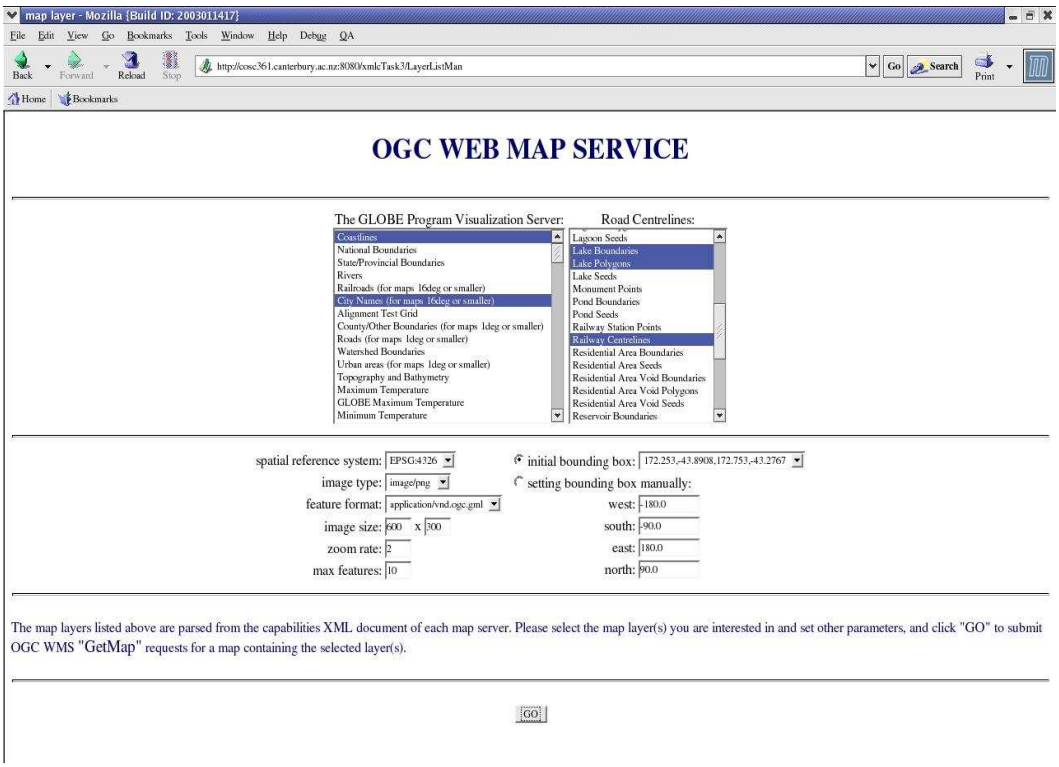

Figure 4.6: Another *layerList* page showing capabilities information of two WMS servers

LegendLayer encapsulates an individual selected layer's information that is needed for requesting map and feature information. This object is not necessary for implementing level one functionality.

4.2.3 Map Control Module

The structure of the map control module is shown in Figure 4.7. The zooming and panning of the map are controlled by manipulating the spatial parameters encapsulated in *SpatialContext* object.

A map containing all the selected layers with the default view scale is loaded when the *mapViewer* page is invoked. See Figure 4.8, a simple *mapViewer* page is shown with the *layerList* page in a framed page. Included in the *mapViewer* page is the *featureInfo* page, which will be introduced in the implementation of the feature module (Section 4.2.4).

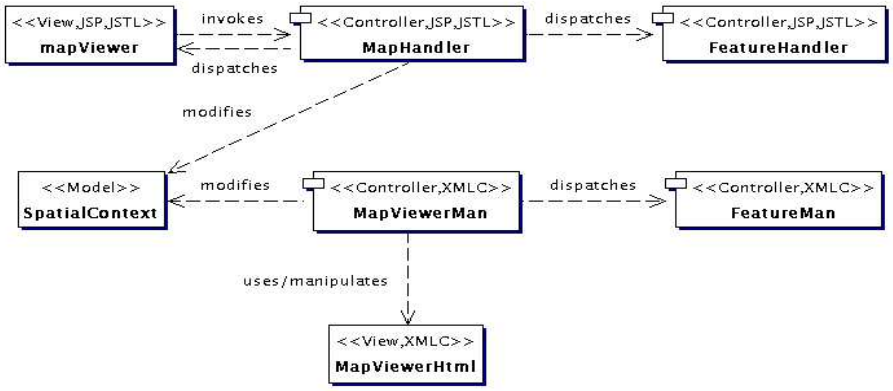

Figure 4.7: The map control module class diagram

The map is implemented as an HTML input element with "image" type, in which the browser automatically sends the (x,y) position of the mouse (relative to the upper-left corner of the image) to the server when the user clicks somewhere on the image. The user can scroll the map in six different directions by clicking the arrows around the map. To the right of the map there are three push buttons that allow the user to either zoom in or out of the map, or recover the map to its default view scale.

A multiple selection list is added in the *mapViewer* page in order to adjust the legends' display order (see Figure 4.9). The map shown on the browser is composed of five overlaying images[3]. In the selection list, the titles of the selected map layers are listed in the order in which they are displayed on the map. The user can use the "Up" and "Down" buttons beside the list to move the highlighted titles and then click the "Go" button to view a map with layers displayed in the requested order. Included in this *mapViewer* page is the *featureSummary* page, which will be introduced in the implementation of the feature module (Section 4.2.4).

We tried using radio buttons instead of push buttons in the *mapViewer* page (see Figure 4.10). The user can select the options and click the map to zoom in/out, recenter, or query information about features of the pointed location. The map shown in Figure 4.10 contains seven layers retrieved from two different WMS servers.

MapHandler is a controller Servlet that is implemented when JSP is used. It is first called from *LayerHandler* and then invoked by *mapViewer.jsp* every time

[3]The implementation of map overlapping is described in Section 4.3.3.

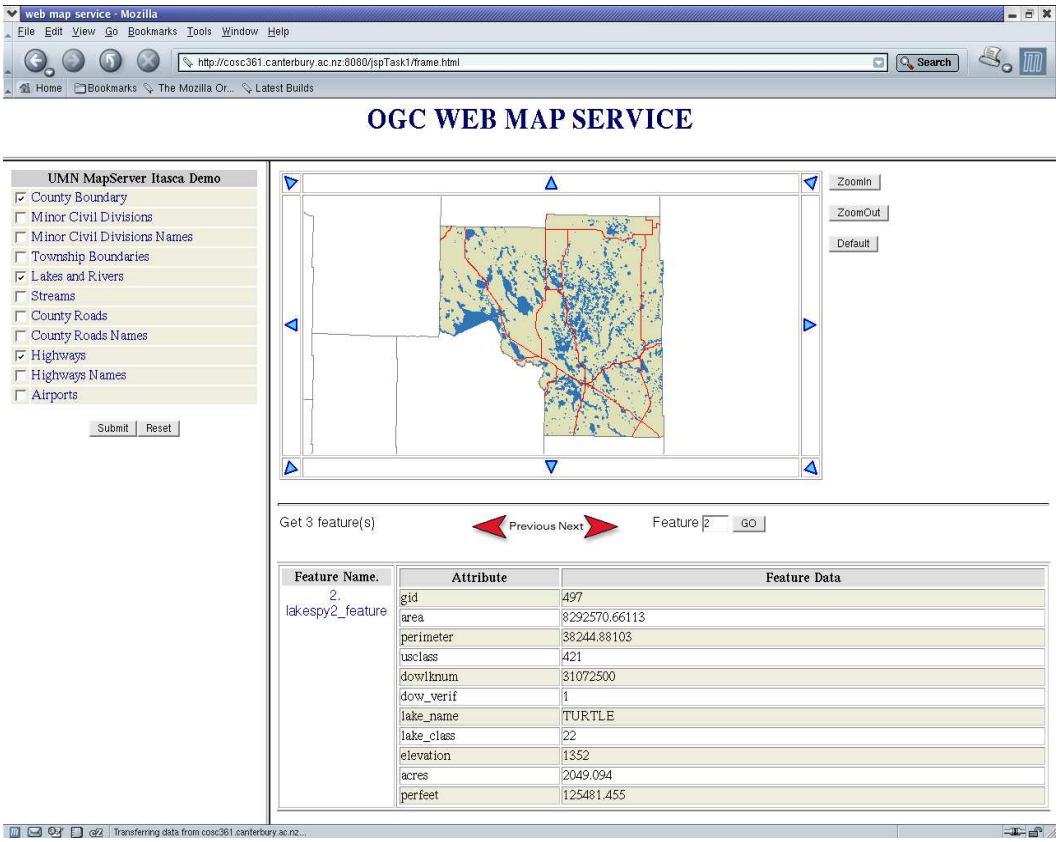

Figure 4.8: A simple *mapViewer* page with the *featureInfo* page included

the map is zoomed, panned, queried, or the legends' display order is changed. When the map is zoomed or panned, it sets or updates spatial parameters in the *SpatialContext* object for a new map query. When the legends' display order is changed, it calls the *LegendInformation* object of the legend control module. Finally, it forwards control back to *mapViewer.jsp* or *FeatureHandler* if features are queried.

MapViewerMan is implemented when using XMLC. It is also first called from *LayerHandler* and then invoked by the *mapViewer* page. It performs similar roles to *MapHandler* but also manipulates *MapViewerHtml*, which is a page class representing the HTML template *mapViewer.html*, and prints out the generated *mapViewer* page. It will forward the control to *FeatureMan* if features are queried.

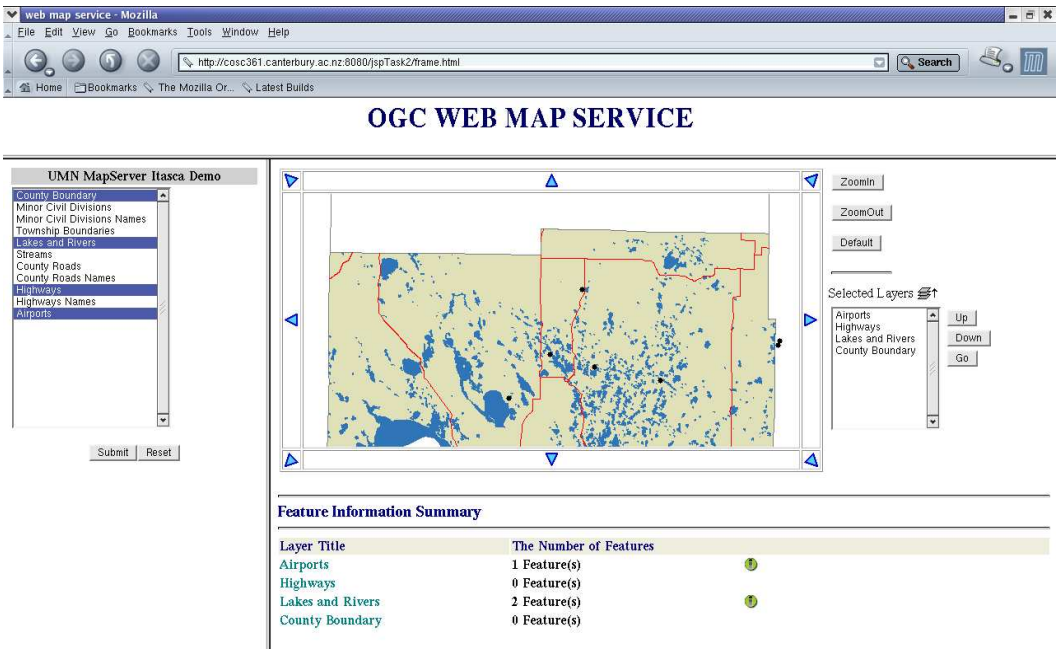

Figure 4.9: The *mapViewer* page with the *featureSummary* page included. Multiple maps are overlapped and the order of layers is changeable

SpatialContext encapsulates and manipulates the spatial extents such as bounding box, SRS, map width and height, that are needed when requesting maps and features.

4.2.4 Feature Module

The structure of the feature module is shown in Figure 4.11. A DAO is also used in this module to access GML documents.

When the user clicks a specific point on the map, the *featureInfo* page is invoked when implementing level one functionality. The *featureInfo* page is included in the *mapViewer* page below the map (see Figure 4.8). The *featureInfo* page displays the feature information that tells the user how many features associated with the pointed location have been retrieved. If more than one feature is retreived then the user can either click the "previous" or "next" arrow to see the previous or the next feature, or enter a sequence number in the entry, and then click the "GO" button to move to the numbered feature directly. If the entered number is out of the range (less than one

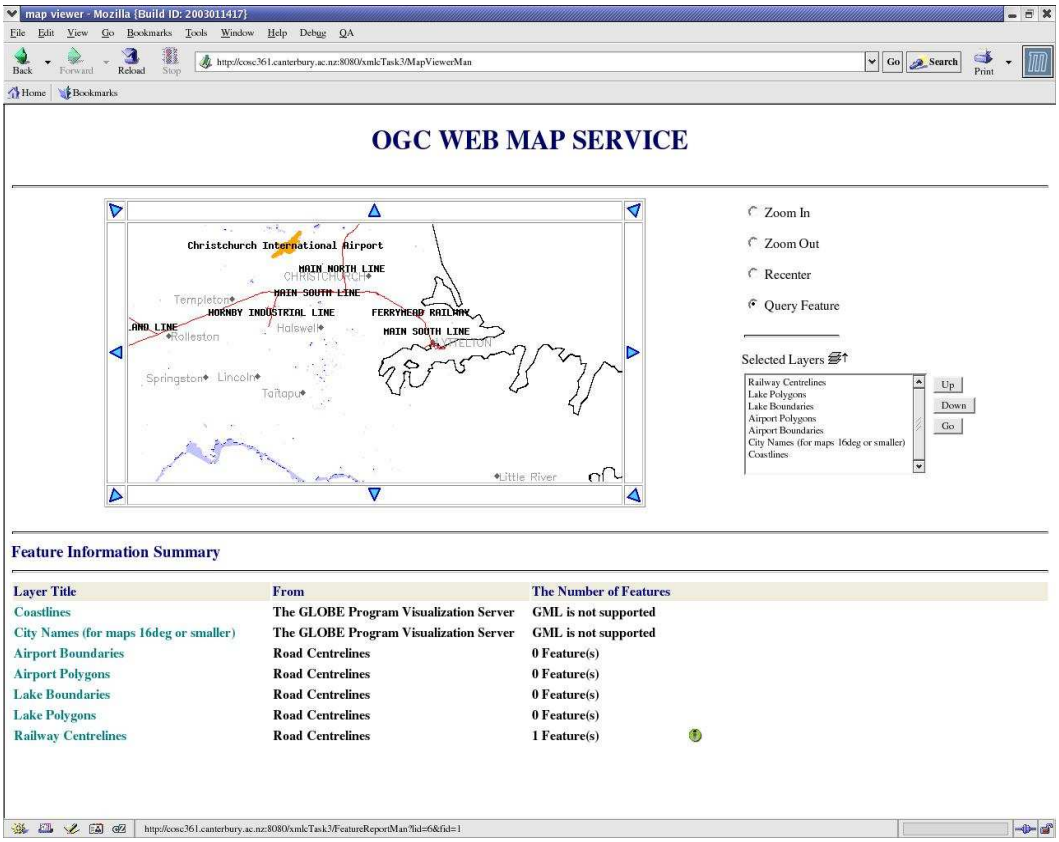

Figure 4.10: Another *mapViewer* page with the *featureSummary* page included. The map is composed of layers retrieved from multiple WMS servers

or larger than the amount of features), an error message will be shown instead of the table listing feature data.

When implementing level two functionality, a *featureSummary* page is invoked first if the user queries features (see Figure 4.12). It specifies how many features are found about each selected layer on the pointed location. When implementing level three functionality, the *featureSummary* page also shows which WMS server each layer comes from (see Figure 4.10). An "i" icon will appear beside the layer with non-zero features. The user can click the icon, which links to the *featureInfo* page (see Figure 4.12), to view detailed information about that layer.

FeatureHandler is a controller Servlet that is implemented when using JSP. It uses *FeatureInformationDAO* to load the feature information about the selected loca-

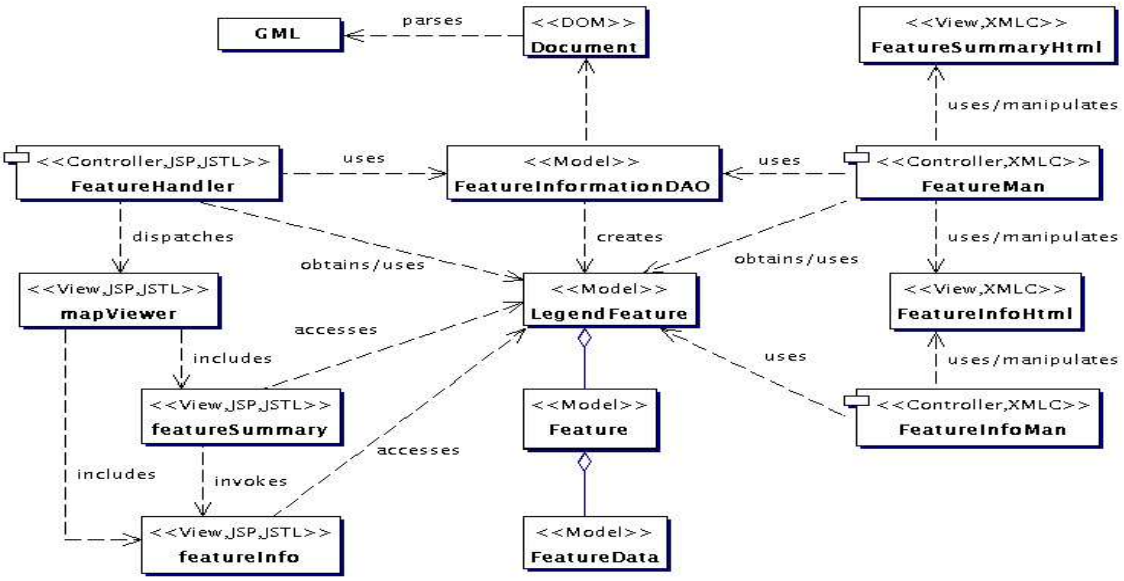

Figure 4.11: The feature module class diagram

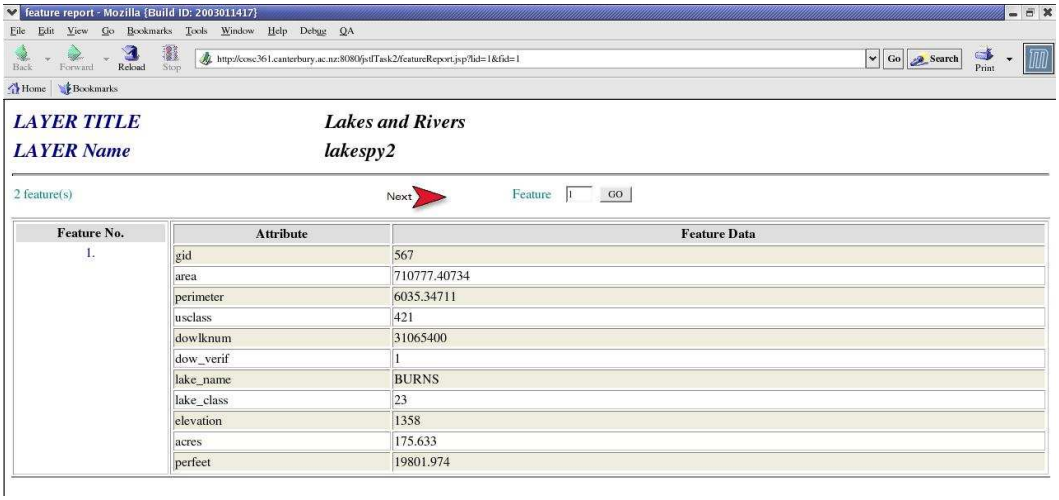

Figure 4.12: The *featureInfo* page

tion on the map. It forwards the request to *mapViewer.jsp*, where *featureInfo.jsp* is included when implementing level one functionality. When implementing level two and level three functionality, *featureSummary.jsp* is invoked first and included in *mapViewer.jsp*, which shows the summary information. Detailed fea-

tures are presented in *featureInfo.jsp*. It is not necessary to use this controller when JSTL "xml" tags are used to handle GML. We tried handling GML documents within *featureSummary.jsp* and *featureInfo.jsp* when using the JSP with JSTL tags approach.

FeatureMan is implemented when using XMLC. It performs similar roles to textitFeatureHandler. In addition, when implementing level one functionality, it manipulates *FeatureInfoHtml*, which is a page class presenting the template page *featureInfo.html*, and prints out the generated *featureInfo* page. When implementing level two and level three functionality, it manipulates *FeatureSummaryHtml*, which is a page class presenting the template page *featureSummary.html*, and prints out the generated *featureSummary* page.

FeatureInfoMan manipulates the output of the *featureInfo* page when using XMLC. It is created for the implementation of level two and level three functionality.

FeatureInformationDAO uses DOM APIs to parse the GML document into a DOM document and read it. It creates *LegendFeature* objects where the feature information of the selected layers is stored.

LegendFeature encapsulates an individual layer's feature information. It integrates *Feature* objects.

Feature encapsulates the data of a single feature. It integrates *FeatureData* objects.

FeatureData encapsulates a single feature data, which contains the data attribute and its value.

4.2.5 Other Components

In addition to the components implemented in the four modules, two other helper objects were implemented:

RequestManager manages the building of request URLs for querying capabilities, maps, and features.

XmlTools contains methods to read data from XML documents using DOM APIs.

4.3 Generic Implementation

In this Section the implementation of three generic operations are introduced. They are: inserting data into tables with XMLC; dealing with XML/GML documents using DOM; and overlaying multiple map images.

4.3.1 The Algorithm for Inserting Data into Tables with XMLC

Tables were used several times in our implementation to present a collection of data on the Web page. In the *layerList* page (see Figure 4.5), a table with two columns is used to display a list of layer titles. The first column of each table row is a checkbox, and the layer title text is presented in the second one.

To generate such a table using XMLC, a prototype table with a sample row is first created in the HTML template (see Figure 4.13). The mock-up data are inserted in the "value" attribute of the ⟨input⟩ element and the second column of the row where the layer title text should be presented, and unique "id" attributes are added to the table, the row, and the cell elements.

In the manipulation class, an algorithm as shown in Figure 4.14 is applied. We first remove the sample row from the prototype table, then repeat the operations of filling the sample row with real data, making a copy of the row, and inserting the copied row into the table within a iteration structure until all data in the collection are inserted.

```
<table id="list" border="0" width="100%">
 <tr bgcolor="#d0d0d0">
  <th id="serviceTitle" colspan="2">service name</th>
 </tr>
 <tr id="option" bgcolor="#efedde">
  <td valign="top">
   <input id="check" type="checkbox" name="checkedLayers"
          value="layerIndex"></td>
  <td><font id="layerTitle" color="#000080">layer title</font></td>
 </tr>
</table>
```

Figure 4.13: The prototype table created in the template page *layerList.html*

```
remove the sample row from sample table;
for(iterate over the collection of data ) {
    edit the sample row and fill in real data,
      overwriting the old or mock-up content;
    clone the sample row;
    append the cloned row to the sample table;
}
```

Figure 4.14: The algorithm used for inserting data into a table using XMLC

```
LayerListHtml layerList = new LayerListHtml();
HTMLInputElement check = layerList.getElementCheck();
HTMLTableElement list = layerList.getElementList();
HTMLTableRowElement option = layerList.getElementOption();

layerList.setTextServiceTitle(server.getService());
list.removeChild(option);
for(int i=0; i<server.getLayerListSize(); i++) {
    // fill in data, then clone, and append
    layer = server.getLayer(i);
    check.setValue(Integer.toString(i));
    layerList.setTextLayerTitle(layer.getTitle());
    list.appendChild(option.cloneNode(true));
}
```

Figure 4.15: In the manipulation class *LayerListMan*, a table with layer titles listed is generated

The code used for creating the table in the manipulation class *LayerListMan* is shown in Figure 4.15.

In the above example each table row is identical except for the title of layers but at other times each row in a table may have a different structure depending on the data to be inserted. In the *featureSummary* page (see Figure 4.9), each row of the table we created to display every layer's feature summary contains three columns: the first one shows the layer title; the second one shows the number of features that have been found, which determines whether or not a small "i" icon will be displayed in the third

```
<table id="summaryTable" border="0" width="100%">
......
 <tr id="summaryRow">
  <td width="30%" align="left">
   <font id="legendTitle" face="Arial Narrow" color="#008080">
    <b>legendTitle</b></font>
  </td>
  <td width="15%" align="left" id="numCell">
   <font face="Arial Narrow"><b>N Feature(s)</b></font>
  </td>
  <td align="left" id="imgCell">
   <a id="imgLink" href="FeatureInfoMan?legend&fid" target="new">
    <img src="image/info.gif" border="0" height="18" width="18"></a>
  </td>
 </tr>
</table>
```

Figure 4.16: The prototype table created in the template page *featureSummary.html*

column. The prototype table created in the template page is shown in Figure 4.16.

The algorithm shown in Figure 4.14 is not suitable for this example because every time a new table row is inserted, we update the data value as well as modify the structure of the sample row—the icon may be removed or appended. We designed another algorithm (see Figure 4.17) to generate this table. When dealing with each row, we cloned all the elements that needed to be changed with their child elements removed, and then reorganised the cloned elements and inserted them into the sample table.

The code used for creating the table in the manipulation class *FeatureMan* is shown in Figure 4.18. The sample row, cell and anchor elements are removed from their parent nodes respectively before generating the table using an iteration structure. When dealing with each row, we cloned the sample elements whose child element had already been removed and then either inserted the cloned anchor element in the cloned cell if one or more features were found or left the cloned cell empty without inserting an anchor element as its child. Finally, we inserted the cloned row, which contains the cloned cell, to the sample table.

```
remove the sample row from sample table;
remove the sample cells from sample row;
remove the sample elements (level m elements) from sample cells;
remove the sample subelements (level m+1 elements) from the sample
  level m elements;
...
remove the sample subsubelements (level n elements which are various
  in each row) from the sample level n-1 elements;
for(iterate over the collection of data ) {
    edit the sample elements (excluding the level n elements) if necessary;
    clone the sample elements (excluding the level n elements);
    edit the sample level n elements depending on certain conditions;
    clone the sample level n elements;
    append the cloned level n elements to the cloned level n-1 elements
      depending on certain conditions;
    append the cloned level n-1 elements to the cloned level n-2 elements;
    ...
    append the cloned level m elements to the cloned cells;
    append the cloned cells to the cloned row;
    append the cloned row to the sample table;
}
```

Figure 4.17: The modified algorithm used for inserting data into a table using XMLC

The algorithms used for the tables can also be used for the creation of the multiple selection list. They are applied when generating the *layerList* page and *mapViewer* page (see Figure 4.6 and Figure 4.9)

4.3.2 The Use of DOM for Dealing with XML/GML

The DOM is a widely used tool for reading XML documents. It represents an XML document as a tree of nodes that can be traversed and edited with its standard APIs (see Section 2.1.3). In our implementation, the DOM was used in the DAOs to deal with the capabilities XML documents and GML documents. Because GML is based on XML, the DOM is also applicable to GML documents.

In *ServerCapabilitiesDAO*, the capabilities XML document (see Appendix B) re-

```
FeatureSummaryHtml featureSumm = new FeatureSummaryHtml();
HTMLTableElement table = featureSumm.getElementSummaryTable();
HTMLTableRowElement row = featureSumm.getElementSummaryRow();
HTMLTableCellElement imageCell = featureSumm.getElementImgCell();
HTMLAnchorElement imageLink = featureSumm.getElementImgLink();

table.removeChild(row);
row.removeChild(imageCell);
imageCell.removeChild(imageLink);
for(int i=0; i<legendNum; i++) {
   int feaNum = 0;
   String legdTitle = legdInfo.getListLegendStr()[i];
   featureSumm.setTextLegendTitle(legdTitle);
   if(this.allLegendFeature.get(legdTitle) == null)
     featureSumm.setTextNumCell("GML is not supported");
   else {
     feaNum =
        ((LegendFeature)allLegendFeature.get(legdTitle)).getFeatureNum();
     featureSumm.setTextNumCell(feaNum + " Feature(s)");
   }
   HTMLTableRowElement rowClone =
                        (HTMLTableRowElement)row.cloneNode(true);
   HTMLTableCellElement imageCellClone =
                       (HTMLTableCellElement)imageCell.cloneNode(true);
   if(feaNum>0) {
     imageLink.setHref("FeatureInfoMan?legend="+legdTitle+"&fid=1");
     imageCellClone.appendChild(imageLink.cloneNode(true));
   }
   rowClone.appendChild(imageCellClone);
   table.appendChild(rowClone);
}
```

Figure 4.18: In the manipulation class *FeatureMan*, a table with feature summary presented is generated

trieved from the WMS server is first parsed into a DOM document (see Figure 4.19). The class *org.apache.crimson.tree.XmlDocument* implements the standard DOM interface *org.w3c.dom.Document*. The argument "getCapaRequestURL" is a string presenting the GetCapabilities request URL. The root element and all its sub-elements are obtained by using the method *getDocumentElement()*. It can then be traversed and accessed with APIs wrapped in the package *org.w3c.dom*.

The same method is used to parse the GML documents (see Appendix B) in *Fea-*

```
import org.apache.crimson.tree.XmlDocument;
import org.xml.sax.*;
import org.w3c.dom.*;
import java.io.*;
......
public class ServerCapabilitiesDAO {
......
  public static Element loadCapabilities(String getCapaRequestURL) {
    Document xmlDoc = null;
    try {
        xmlDoc = XmlDocument.createXmlDocument(getCapaRequestURL);
        Element root = xmlDoc.getDocumentElement();
        return root;
    } catch(FileNotFoundException e) {
        System.err.println("FileNotFoundException " + e.getMessage());
    } catch(IOException e) {
        System.err.println("IOException " + e.getMessage());
    } catch(SAXException e) {
        System.err.println("SAXException " + e.getMessage());
    }
    return null;
  }
......
}
```

Figure 4.19: The WMS server's capabilities XML is parsed into a DOM document in the ServerCapabilitiesDAO

```
public static ServerData getServerData(Element root) {
  ServerData server = new ServerData();
  ...
  NodeList layerNodes = root.getElementsByTagName("Layer");
  for(int loop=0; loop<layerNodes.getLength(); loop++) {
      Element layerElement = (Element)layerNodes.item(loop);
      int childLayerNum =
               XmlTools.getChildElementSize(layerElement, "Layer");
      if(childLayerNum == 0)
         server.addLayer(getLayer(layerElement));
  }
  ......
  return server;
}
```

Figure 4.20: Retrieve map layers from a WMS server's capabilities XML file

tureInformationDAO. The request string becomes the GetFeatureInfo request URL.

Below we will show in two examples how DOM APIs were used to access the document being parsed.

In the first example *ServerCapabilitiesDAO*, we retrieved map layers from a WMS server's capabilities XML document (See Figure 4.20). The root element of the XML document was input in the method *getServerData()*, where the root element and all its sub-elements were traversed and the capabilities data were retrieved and stored in a *ServerData* object. All "Layer" elements that did not have child "Layers" were accessed. As we could not find a method in DOM APIs to arrive at the number of child elements by supplying the parent element and the name of the child element, we implemented such a function *getChildElementSize(Element e, String tagName)* in the class *XmlTools* by ourselves. Each retrieved "Layer" element was input to the method *getLayer()*, where the metadata of the layer, such as title, name, and style, were accessed.

The second example concerns *FeatureInformationDAO*. The features of each legend showing on the map were retrieved (See Figure 4.21). Similar to handling the capabilities XML, the root element of the GML document was input in the method *getLegendFeature()*. Because the name of the element containing layer features was

```
public static LegendFeature getLegendFeature(Element root) {
  // the GML is not supported
  if(root.getNodeName().equals("ServiceExceptionReport"))
     return null;
  LegendFeature legendFeature = new LegendFeature();

  // get the 2nd level nodes
  NodeList layerNodes = root.getChildNodes();
  int featureNum = 0;
  for(int loop=0; loop<layerNodes.getLength(); loop++) {
      Node layerNode = layerNodes.item(loop);
      if(!layerNode.getNodeName().equals("#text")) {
         // get the third level nodes
         NodeList featureNodes = layerNode.getChildNodes();
         for(int inloop=0; inloop<featureNodes.getLength(); inloop++){
             Node featureNode = featureNodes.item(innerloop);
             if(!featureNode.getNodeName().equals("#text"))
                legendFeature.addFeature(getFeature(featureNode));
         } // end innerloop
         break;
       }
  } // end loop
  return legendFeature;
}
```

Figure 4.21: Retrieve legend features from a GML file

different for each layer, we traversed the entire DOM tree to get the elements with the name ⟨layerName_feature⟩, which was in the third level of the tree (under the root element ⟨msGMLOutput⟩ and its parent element ⟨layerName_layer⟩). From each second-level node whose name was not "#text"[4], we gained the third-level nodes. These were also checked for "#text". The non-text elements were the feature elements we needed and their metadata were stored in a *LegendFeature* object.

[4]Each node in the DOM tree contains a default child node named "#text"

4.3.3 The Use of CSS for Supporting Maps Overlapping

A Cascading Style Sheet (CSS) was used in the *mapViewer* page to define "style classes" for overlaying maps. We overlapped multiple maps of identical size by putting them in the same position on the Web page. To implement that, we used CSS Positioning properties to define the position of an HTML element, as follows:

```
<style type="text/css">
.background {position: relative; top: 0px; left: 0px;}
.layer {position: absolute; top: 0px; left: 0px;}
</style>
```

We defined the ".background" class with a position relative to the left and top of its parent element. The map contained in the HTML element using the ".background" class will be the container for all other maps contained in the HTML elements using the ".layer" class. That is, the first map requested becomes the background and the following maps are overlapped sequentially on the background map (see Figure 4.22).

When implementing *mapViewer* page, the style classes were applied to the HTML ⟨div⟩ tag. The "div.background" tag was nested in a table cell with a size equivalent to the size of the map to be requested. The first map requested was put in the

```
<td width="600" height="300" valign="top">
 <form method="POST" action="FeatureHandler">
  <div class="background">
   <input border=0 src="map1 URL" name="map" type="image">
    <div class="layer">
     <input border=0 src="map2 URL" name="map" type="image">
    </div>
    <div class="layer">
     <input border=0 src="map3 URL" name="map" type="image">
    </div>
    ......
  </div>
 </form>
</td>
```

Figure 4.22: The map is overlapped using the styles defined in CSS

"div.background" tag and displayed at the bottom. All maps that followed were put in the "div.layer" tags nesting within the "div.background" tag and displayed at the top.

4.4 Summary

In this Chapter, we described the building of a local Web mapping system and the implementation of a generic OGC WMS client.

Open source software was used locally to set up the entire system, including the Web server, JSP/Servlet container, OGC WMS server, and database.

The OGC WMS client was implemented in terms of the functionality and modules identified in Chapter 3. Each module was composed of Model, View, Controller components. The views were implemented three times, one for each approach under investigation. The controllers were implemented depending on the views, the approaches, and the functionality. Similarly, the implementation of models depended on the functionality but it was independent of the approaches we used.

Finally, the implementation of three generic operations were introduced. They were inserting data into tables with XMLC, dealing with XML/GML documents using DOM, and using CSS to support multiple maps overlapping.

Chapter 5

Evaluation

In this Chapter, the three approaches—JSP with embedded Java, JSP with JSTL tags, and XMLC—are evaluated against a set of criteria from the perspective of architecture, development, and maintenance. Each approach will be rated with one to three stars ("*", "**" or "***") for each criterion. The more stars, the better or easier. First of all, the evaluation criteria are introduced.

5.1 Evaluation Criteria

Many vendors have provided frameworks for Web application development with JSP and XMLC. The typical ones are Apache Struts [3] and Enhydra Barracuda [11]. Lutris Technologies has made a detailed comparison between Barracuda and Struts [20]. The Jcorporate also evaluated its product—Expresso and many other open source application frameworks including Struts and Enhydra [15]. Although those frameworks were not used in this research to implement the OGC WMS client, some criteria they proposed can still be applied here to evaluate the JSP and XMLC. Below is a set of criteria we developed to evaluate the three approaches.

Separating Content from Presentation

In a dynamic Web page, the content typically indicates those raw data generated in the business logic, which is usually implemented in programming language. In a J2EE Web application, the default programming language is Java. The presentation allows

those data to be formatted and dynamically presented with Web technologies like JSP and XMLC. Two measures were used to assess how well the content and presentation are separated:

1. Separating Markup from Code: Separating markup from code means a Web page document contains only markup elements without any Java code. In other words, HTML do not contain Java, and Java do not contain HTML.

2. Separating Presentation-oriented Tasks from Data-oriented Ones: Presentation-oriented tasks define how to present, and data-oriented ones generate what to present. Separating those two kinds of tasks means the content to be presented should not be generated in the presentation.

Ease of Development

Seven measures are used to evaluate how easy to develop using the three approaches. These issues are essential and frequently happened in dynamic Web page development, not only for the OGC WMS client, but all other Java-based Web applications.

1. Method Calling: In a Web application with MVC architecture, the data presented on the Web page (View) are usually got from JavaBeans and/or other Java objects (Model). The Model provides a number of public methods that process the data as required and return them; these methods can be called from within the View or by other objects within the application scope.

2. Data Type Conversion: Data is often converted between different types according to the context in which this data is to be used. A data been accessed in the View may not always exactly match the type expected. In the context of Model, the data is usually generated in their initial types and/or encapsulated in some collection objects. In the context of View, the type of the data retrieved from the Model needs to be converted at times for the purpose of display.

3. Conditional Web Page Output: The output of a Web page is often conditional depending on the value of dynamic data. As a result, the same page could display various content every time it is invoked.

4. Inserting Content into Tables: Often, a collection of data is displayed in a table, one data value in each column of a table row. In simple cases, each table row has identical structure except the value of the data. While in some complicate situations, the structure may be distinct among rows in a table as the elements to be displayed in each row, even each cell of a row, are various depending on the data presented.

5. Dealing with XML/GML: XML is now widely used in the Web for exchanging information. When developing a dynamic Web page, we often need to display some content extracted from an XML document. In the case of the OGC WMS client, the WMS server capabilities are provided in the form of an XML document, and the feature information are also possibly presented using GML.

6. Overlapping Multiple Maps: CSS is a client-side technology usually used to define how HTML elements are displayed. In our implementation, we defined HTML structures in the CSS for multiple maps overlapping (see Section 4.3.3). We want to see how easily the three approaches make use of styled HTML elements to overlap maps on the Web page.

7. Error Localisation: To reduce the number of delivered errors, debugging is very important in software development. Errors have to be located and corrected. The localisation of the errors can be very time consuming because of the confusing error message. How easily can errors be diagnosed using the three approaches will be discussed here.

Ease of change

After completing the development of a Web application, we usually need to do some revisions or updating. We will discuss the maintenance of a dynamic Web page from three aspects.

1. Changing the Page Appearance: Changing a Web page's appearance indicates the modification or update of static HTML elements. We may add a image, change the background, or use another HTML structure to present the same data.

2. Changing the Content: The change of content may be caused by the revision of the back-end Java components or the change of page appearance. The code used to control the Web page may be modified, and the data being displayed could be reorganised.

3. Rebuilding Updated Pages: Once completing the change, the updated page needs to be rebuilt in the Web server.

We have concluded the description of the criteria in this Section. From the next Section, we will describe the evaluation in terms of these criteria.

5.2 Separating Content from Presentation

As introduced previously, the separation of content from presentation is discussed in terms of: separating markup from code and separating presentation-oriented tasks from data-oriented ones.

5.2.1 Separating Markup from Code

JSP with Embedded Java

As introduced in Section 2.2, JSP allows Java to be inserted into the Web page. When using JSP with embedded Java, the markup (HTML) and code (Java programming) are not mixed in the Java class like Servlet, instead, they are mixed together within the JSP file. The HTML is responsible for presenting the page template, which is static; and the embedded Java is responsible for controlling the dynamic content presentation. For example, in *mapViewer.jsp* (see Figure 4.9 on Page 42), we displayed titles of selected layers in a multiple selection list using the code as shown in Figure 5.1. Although we can write our own custom tags or use JSTL tags to take the place of the Java code, that is not mandatory. Due to the explicit mixture of HTML and Java, we mark it with "*".

JSP with JSTL Tags

JSTL uses HTML-like tags and Expression Language to manipulate the dynamic content in the JSP file, ideally, Java code will occur less frequently. As a result, the JSP

```
<select name="selectedLayers" size="10" multiple>
 <% String[] listLegends = legdInfo.getListLegendStr();
    for(int i=0; i<lgNum; i++) {
        String legend = listLegends[i];
 %>
 <option> <%= legend %>
 <% } %>
</select>
```

Figure 5.1: Sample code from *mapViewer.jsp*, displaying selected layer titles in a selection list using JSP with embedded Java

```
<select name="selectedLayers" size="10" multiple>
 <c:forEach var="legend" items="${sessionScope.legdInfo.listLegendStr}">
  <option> <c:out value="${legend}"/>
 </c:forEach>
</select>
```

Figure 5.2: Sample code from *mapViewer.jsp*, displaying selected layer titles in a selection list using JSP with JSTL tags

file may contain only markup such as HTML elements, JSTL tags, and JSP tags (no scriplets). For example, in *mapViewer.jsp* that is implemented using JSP with JSTL tags, the multiple selection list was created using the code shown in Figure 5.2.

Note that, although almost all tasks that are needed to create a JSP file can be achieved using JSTL tags, Java may still occasionally be used in a JSP file to implement functionality not attainable from JSTL tags. Therefore, we mark it with "**".

XMLC

When using XMLC, the HTML is used to create the template Web page that has no embedded Java. The template page is compiled into a Java class, which is manipulated using DOM APIs in another Java class (manipulation class). Therefore, the HTML and Java are perfectly separated. No Java is included in the template page, and no HTML will appear in the manipulation class. For example, to generate the multiple selection list in the *mapViewer* page, a sample selection list with a sample option element is created in the template page *mapViewer.html* (see Figure 5.3). The template page is

```
<select id="legendSelection" name="selectedLayers" size="10" multiple>
 <option id="option">legend layer
</select>
```

Figure 5.3: Sample code from template page *mapViewer.html*, creating a sample multiple selection list

```
public class MapViewerMan extends HttpServlet{
  ......
  MapViewerHtml mapViewer = new MapViewerHtml();
  HTMLSelectElement selectionList = mapViewer.getElementLegendSelection();
  HTMLOptionElement option = mapViewer.getElementOption();
  selectionList.removeChild(option);
  for(int i=0; i<legendNum; i++) {
      .....
      mapViewer.setTextOption(listLegends[i]);
      selectionList.appendChild(option.cloneNode(true));
  } ......
}
```

Figure 5.4: Sample code from the manipulation class *MapViewerMan*, displaying layer titles in a selection list using XMLC

compiled into a Java class *MapViewerHtml*, which is manipulated in the manipulation class *MapViewerMan* (see Figure 5.4) where the titles of selected layers are inserted into the sample selection list using the algorithm we introduced in Figure 4.14 on Page 47. We award XMLC "***".

5.2.2 Separating Presentation-oriented Tasks from Data-oriented Ones

JSP with Embedded Java

As introduced in Section 3.3.1, "Model 2 JSP" is such an architecture conforming to MVC, which enforces the separation of presentation logic and business logic. The View handles the presentation logic doing presentation-oriented tasks, and the Model deals with business logic doing data-oriented ones; the Controller connects them to-

gether. The Java fragments embedded in JSP files are purely used to make the page dynamic and aid presentation. For example, the code shown in Figure 5.5 uses a loop structure to display the title and the number of features of each selected map layer in *featureSummary.jsp*. The data presented in the page are managed in the "bean" class *LegendInformation* and a *HashMap* with session context.

However, using "Model 2 JSP" is not mandatory. Because JSP allows Java code to be inserted into a page, some data-oriented tasks are quite easy to be included in. Even worse, the controller logic such as dealing with the HTTP request may also be mixed in. Therefore, we give JSP with embedded Java "**".

JSP with JSTL Tags

"Model 2 JSP" is also applicable to the JSP with JSTL tags. We can leave all business logic to be achieved in the Model, and just use JSTL "core" tags to control the presentation. Thus, the presentation-oriented tasks and data-oriented tasks are separated.

As discussed previously, however, using "Model 2 JSP" is not mandatory. JSTL provides other functional tags (see Section 2.3) besides the "core" tags that let the developer able to do many simple data-oriented tasks, such as XML document accessing and data formatting, within the JSP document. For example, we tried requesting and parsing the GML document within *featureSummary.jsp* using JSTL "xml" tags as well as "core" tags (see Figure 5.6). As a result, the Model are mixed into the View. Therefore, we mark it with "**".

XMLC

The XMLC approach makes the application architecture easier to be conformed to the MVC paradigm. The template pages and the page classes representing each template page can be regarded as the Views defining how the page will be displayed. The manipulation classes are the Controllers of each template page, which interact with the Model and manipulate the template page to determine what content will be displayed. The presentation-oriented tasks and data-oriented ones are handled separately. For example, to create the *featureSummary* page, we developed a template page *featureSummary.html* that defines how the feature summary should be presented and compiled it into a page class *FeatureSummaryHtml*. The data-oriented tasks are done

```
<jsp:useBean id="legdInfo" scope="session" class="wms.LegendInformation"/>
<jsp:useBean id="allLegendFeature" scope="session"
             class="java.util.HashMap"/>
......
<table border="0" width="100%">
......
<% for(int i=0; i<legdInfo.getLegendNum(); i++) {
       int feaNum = 0;
       String legdTitle = legdInfo.getListLegendStr()[i]; %>
 <tr>
  <td width="30%" align="left">
   <font face="Arial Narrow" color="#008080"><b><%= legdTitle %>
    </b></font></td>
  <td width="30%" align="left"><font face="Arial Narrow"><b>
<% if(allLegendFeature.get(legdTitle) == null)
     out.println("GML is not supported");
   else {
     feaNum =
        ((LegendFeature)allLegendFeature.get(legdTitle)).getFeatureNum();
     out.println(feaNum+" Feature(s)");
   } %>
   </b></font></td>
  <td align="left">
<% if(feaNum > 0) { %>
   <a href="featureInfo.jsp?legend=<%= legdTitle %>&fid=1" target="new">
    <img src="image/info.gif" border="0" height="18" width="18"></a>
<% } %>
  </td>        ......
 </tr>
<% } //end for loop %>
</table>
```

Figure 5.5: Sample code from *featureSummary.jsp*, implementing using JSP with embedded Java

```
<table border="0" width="100%">
......
<c:forEach var="entry" items="${sessionScope.legdInfo.legendBuf}"
           varStatus="status">
 <c:set var="legendTitle" value="${entry.key}"/>
 <c:set var="legend" value="${entry.value}"/>
 <c:set var="request"
value="${legend.urlPrefix}${sessionScope.reqMange.featureInfoRequestParams}
  &LAYERS=${legend.layerStr}&STYLES=${legend.styleStr}
  &QUERY_LAYERS=${legend.name}&X=${param['map.x']}&Y=${param['map.y']}"/>
 <c:import url="${request}" var="gml"/>
 <x:parse xml="${gml}" var="doc"/>
 <tr>
  <td width="25%" align="left"><font face="Arial Narrow" color="#008080">
   <b><c:out value="${legendTitle}"/></b></font></td>
  <td width="25%" align="left"><font face="Arial Narrow">
   <b><c:out value="${legend.service}"/></b></font></td>
  <td width="15%" align="left"><font face="Arial Narrow"><b>
   <x:choose><x:when select="name($doc/*)='ServiceExceptionReport'">
     <c:out value="GML is not supported"/>
   </x:when><x:otherwise>
     <x:set var="features" select="$doc/*/*/*"/>
     <c:set target="${feaMap}" property="${legendTitle}"
            value="${features}"/>
     <x:out select="count($doc/*/*/*)"/> Feature(s)
   </x:otherwise>
   </x:choose>
  </b></font></td>
   ......
 </tr>
</c:forEach>
</table>
```

Figure 5.6: Sample code from *featureInfo.jsp*, accessing GML file using JSTL tags

in a DAO that accesses the GML documents and stores the data in JavaBeans and other objects. A manipulation class *FeatureMan* was developed to use the DAO and control the page class by inserting data got from those data objects (see Section 4.2.4). Therefore "***" is rewarded to XMLC.

5.3 Ease of Development

In this section, seven issues are discussed to evaluate the three approaches to developing dynamic Web pages: method calling; data type conversion; inserting content into tables; conditional Web page output; dealing with XML/GML; overlapping multiple maps; and error localisation. We assume that the Web page developer has the knowledge of HTML, JSP, JSTL, and Java programming.

5.3.1 Method Calling

JSP with Embedded Java

In a JSP file with embedded Java, we usually use simple Java statements to get required data by calling JavaBeans properties and/or specific methods performed in the JavaBeans and other back-end Java components. For example, to request maps for each layer in *mapViewer.jsp*, we get GetMap request strings from *reqMange*—an instance of the class *RequestManager* (see Figure 5.7). As each map represents a specific layer, the title of the layer is needed to be provided as a parameter when getting the GetMap request string by calling the method *getMapRequest()* implemented in the class *RequestManager* (See Figure 5.8). The titles of those layers to be displayed on the map are collected in a string array got from *legdInfo*—an instance of another class *LegendInformation* by calling the method *getMapLegendStr()*.

Obviously, the *MapRequest* is not a JavaBean property, because it is not follow the JavaBeans property conventions introduced in Section 2.2. While the *MapLegengStr* can be regarded as a read only JavaBean property (without set method provided) and accessed using the JSP action tag alternatively:

```
<jsp:getProperty name="legdInfo" property="MapLegendStr" />
```

From the view of Java programming, however, calling JavaBean properties and normal Java methods have no serious difference. Therefore, we mark it with "***".

```
<td width="<%= spaContx.getWidth() %>" height="<%= spaContx.getHeight() %>"
    valign="top">
 <div class="background">
<% int lgNum = legdInfo.getLegendNum();
   String[] mapLegends = legdInfo.getMapLegendStr();
   for(int i=0; i<lgNum; i++) {
       String mapRequest = reqMange.getMapRequest(mapLegends[i]);
       if(i>0) { %>
  <div class="layer">
<%     } %>
   <input border=0 src="<%= mapRequest %>" name="map" type="image">
<% if(i>0) {  %>
  </div>
<% }   } %>
 </div>
</td>
```

Figure 5.7: Sample code from *mapViewer.jsp*, overlapping maps using JSP with embedded Java

```
public class LegendInformation {
  String mapLegendStr;
  ......
  public String[] getMapLegendStr () {...}
  ......
}
public class RequestManager {
  String mapRequest;
  ......
  public String getMapRequest (String layerTitle) {...}
  ......
}
```

Figure 5.8: The methods *getMapRequest()* and *getMapLegendStr()* implemented in classes *RequestManager* and *LegendInformation*

JSP with JSTL Tags

JSTL provides a convenient way for accessing JavaBean properties. As introduced in Section 2.3, JSTL supports an Expression Language(EL). When using EL to access the JavaBeans properties, we only need to simply put the property name after the convention of Java identifiers. We still use the example shown in Figure 5.8. The string array containing a set of layer titles can be accessed conveniently using the JSTL expression:

```
"${legdInfo.mapLegendStr}"
```

because the "mapLegendStr" is a JavaBean property with a get method *getMapLegendStr()* returning a string array (see Figure 5.9).

```
<td width='<c:out value="${sessionScope.spaContx.width}"/>'
    height='<c:out value="${sessionScope.spaContx.height}"/>'
    valign="top">
 <div class="background">
  <c:forEach var="legdTitle" items="${legdInfo.mapLegendStr}"
             varStatus="status">
   <c:set target="${sessionScope.reqMange}" property="activeLegendTitle"
          value="${legdTitle}"/>
   <c:if test="${status.first==false}">
    <div class="layer">
   </c:if>
   <input border=0 name="map" type="image"
          src='<c:out value="${sessionScope.reqMange.mapRequest}"/>' >
   <c:if test="${status.first==false}">
    </div>
   </c:if>
  </c:forEach>
 </div>
</td>
```

Figure 5.9: Sample code from *mapViewer.jsp*, overlapping maps using JSP with JSTL tags

```
public class RequestManager {
  String activeLegendTitle;
  String mapRequest;
  ......
  public Void setActiveLegendTitle (String activeLegendTitle) {...}
  public String getActiveLegendTitle () {...}
  public String getMapRequest () {...}
  ......
}
```

Figure 5.10: The methods implemented in the class *RequestManager* are conformed to the JavaBeans property conventions

The methods that do not conform to the JavaBeans property conventions, perhaps because a method has input arguments or the method name is not start with "get", are inaccessible from JSTL EL, but would be accessible using Java. For example, the method *getMapRequest()* in Figure 5.8 cannot be accessed using JSTL EL due to the input argument "layerTitle". In this situation, we must define setter/getter methods that conform to the JavaBeans property conventions to make it accessible from the JSTL EL. We defined a JavaBean property called "activeLegendTitle" that can be set using the setter method *setActiveLegendTitle()* and then be accessed from within the accessor method *getMapRequest()* using *getActiveLegendTitle()* (see Figure 5.10). Thus, the "mapRequest" is a JavaBean property that can be accessed from the JSP using JSTL EL (See Figure 5.9).

In a word, when implementing using JSP with JSTL tags, the methods implemented in the Java classes must be carefully defined to conform to the JavaBeans property conventions if they are to be accessed from within JSP files. As a result, some extra properties might be defined. Therefore, we mark the JSP with JSTL tags with "**".

XMLC

Since the Web page is manipulated in a Servlet class when using XMLC, there is no obstacle on the way to access any kind of methods implemented in other Java objects. For example, in *MapViewerMan*, which is a manipulation class create the *mapViewer* page, the "mapLegendStr" and the "mapRequest" are retrieved using the code shown

```
int lgNum = this.legdInfo.getLegendNum();
String[] mapLegends = this.legdInfo.getMapLegendStr();
for(int i=0; i<lgNum; i++) {
    String mapRequest = this.reqMange.getMapRequest(mapLegends[i]);
    ......
}
```

Figure 5.11: Sample code from the manipulation class *MapViewerMan*, demonstrating the method calling using XMLC

in Figure 5.11. We reward XMLC "***" for this criterion.

5.3.2 Data Type Conversion

JSP with Embedded Java

When using Java in a JSP document to convert one type of data into another, we usually follow the rules of "casting" ([31] Chapter 5.5). That is, if we want to convert type A data into type B data (cast type A to type B), put the type B name in parentheses in front of the type A data. See the example shown in Figure 5.5 on Page 64, the number of features was retrieved using the statement below:

```
feaNum = ((LegendFeature)allLegendFeature.get(legdTitle)).getFeatureNum();
```

The "allLegendFeature" is a *HashMap* object containing a collection of *LegendFeature* objects. In Java, the collection classes like *HashMap* and *Vector* deal with its elements as type *Object*, which is the root class of all classes in Java. We must restore each *Object* to its specific type before using it. Therefore, each *Object* got from "allLegendFeature" must be cast to *LegendFeature*.

The other kind of data type conversion that always happens is the conversion between *String* and those primitive types like *int* and *double*. In this case, Java provides methods to realise the type conversion. For example, see the code below got from *featureInfo.jsp*, a parameter named "fid" indicating which feature is going to be display is got from the request. The type of the gained data is a *String*. We used the static method *Integer.parseInt()* to convert the *String* to an *int* value.

```
int fid = Integer.parseInt(request.getParameter("fid"));
```

```
<c:set var="feaNum"
       value="${sessionScope.allLegendFeature[legdTitle].featureNum}"/>
<c:out value="${feaNum} Feature(s)"/>
```

Figure 5.12: Sample code from *featureSummary.jsp*, showing automatic data type conversion using JSTL

Overall, we mark "**" to the JSP with embedded Java. Note that, this issue is more associated with the Java programming language than the JSP itself; it occurs in the JSP development because of the insertion of Java. The good news is that the problem is likely to be addressed by the new Generics feature of J2SE 1.5[1], which provides compile-time type safety for collections and eliminates the drudgery of casting.

JSP with JSTL Tags

The JSTL EL provides automatic data type conversion, which coerces the type of resulting data to the expected type. For example, the number of feature was retrieved using the code shown in Figure 5.12 when JSTL tags were used in *featureSummary.jsp*.

Each object got from "allLegendFeature" is automatically coerced to a *LegendFeature* object. Therefore, the page author does not need to worry about the specific type of the collection of objects to iterate over. Similarly, the conversion between *String* and primitive types is also automatic. Therefore, we reward JSP with JSTL tags "***".

XMLC

When using XMLC, the operations on the data are performed in the manipulation classes using Java. There is no difference from that implemented using JSP with embedded Java; therefore, we also mark it with "**". As mentioned when evaluating the JSP with embedded Java, this issue is more associated with the Java programming language. If the problem can be removed by the new version of J2SE, we would award XMLC "***".

[1]The beta of Java 2 Platform, Standard Edition 1.5 may be released later 2003

5.3.3 Conditional Web Page Output

JSP with Embedded Java

Conditional structures like "if" and "if-else" are usually used in a JSP file to determine which part should be displayed every time when the page is invoked. For example, in the *featureInfo* page (see Figure 4.12 on Page 44), two red arrows are used to let the user turn over the features forwards or backwards. If the feature displaying is the first one, the "previous" arrow will be removed; if the feature displaying is the last one, the "next" arrow will be removed. Moreover, if the feature sequence number that the user typed in the entry is out of the range, an error message will be shown instead of the table listing feature data, and both arrows will be removed. In *featureInfo.jsp*, an "if" statement is used to control which arrow will be shown, and an "if-else" is adopted to determine whether the error message, or the table listing feature data, will be displayed for the user (see Figure 5.13). The structure and Java statement are clear and simple; we mark it with "***"

JSP with JSTL Tags

In a JSP file with JSTL tags, the ⟨c:if⟩ tag, corresponding to "if" statement of Java, can be used to do conditional control. In addition, the "if-else" structure of Java can also be achieved using ⟨c:choose⟩/⟨c:when⟩/⟨c:otherwise⟩ tags. See the code shown in Figure 5.14, which generates the same part of the *featureInfo* page as the code shown in Figure 5.13 does. These JSTL tags are meaningful and easy to use; we also reward the JSP with JSTL tags "***".

XMLC

When using XMLC, which part of the page should be displayed is manipulated in manipulation classes. We need to put all possibilities in template HTML pages, because the manipulate class does not know what should be displayed until it is executed at run time. See the HTML code shown in Figure 5.15, which is extracted from the template page *featureInfo.html*. It displays three sections of the page: a table with a single row containing two arrows and others content, an error message, and a table for displaying feature data. Each element that might be removed is marked with unique

```
<table border="0" width="100%">
 <tr>
  <td valign="top" align="left" width="25%">
   <font face="Arial" color="#008080"><%= feaNum %> feature(s)</font>
  </td>
  <td valign="top" width="11%">
<% if(fid>1 && fid<=feaNum) { %>
   <a href="featureInfo.jsp?legend=<%= legendTitle %>&fid=<%= fid-1 %>">
    <img SRC="image/previous.gif" BORDER=0></a>
<% } %>
  </td>
  <td valign="top" width="12%">
<% if(fid>=1 && fid<feaNum) { %>
   <a href="featureInfo.jsp?legend=<%= legendTitle %>&fid=<%= fid+1 %>">
    <img SRC="image/next.gif" BORDER=0></a>
<% } %>
  </td>
  ......
 </tr>
</table>
<% if(fid>feaNum || fid<0) { %>
<hr>
<font face="Arial" color="#000080">
 The feature you selected is not in the list!</font>
<% } else { %>
<table border="2" frame="hsides" width="100%">
......
</table>
<% } %>
```

Figure 5.13: Sample code from *featureInfo.jsp*, demonstrating the conditional output using JSP with embedded Java.

"id" attribute.

In the manipulation class *FeatureInfoMan*, some parts in the template page are

```
<table border="0" width="100%">
 <tr>
  <td valign="top" align="left" width="25%">
   <font face="Arial" color="#008080">
    <c:out value="${feaNum} feature(s)"/></font>
  </td>
  <td valign="top" width="11%">
   <c:if test="${fid>1 and fid<=feaNum}">
    <a href='featureInfo.jsp?legend=<c:out value="${legendTitle}"/>
&fid=<c:out value="${fid-1}"/>'><img SRC="image/previous.gif" BORDER=0></a>
   </c:if>
  </td>
  <td valign="top" width="12%">
   <c:if test="${fid>=1 and fid<feaNum}">
    <a href='featureInfo.jsp?legend=<c:out value="${legendTitle}"/>
&fid=<c:out value="${fid+1}"/>'><img SRC="image/next.gif" BORDER=0></a>
   </c:if>
  </td>
  ......
  </tr>
</table>
<c:choose><c:when test="${fid>feaNum or fid<0}">
<hr>
<font face="Arial" color="#000080">
 The feature you selected is not in the list!</font>
</c:when><c:otherwise>
<table border="2" frame="hsides" width="100%">
......
</table>
</c:otherwise>
</c:choose>
```

Figure 5.14: Sample code from *featureInfo.jsp*, demonstrating the conditional output using JSP with JSTL tags.

```
<table border="0" width="100%">
 <tr>
  <td valign="top" align="left" width="25%">
   <font face="Arial" color="#008080">
    <span id="num">N</span> feature(s)</font></td>
  <td valign="top" width="11%">
   <a id="prev" href="FeatureInfoMan?lid&fid">
    <img SRC="image/previous.gif" BORDER=0></a></td>
  <td valign="top" width="12%">
   <a id="next" href="FeatureInfoMan?lid&fid">
    <img SRC="image/next.gif" BORDER=0></a></td>
  ......
  </tr>
</table>
<div id="error"><hr>
 <font face="Arial" color="#000080">
  The feature you selected is not in the list!</font>
</div>
<table id="report" border="2" frame="hsides" width="100%">
......
</table>
```

Figure 5.15: Sample code from template page *featureInfo.html*

kept and some are removed depending on the value of the variable named "fid", which is the sequence number of the feature the user requested (see Figure 5.16). When we remove an HTML element, its all child elements are removed together with it. That makes mistakes easy to be made when implementing, although the programming is not difficult for a Java developer. Therefore, we mark the XMLC with "**".

5.3.4 Inserting Content into Tables

JSP with Embedded Java

As introduced in criteria description in Section 5.1, a table being created to display data could be simple or complex. We are using same examples described in Section 4.3.1 to

```
FeatureInfoHtml featureInfo = new FeatureInfoHtml();
HTMLAnchorElement prev = featureInfo.getElementPrev();
HTMLAnchorElement next = featureInfo.getElementNext();
HTMLDivElement error = featureInfo.getElementError();
HTMLTableElement report = featureInfo.getElementReport();
......
if(fid>1 && fid<=feaNum) {
   int newFid = fid - 1;
   prev.setHref("FeatureInfoMan?lid=" + lid + "&fid=" + newFid);
}
else
   prev.getParentNode().removeChild(prev);
if(fid>=1 && fid<feaNum) {
   int newFid = fid + 1;
   next.setHref("FeatureInfoMan?lid=" + lid + "&fid=" + newFid);
}
else
   next.getParentNode().removeChild(next);
if(fid<1 || fid>feaNum)
   report.getParentNode().removeChild(report);
else {
   error.getParentNode().removeChild(error);
......
}
```

Figure 5.16: Sample code from the manipulation class *FeatureInfoMan*, demonstrating the conditional output using XMLC.

discuss inserting content into tables using JSP with embedded Java.

The first example is the table created in the *layerList* page (see Figure 4.5 on Page 38), displaying a list of layer titles. Iteration structures such as "for" or "while" are usually used to insert rows of data in such a simple table. In *layerList.jsp*, a list of checkboxes and layer titles are inserted in a table with two columns using "for" structure, each title per row with same display style (see Figure 5.17).

In the case that each row or cell in a table displays different components, condi-

```
<table border="0" width="100%">
......
<% for(int i=0; i<server.getLayerListSize(); i++) {
      layerData layer = server.getLayer(i);
%>
 <tr>
  <td width="1%" height="21" valign="top" bgcolor="#efedde">
   <input type="checkbox" name="checkedLayers" value="<%= i %>">
  </td>
  <td bgcolor="#efedde"><font color="#000080">
   <%= layer.getTitle() %></font>
  </td>
 </tr>
<% } %>
</table>
```

Figure 5.17: Sample code from *layerList.jsp*, displaying a list of layer titles in a table using JSP with embedded Java.

tional statements such as "if" or "if-else" are usually used within iteration structure to determine what, or if anything, is to be displayed. The example we are using is the table created in the *featureSummary* page (see Figure 4.9 on Page 42), displaying the feature summary. In *featureSummary.jsp*, we used "if" statement in the "for" structure to decide whether a small "i" icon linking to *featureInfo.jsp* should be added in the third column of each row. (see Figure 5.5 on Page 64).

From both examples discussed above, we can see that a table with a collection of data inserted can be easily created in the JSP file using simple Java iteration and conditional structures. Therefore, we award JSP with embedded Java "***".

JSP with JSTL Tags

The two tables created in the *layerList* page and the *featureSummary* page are still used to discuss inserting content into tables using JSP with JSTL tags.

JSTL ⟨c:forEach⟩ tag provides a iteration structure that can be easily used to iterate over a collection of objects and build a table, one row per loop. In *layerList.jsp*, the

```
<table border="0" width="100%">
 ......
 <c:forEach var="layer" items="${sessionScope.server.layerList}"
            varStatus="status">
  <tr>
   <td width="1%" height="21" valign="top" bgcolor="#efedde">
    <input type="checkbox" name="checkedLayers"
           value='<c:out value="${status.count-1}"/>'>
   </td>
   <td bgcolor="#efedde">
    <font color="#000080"><c:out value="${layer.title}"/></font>
   </td>
  </tr>
 </c:forEach>
</table>
```

Figure 5.18: Sample code from *layerList.jsp*, displaying a list of layer titles in a table using JSP with JSTL tags.

⟨c:forEach⟩ tag is used to retrieve each layer title from a collection object and display it in the second cell of the row being processed; a checkbox is put in the first cell of each row (see Figure 5.18).

In *featureSummary.jsp*, the conditional JSTL tag ⟨c:if⟩ is used within the ⟨c:forEach⟩ tag to determine the output of the small "i" icon (see Figure 5.19). It is also straightforward and easy to implement; therefore, we award the JSP with JSTL tags "***".

XMLC

Inserting content into tables using XMLC has been introduced in Section 4.3.1. Because creating the prototype table and manipulating it are performed in two separate files, inserting data into a table using XMLC is not straightforward, especially in the case that the components displayed in each row of a table are various, such as the table created in the *featureSummary* page. As each row been inserted in the table is built on the same sample row, the sample row has to be very carefully protected without any

```
<table border="0" width="100%">
......
 <c:forEach var="entry" items="${sessionScope.legdInfo.legendBuf}"
            varStatus="status">
  <c:set var="legdTitle" value="${entry.key}"/>
  <c:set var="legend" value="${entry.value}"/>
  <tr>
   <td width="30%" align="left">
    <font face="Arial Narrow" color="#008080"><b>
     <c:out value="${legdTitle}"/>
    </b></font></td>
   <td width="30%" align="left"><font face="Arial Narrow"><b>
    <c:choose>
    <c:when test="${empty sessionScope.allLegendFeature[legdTitle]}">
     <c:out value="GML is not supported"/>
    </c:when><c:otherwise>
     <c:set var="feaNum"
        value="${sessionScope.allLegendFeature[legdTitle].featureNum}"/>
     <c:out value="${feaNum} Feature(s)"/>
    </c:otherwise></c:choose>
    </b></font></td>
   <td align="left">
    <c:if test="${feaNum>0}">
     <a href='featureInfo.jsp?legend=<c:out value="${legdTitle}"/>&fid=1'
        target="new">
      <img src="image/info.gif" border="0" height="18" width="18"></a>
    </c:if>
   </td>
  </tr>
 </c:forEach>
</table>
```

Figure 5.19: Sample code from *featureSummary.jsp*, displaying feature summary in a table using JSP with JSTL tags.

alteration; otherwise mistakes may take place when printing out the generated Web page at run time. That makes the manipulation on the table complex and fallible. Therefore, we give XMLC "*" for this criterion.

5.3.5 Dealing with XML/GML

JSP with Embedded Java

To prevent the Web page from containing too much Java code and becoming too complex, the XML/GML document is usually handled in the Java classes. In our implementation, the WMS server's capabilities XML and GML documents are parsed and accessed using DOM in DAOs (see Section 4.3.2). The data extracted from the XML/GML document are stored in JavaBeans and other Java objects, which can be easily accessed from within the JSP document using simple Java statements. Therefore, whether it is easy to handle XML/GML depends on what XML parser is used.

Using DOM to handle XML/GML is not quite simple. Although DOM provides standard APIs for XML handling, some functions are not available and the developer need to implement their own APIs based on DOM. Therefore, we would only mark it with "*". Lots of optional tools are available besides DOM to handle XML, such as Simple API for XML (SAX), XML Path Language (XPath), Java API for XML Parsing (JAXP), and JDOM. These tools can be used individually or in combinaiton with each other to make XML handling easier.

JSP with JSTL Tags

When using JSP with JSTL tags, we have three options to consider.

1. The XML/GML document is entirely parsed and accessed in the JSP file, no more Java classes needed.

2. We can parse an XML/GML file into a DOM document in the Java class, and use JSTL "xml" tags, and probably "core" tags as well, to access the DOM and display the data in a JSP file.

3. Like JSP with embedded Java, the XML/GML document is totally handled in Java classes. We use JSTL "core" tags to get and display data in the JSP file.

As introduced in Section 2.3, the JSTL "xml" tags allow you to parse an XML file and access its data within a JSP file. We tried it to handle the GML document in *featureSummary.jsp* and *featureInfo.jsp*. See the sample code from *featureSummary.jsp* in Figure 5.6 on Page 65. The GML document was parsed using ⟨x:parse⟩ tag, and the result was set to a variable named "doc". The feature elements that are located in the third level of the hierarchy tree can be easily accessed using XPath expression as below:

```
<x:set var="features" select="$doc/*/*/*"/>
```

These feature elements were stored in an *HashMap* object, which were then accessed using other "xml".tags in *featureInfo.jsp*.

For the second option, we can handle the XML parsing in the Java classes such as controller Servlet or DAO, and access the parsed document in the JSP document to display required XML-based data. We did not try this option in our implementation. We still can see that, however, using this approach, the complex manipulation on the data that retrieved from the XML file can be performed in the Java classes, and in the JSP document the "xml" tags will be purely used for XML-based data presentation. Thus, the behaviors performed in the JSP document are still presentation-oriented without business logic included in.

The third option handles the XML document in the Java classes, which is same as that implemented using JSP with embedded Java. Thus, "xml" tags are not necessary to be used in the JSP file. We can use only "core" tags to call data from the JavaBeans or other Java objects and display them on the page.

The simplicity of handling XML/GML using JSP with JSTL tags is various in different cases. It is easy to parse and access XML file using JSTL "xml" tags and XPath expression, but the JSP file might become complex. Handling XML with DOM in the Java classes keeps the good architecture of the application, but accessing XML-based data could be harder. Therefore, we reward "**" to it.

XMLC

When using XMLC, the capabilities XML and GML documents were also handled using DOM in DAOs. There is no difference from using JSP with embedded Java. As mentioned when evaluating the JSP with embedded Java, it is more associated with the XML tools than the approach itself. Therefore, we would also mark "*" here.

5.3.6 Overlapping Multiple Maps

JSP with Embedded Java

The styles defined in CSS for multiple maps overlapping was introduced in Section 4.3.3. In a JSP file with Java embedded, the CSS is able to be defined and applied to the HTML elements just like in a general HTML page. The outlay of the elements been styled can then be easily controlled using simple Java code. In *mapViewer.jsp*, we simply used "for" and "if" structure to nest the styled elements (see Figure 5.7 on Page 67). The issues are quite related to those we discussed in the Sections about conditional output (Section 5.3.3) and inserting data into table (Section 5.3.4). Therefore, we reward the JSP with embedded Java "***".

JSP with JSTL Tags

The same idea as that used for JSP with embedded Java was used for JSP with JSTL tags. The styles were easily defined in *mapViewer.jsp*. The styled HTML elements "div.background" and "div.layer" were manipulated using the JSTL tags to help the overlapping of the map (see Figure 5.9 on Page 68). We also mark the JSP with JSTL tags with "***".

XMLC

When using XMLC, the CSS is defined and applied on the prototype HTML elements in the template page (see Figure 5.20). In the manipulation class, the styled elements are treated like other general HTML elements. The developer do not need to care about what the style of the element is. What they are interested in are those elements with "id" attribute (see Figure 5.20).

Defining styles for HTML elements in the template page is simple, while mistakes may be made when assembling those elements in the manipulation class, as we discussed in the Sections about conditional output (Section 5.3.3) and inserting content into tables (Section 5.3.4). Therefore, we mark the XMLC "**".

```
<td id="mapCell" width="mapWidth" height="mapHeight" valign="top">
 <div class="background" id="baseDiv">
  <input id="baseMap" border=0 src="mapRequest" name="map" type="image">
  <div class="layer" id="topDiv">
   <input id="topMap" border=0 src="mapRequest" name="map" type="image">
  </div>
 </div>
</td>
```

Figure 5.20: In the template page *mapViewer.html*, the styled prototype HTML elements are used

```
baseDiv.removeChild(topDiv);
......
for(int i=0; i<lgNum; i++) {
    String mapRequest = this.reqMange.getMapRequest(mapLegends[i]);
    if(i==0)
       baseMap.setSrc(mapRequest);
    if(i>0) {
       topMap.setSrc(mapRequest);
       baseDiv.appendChild(topDiv.cloneNode(true));
    }
......
}
```

Figure 5.21: Sample code from the class *MapViewerMan*, manipulating the map overlapping.

5.3.7 Error Localisation

JSP with Embedded Java

A JSP file is compiled when it is executed at run time. As introduced in Section 2.2, the JSP file is converted to a Servlet first, and the Servlet is then compiled and executed. The compilation may be failed because of syntax errors or semantics errors. When it is a syntax error, such as forgetting a semi-colon, missing a tag, or spelling mistake, Tomcat (Version 4.0.3) will generate an error message containing the line number where the error happened in the JSP file. The error is easy to be located. For example, if we

write a scriptlet instead of an expression by forgetting the equal sign like below:

```
<% layer.getTitle() %>
```

Tomcat generates the error:

```
An error occurred at line: 26 in the jsp file: /layerList.jsp
Generated servlet error:
work/localhost/jspTask2/layerList$jsp.java:110: Invalid type expression.
                 layer.getTitle()
                               ^
An error occurred between lines: 26 and 27 in the jsp file: /layerList.jsp
Generated servlet error:
work/localhost/jspTask2/layerList$jsp.java:113: Invalid declaration.
                out.write("\r\n");
                         ^
```

When it is a semantics error, such as the Array index is out of range, however, the error message only indicates the line number that matches to the error in the generated Servlet. Debugging such an error is hard and time consuming, because the developer has no way to locate the error in the JSP file, and often require to look at the generated Servlet code to diagnose what cause the error. Therefore, we award JSP with embedded Java "**".

JSP with JSTL Tags

JSP with JSTL Tags suffers from the the same problem as JSP with embedded Java. Besides that, the syntax error on the JSTL tags is also difficult to be located. For example, if we accidentally miss a slash in a JSTL tag like blow:

```
<c:out value="${layer.title}" >
```

Tomcat generates the error:

```
org.apache.jasper.compiler.ParseException: End of content reached while
more parsing required: tag nesting error?
        at org.apache.jasper.compiler.JspReader.popFile(JspReader.java:293)
......
```

without saying which line in the JSP file cause the error. We have to go through the entire JSP file from the first line to catch such a small bug. Therefore, we award JSP with JSTL tags "*".

XMLC

XMLC compiles the template page and the manipulation class prior to the run time. The HTML syntax error is diagnosed when converting the template page to a page class. The Java syntax error is located when compiling the manipulation class using *javac*. As for the semantics errors happened in the manipulation class, they are generated at run time when executing the manipulation class that is implemented as a Servlet. The error message contains the line number in the manipulation class of where the error happened. Therefore, locating the error is not hard. We award XMLC "***".

5.4 Ease of change

We discuss how easily to make changes from three aspects as introduced earlier in this Chapter: changing the page appearance; changing the content; and rebuilding updated pages.

5.4.1 Changing the Page Appearance

JSP with Embedded Java

Because of the embedding of Java and other JSP tags, the JSP document cannot be easily edited using typical HTML design tools such as FrontPage and DreamWave. For example, non-HTML or non-HTM files are refused to be loaded into the FrontPage 2000. That could be a problem for those page designers who do not know JSP tags and Java. They might transfer a JSP document to a HTML file by simply changing the file name's extension to make it editable using HTML editor, but consequently the data could be appeared at the position that are not expected at run time. That is because the Web designer may not know what the means of the Java code and JSP tags and where they should be put in the page being edited. In addition, the change on the HTML elements may also cause the change on the code that controls them. As

a result, the programmer usually ends up reinserting the Java code and JSP tags into the renewed page.

However, the work could be easier for a developer who knows HTML, JSP, as well as Java. For example, same Java code was used in *layerList.jsp*, no matter a table with checkboxes (see Figure 5.17 on Page 77) or a multiple selection list (see Figure 5.22) is used, to control the display of layer titles. If the Web designer knows what those Java code is used for, the revision should be quite easy. Overall, we give JSP with embedded Java "*".

```
<select name="checkedLayers" size="20" multiple>
<% for(int i=0; i<server.getLayerListSize(); i++) {
       layerData layer = server.getLayer(i);;
%>
   <option value="<%= i %>"><%= layer.getTitle() %>
<% } %>
```

Figure 5.22: Sample code from *layerList.jsp*, displaying a list of layer titles in a multiple selection list using JSP with embedded Java

JSP with JSTL Tags

As JSTL tags are also typically not recongised by HTML editors, we can only edit the JSP document in a text editor. However, the JSTL tags are intended to be easier to learn than Java for those Web designers, because they are HTML-like tags with similar syntax. Even that, the logics such as scoped variable setting, data looping, and condition evaluating might still make the designers feel complex to edit a Web page. If the page designer knows how to use JSTL and has some JSP knowledge, the page modification should not be hard. Therefore, we award the JSP with JSTL tags "**".

XMLC

When using XMLC, we change the Web page appearance by revising the template HTML page, which can be edited using any HTML editors. It is quite convenient for the Web designers. Only if the elements marked with "id" attribute are not removed or replaced by other HTML elements, the template page can be freely updated without

changing any Java code. For example, we use push buttons in the *mapViewer* page (Figure 4.9 on Page 42) for zoom in/out options, and these push buttons are then replaced by radio buttons (Figure 4.10 on Page 43). Because these HTML elements do not concern any dynamic content presentation and no "id" attribute is marked on them, their changes do not impact any Java code implemented in the manipulation class.

Another example, see the code in *layerList.html* (Figure 4.13 on Page 47), where a table with checkboxes are created. We can vary the background color of table rows, or even move the two buttons from below the table to on top of it, but still use the original manipulation class (Figure 4.15 on Page 47) without doing any change in it.

If there are any marked elements are removed or replaced by other elements, however, the class that manipulate the page has to be revised too. For example, we vary the *layerList* page by using a selection list instead of the checkboxes to list the layer titles. See the revised template HTML in *layerList.html* (Figure 5.23), the id attributes "check" and "layerTitle" are removed along with the ⟨input⟩ and ⟨font⟩ elements. Although the id attributes "list" and "option" are still there, they have already belonged to other HTML elements. As a result, the manipulation class *LayerListMan* has to be altered as well (see Figure 5.24).

In a word, changing the page appearance by modifying the template page is easy, but should be done carefully when altering the elements with "id" attribute, which may cause the change on its manipulation class. Therefore, we award XMLC "**".

5.4.2 Changing the Content

JSP with Embedded Java

Because the HTML and Java are coupled together in JSP file, some changes on back-end Java components may cause the modification of the code embedded in the JSP file correspondingly if those revised objects are refered to. In addition, the exhibition of HTML elements could also need to be adjusted along with the change of content. In an application with good architecture such as "Model 2 JSP", the change of content in the JSP file is usually very subtle. However, it is still a kind of hard work for designers who do not know Java. Therefore, we award JSP with Embedded Java "**".

```
<table border="0" width="100%">
 <tr bgcolor="#d0d0d0"><th id="serviceTitle" colspan="2" >service name</th>
 </tr>
 <tr>
  <td>
   <select id="list" name="checkedLayers" size="20" multiple>
    <option id="option" value="layerIndex">layer title
   </select>
  </td>
 </tr>
</table>
```

Figure 5.23: The prototype selection list created in the revised template page *layerList.html*

```
LayerListHtml layerList = new LayerListHtml();
HTMLSelectElement list = layerList.getElementList();
HTMLOptionElement option = layerList.getElementOption();

layerList.setTextServiceTitle(server.getService());
list.removeChild(option);
for(int i=0; i<server.getLayerListSize(); i++) {
    layer = server.getLayer(i);
    option.setValue(Integer.toString(i));
    layerList.setTextOption(layer.getTitle());
    list.appendChild(option.cloneNode(true));
}
```

Figure 5.24: The revised manipulation class *LayerListMan*

JSP with JSTL Tags

Like JSP with embedded Java, some changes in the back-end Java components will cause the small modification on the JSP file using JSTL tags if the application has good MVC architecture. However, if there are huge number of XML accessing or database reading operations being done in the JSP file with JSTL "xml" or "sql" tags, In this comparison, we assume that the Web page developer has the knowledges of HTML,

JSP, JSTL, and Java programmingrevising those part of codes for a page author are not so easy. Therefore, we award JSP with JSTL tags "**".

XMLC

When using XMLC, any dynamic content and server side modifications are done in the Java classes by programmer, which will not affect the HTML code in the template page. Therefore, the page designer can pay their attention on the page design and decoration without needing to know any programming things. We award XMLC "***".

5.4.3 Rebuilding Updated Pages

JSP with Embedded Java

Once the revision of the JSP file is completed and the revised page is saved and deployed, it will be compiled by the JSP container automatically at run time. No more work need to be done by the developer. We would mark it with "***".

JSP with JSTL Tags

No manual compilation is needed because the JSP container will compile the updated JSP file automatically at run time. It should also be reasonable to mark it with "***".

XMLC

Every time the template page is changed, even just a little revision, it has to be recompiled and redeployed. If alteration is needed in the manipulation class due to the change on the template page, the manipulation class should also be recompiled and redeployed manually[2]. Therefore, we only mark the XMLC with "*".

5.5 Summary

In this Chapter we evaluated the three approaches against a set of criteria. The evaluation results are summarised in Figure 5.25, where "***" means good or easy, "**" means unsure, depending on specific situation, and "*" means not good, or complex.

[2]all discussion is based on the use of XMLC without the Enhydra server

Criteria	Sub-criteria	JSP with embedded Java	JSP with JSTL Tags	XMLC
Separating content from presentation	Separating markup from code (5.2.1)	*	**	***
	Separating presentation-oriented tasks from data-oriented ones (5.2.2)	**	**	***
Easy to development	Method calling (5.3.1)	***	**	***
	Data type conversion (5.3.2)	**	***	**
	Conditional Web page output (5.3.3)	***	***	**
	Inserting content into tables (5.3.4)	***	***	*
	Dealing with XML/GML (5.3.5)	*	**	*
	Overlapping multiple maps(5.3.6)	***	***	**
	Error localisation(5.3.7)	**	*	***
Easy to change	Changing the page appearance (5.4.1)	**	**	**
	Changing the content (5.4.2)	**	**	***
	Rebuilding updated pages (5.4.3)	***	***	*

Figure 5.25: Evaluation summary

In everage, 2.25 stars are awarded to JSP with embedded Java, 2.33 stars are awarded to JSP with JSTL tags, and 2.17 stars are awarded to XMLC. JSP with JSTL tags gets the most stars among the three approaches. A detailed comparison is provided in Chapter 6.

Chapter 6

Comparison

In this Chapter, we compare the three approaches—JSP with embedded Java, JSP with JSTL tags, and XMLC—based on the evaluation discussed in Chapter 5. For the comparison, we have applied the identical criteria used in the evaluation (Section 5.1).

6.1 Separating Content from Presentation

Separating Markup from Code

JSP with embedded Java merges the HTML and the Java code into one file. The other two approaches—JSP with JSTL tags and XMLC—permit markup and code to be separated. In a JSP file with JSTL tags, Java code is not required because JSTL tags can be used for almost all the operations required to develop a JSP document. However, JSP does not forbid Java code to be used in a JSP file where JSTL or other custom tags are also used. Separating markup from code in a JSP file is practicable, but not obligable. Of the three approaches, XMLC is the preferred one as it physically separates the template HTML page from the code that manipulates the page into two files. Separating markup from code in a Web page implemented using XMLC is enforced.

Separating Presentation-oriented Tasks from Data-oriented Ones

The architecture of applications implemented using XMLC enforces the separation of presentation logic and business logic. Thus it conforms to the MVC pattern (Sec-

tion 3.3.1) of separating presentation-oriented and data-oriented tasks. By applying "Model 2 JSP" architecture, separation can also be achieved using the other two approaches. But the use of "Model 2 JSP" is neither mandatory nor universal as data can be generated using Java in JSP files and XML documents and database can also be accessed using JSTL tags. Choice of architecture is ultimately influenced by the application's requirements.

6.2 Ease of Development

In this comparison, we assume that the Web page developer has knowledge of HTML, JSP, JSTL, and Java programming

Method Calling

It is easy to use simple Java statements to perform method calling in a JSP file with Java embedded, or in manipulation classes if XMLC is used. JSTL EL provides a convenient way to get JavaBean properties but will only work on methods that conform to JavaBeans property conventions. In some situations using Java statements offers greater flexibility as only some of the Java components handling the business logic at the back-end may conform strictly to the JavaBeans property conventions or be JavaBeans themselves. However, data processing can be accomplished implicitly in normal Java methods and the results of the processing can be provided using *getter* methods. The *getter* methods can be called from within JSP files or any other Java object regardless of whether Java or JSTL tags have been used.

Data Type Conversion

Data type conversion is most easily achieved using the JSP with JSTL tags approach as JSTL EL can convert data types automatically in a JSP file. Therefore, the page author does not need to worry about the type of data retrieved from the Java objects or session context. As for the other two approaches, the developer has to treat data type conversion manually and carefully using Java, otherwise errors may occur when compiling or erroneous results may be produced. Fortunately, this problem is likely to be solved by the generic features of J2SE 1.5 and data type conversion will be also

convenient when using JSP with embedded Java and XMLC.

Conditional Web Page Output

Web page output can be easily controlled in the JSP file using some conditional structures, regardless of whether Java code or JSTL tags are used. The same is not true for XMLC, for which the process is neither simple nor straightforward. As we explained earlier, the template page and the control of the page output are performed separately in two files when using XMLC, which introduces a requirement for all achievable outcomes to be listed in the template page. These are used to manipulate the Web page output by including or removing related HTML elements in the manipulation class according to certain conditions. The intricacy of this process increases the probability of mistakes being made, especially when manipulating pages with complex functionality.

Inserting Content into Tables

Inserting content into tables using XMLC is a complex operation, especially when the components to be displayed in each row or cell are conditional on the data to be displayed along with them. This arises from the need to use complex algorithms to preserve the original HTML structure and to organise the output according to certain conditions. When working with manipulation classes, XMLC uses the parameters of the appropriate sample row in the template page to create each new row in a table. In contrast, the JSP approaches are much more straightforward as both the table creation and content insertion are performed in a single JSP file. This means that by using Java statements or JSTL tags, simple iteration and conditional structures can easily be used to perform these outputs.

Dealing with XML/GML

XML/GML can be handled in back-end Java components such as DAOs or Servlets. Accordingly, the XML processing tools being used are a more important influence on how easily XML/GML is dealt with than are the three approaches themselves. We used DOM, which provides standard APIs for accessing XML documents. However, the DOM APIs did not cover all our needs to access XML-based data.

Using JSTL "xml" tags and XPath to handle XML documents is convenient, as there is no need to write back-end Java components. The failing is that the JSP file may become very complex. Moreover, the architecture of the application could be yet more complex. This approach is suitable only for small applications that simply display XML-based data in a single page. In bigger applications it is preferable for Java components to perform the complex manipulation of XML-based data. For example, we preferred handling the capabilities XML file in a DAO because many types of capabilities data such as styles of a layer need not be displayed to the user. Furthermore, certain types of capabilities data are manipulated frequently; bounding box is one such example. Although GML can be handled in a JSP file, using the DAO ensures that the application business logic remains unchanged by different WMS servers, each of which may provide features with various schemes. The DAO is able to retain the integrity of the application business logic because it can implement a common interface for retrieving GML documents from different WMS servers.

Overlapping Multiple Maps

We defined styles for HTML elements using CSS position properties to support overlapping multiple maps. The overlapping of multiple maps is equally straightforward for all three approaches because defining styles using CSS, which is a client-side technology, is independent of server-side Web technologies. However, differences between the approaches become apparent when considering how styled HTML elements are used for dynamic Web page generation. As this issue relates to others about conditional output and the insertion of content into tables, both of the JSP approaches would be easier to use than XMLC.

Error Localisation

The errors in a JSP file cannot be diagnosed until we execute it at run time. For most syntax errors, the container will generate an error message containing the line number in the JSP file where the error happened. Therefore, locating the error is not hard. This is not the case when a syntax error occurs in a JSTL tag; here the error message will not contain a line number. Furthermore, when semantic errors occur in a JSP file the error message references the line number that matches the error in the

generated Servlet code, not the JSP file. This makes debugging much harder and far more time consuming. The converse is true of locating errors in XMLC as syntax errors can be diagnosed when compiling template pages and manipulation classes. Moreover, semantic errors are reported at run time and are easily located as the error message shows the line numbers corresponding to the errors in the manipulation class.

6.3 Ease of Change

Changing the Page Appearance

No HTML editor known to us has the ability to edit JSP documents and almost none support non-HTML elements. Consequently, editing JSP documents is a difficult task for Web designers who know HTML only and who work exclusively with HTML editors. Clearly, knowledge on the part of the Web designer of Java and JSTL makes the editing of JSP documents much easier but where this knowledge is absent, designers should consider choosing XMLC as this approach minimizes the degree of coupling between the HTML and the Java components. Designers can change the page appearance freely by editing the template page in any WYSIWYG[1] HTML editor. A shortcoming worth noting is that modifications to the template page might result in modifications to the manipulation class.

Changing the Content

When using XMLC, changes to Web page content are performed in the manipulation class and the template page is left unchanged. However, when using JSP approaches, Web page content is changed in the JSP document, which could affect the presentation of HTML elements. For this reason, Web page designers need to know Java or JSTL in addition to HTML. In an application with a well-designed MVC architecture, even big changes to the back-end Java code would result in only small content changes in JSP files. When using JSTL tags in the JSP file, the level of complexity and the number of changes required may increase significantly once the XML file or database has been accessed or changed. For these reason, XMLC is the preferred approach for making changes to Web page content.

[1] WYSIWYG stands for What You See Is What You Get

Rebuilding Updated Pages

The JSP container performs the compilation of a JSP file automatically at run time. For this reason, it takes a long time to load a new or updated JSP file for the first time. When using XMLC, it is possible to rebuild pages frequently during development but it is likely that the manipulation class will need to be manually recompiled along with the template page every time a Web page is updated. In this instance the two JSP approaches are preferable as they do not require frequent manual recompilation.

Chapter 7

Conclusions

In this thesis we have evaluated and compared three approaches to developing dynamic Web pages for a J2EE Web application. These approaches are JSP with embedded Java, JSP with JSTL tags, and XMLC. We used each of them to develop a client that interacted with servers implementing the OGC WMS specification. We identified functionality that was representative of a generic OGC WMS client and divided it into three levels with the complexity increasing from level one to level three: a simple prototype interacting with a single WMS server; that enables multiple requests; and that interacts with multiple WMS servers. The evaluation of the three approaches was based on the experience obtained through the implementation of this functionality.

We evaluated the approaches from three perspectives: architecture; development; and maintenance. During the implementation, each approach revealed advantages and disadvantages in various area.

Using JSP with embedded Java is a traditional approach to implement JSP files. Its advantage is that if the page author has the ability to use both HTML and Java, the development and update of the Web page will be straightforward. Because the HTML and Java code share a single file, the development process is simple. The disadvantage is that the mixture of HTML and Java provides an opportunity for the use of a bad application architecture. In addition, the development and maintenance of a JSP file will be hard for a Web designer who knows only HTML. Furthermore, page appearance changes might call for frequent changes to Java code, and vice versa.

The JSP with JSTL tags approach is a relatively new solution to implementing JSP files. Its advantage is that using JSTL tags in JSP files is straightforward. The

most important improvement introduced by using JSTL tags in JSP files is that the tags can replace Java code, which means that markup and Java can be separated. The disadvantage is that business logic can also be included in JSP files by using functional JSTL tags such as "xml" or "sql" tags. The presence of these tags could increase the complexity of JSP files and thereby make it more difficult to update Web pages. We also noticed that in instances where they were to be called from within a JSP file, there was a requirement for the methods implemented in Java components to conform to JavaBeans property conventions. This was because JSTL EL can access only JavaBeans properties and not normal Java methods.

The application implemented using XMLC closely conforms to the MVC architecture. This is the biggest advantage of XMLC. The presentation logic and business logic are clearly separate; the HTML is also totally isolated from Java. Thus the roles performed by page designers and Java programmers are distinct. Consequently updating an application becomes easier as changes to the user interface and the back-end Java component are less likely to affect each other. The disadvantage is that decoupling the HTML template and the manipulation code results in a far more complex development process compared to the process that would be followed with JSP. Furthermore, the separation of the HTML template from the manipulation code also requires that additional Java code be implemented 'behind the scenes' to manipulate the page output.

The Java programming language and XML parsers such as DOM are always used along with JSP and XMLC to implement Web applications. The presence of these elements have the effect of making Web page development more or less complex.

Our research shows that "Model 2 JSP" architecture should be implemented when using the JSP with embedded Java approach as it conforms to the MVC paradigm by separating presentation logic from business logic in the application. JSP with embedded Java is viable in situations where the page author can program with Java and where Web page decoration is an insignificant factor.

Using JSP with JSTL tags releases Java programmers from having to write various Java components as certain simple business tasks can be handled within the JSP file. Accordingly, this approach is the preferred one in situations where there is no requirement to display anything other than simple XML-based data or database data in a Web page, as well as no need to perform complex manipulation. In all other situations where the JSP with JSTL tags approach is used, "Model 2 JSP" is the

preferred approach.

In situations where the page author uses only HTML editor to edit Web pages, XMLC might be the preferred approach. However, implementing using XMLC becomes difficult when the Web page has a complex structure and contains complicated functionality.

Chapter 8

Future Work

The evaluation of the three approaches—JSP with embedded Java, JSP with JSTL tags, and XMLC—in this research was based on the implementation of the OGC WMS client. Consequently, some features of JSP, JSTL and XMLC have not been covered. For example, since the WMS client does not connect to a database directly, we did not used JSTL "sql" tags in our implementation. Similarly, the WMS client is a relatively simple application capable of processing only a small number of transactions. Its limitations restricted the scope of our research as there was no need to consider security requirements, scalability or consistency issues.

A client for OGC Web Feature Service (WFS) [26] could be a good case for further investigation. WFS allows data transactions on geographic features. Users can delete, update, or create a new feature instance in addition to the read-only operations. Scalable Vector Graphics (SVG) [41] will be created from GML and displayed on the Web page. EJB could be considered as the middle tier between the Web page and WFS server in the implementation of a WFS client.

Further investigation can also include other Web technologies with different development methodologies. Apache Cocoon [1] uses a new approach to generate dynamic Web pages. It is based entirely on the XML and XSLT technologies. Java Server Faces (JSF) [19] is another new technology that provides an extensible server-side user interface framework integrating with JSP and JSTL. Its goal is to make it much easier to build user interfaces for Web application. Apache Velocity [4] is also a Java-based technology that is similar to JSP. It uses Velocity Template Language (VTL) to incorporate dynamic content in a Web page.

Yet another large area for future work is the JSP editor. We have discussed in Section 5.4.1 that maintaining a JSP file is hard for a Web page designer who knows only HTML. JSP files would be far easier to maintain if a a WYSIWYG JSP editor were available to provide a graphic environment and to support JSP as well as HTML elements.

Appendix A

Sample WMS Capabilities XML Document

This document describes the capabilities of the UMN MapServer demonstration application for Itasca County located in north central Minnesota.

```
<?xml version='1.0' encoding="ISO-8859-1" standalone="no" ?>
<!DOCTYPE WMT_MS_Capabilities SYSTEM "http://www.digitalearth.gov/wmt/
xml/capabilities_1_1_0.dtd"
 [
 <!ELEMENT VendorSpecificCapabilities EMPTY>
 ]>  <!-- end of DOCTYPE declaration -->

<WMT_MS_Capabilities version="1.1.0" updateSequence="0">
<Service> <!-- a service IS a MapServer mapfile -->
  <Name>GetMap</Name> <!-- WMT defined -->
  <Title>UMN MapServer Itasca Demo</Title>
  <Abstract>This is the UMN MapServer demonstration application for
Itasca County located in north central Minnesota.</Abstract>
  <OnlineResource xmlns:xlink="http://www.w3.org/1999/xlink"
   xlink:href="http://localhost:8080/cgi-bin/mapserv?"/>
  <ContactInformation>
  </ContactInformation>
  <AccessConstraints>none</AccessConstraints>
</Service>
```

```
<Capability>
  <Request>
    <GetCapabilities>
      <Format>application/vnd.ogc.wms_xml</Format>
      <DCPType>
        <HTTP>
          <Get><OnlineResource xmlns:xlink="http://www.w3.org/1999/xlink"
                xlink:href="http://localhost:8080/cgi-bin/mapserv?"/></Get>
          <Post><OnlineResource xmlns:xlink="http://www.w3.org/1999/xlink"
                xlink:href="http://localhost:8080/cgi-bin/mapserv?"/></Post>
        </HTTP>
      </DCPType>
    </GetCapabilities>
    <GetMap>
      <Format>image/gif</Format>
      <Format>image/png</Format>
      <Format>image/wbmp</Format>
      <DCPType>
        <HTTP>
          <Get><OnlineResource xmlns:xlink="http://www.w3.org/1999/xlink"
               xlink:href="http://localhost:8080/cgi-bin/mapserv?"/></Get>
          <Post><OnlineResource xmlns:xlink="http://www.w3.org/1999/xlink"
                xlink:href="http://localhost:8080/cgi-bin/mapserv?"/></Post>
        </HTTP>
      </DCPType>
    </GetMap>
    <GetFeatureInfo>
      <Format>text/plain</Format>
      <Format>text/html</Format>
      <Format>application/vnd.ogc.gml</Format>
      <DCPType>
        <HTTP>
          <Get><OnlineResource xmlns:xlink="http://www.w3.org/1999/xlink"
               xlink:href="http://localhost:8080/cgi-bin/mapserv?"/></Get>
```

```
          <Post><OnlineResource xmlns:xlink="http://www.w3.org/1999/xlink"
                xlink:href="http://localhost:8080/cgi-bin/mapserv?"/></Post>
        </HTTP>
      </DCPType>
    </GetFeatureInfo>
  </Request>
  <Exception>
    <Format>application/vnd.ogc.se_xml</Format>
    <Format>application/vnd.ogc.se_inimage</Format>
    <Format>application/vnd.ogc.se_blank</Format>
  </Exception>
  <VendorSpecificCapabilities />
  <Layer>
    <Name>DEMO</Name>
    <Title>UMN MapServer Itasca Demo</Title>
    <SRS>EPSG:26915</SRS>
    <LatLonBoundingBox minx="-94.5002" miny="46.9476"
                       maxx="-92.9892" maxy="47.9717" />
    <BoundingBox SRS="EPSG:26915" minx="388014" miny="5.2004e+06"
                                  maxx="500802" maxy="5.31316e+06" />
    <Layer queryable="0" opaque="0" cascaded="0">
        <Name>ctybdpy2</Name>
        <Title>County Boundary</Title>
        <Abstract>Itasca County boundary shapefile. See
http://deli.dnr.state.mn.us/metadata/full/ctybdne2.html for more
information.</Abstract>
    </Layer>
    <Layer>
      <Name>cities</Name>
      <Title>cities</Title>
      <Layer queryable="1" opaque="0" cascaded="0">
        <Name>mcd90py2</Name>
        <Title>Minor Civil Divisions</Title>
        <Abstract>Minor civil divisions for Itasca County.(boundaries only)
        </Abstract>
```

```
      </Layer>
      <Layer queryable="0" opaque="0" cascaded="0">
        <Name>mcd90py2_anno</Name>
        <Title>Minor Civil Divisions Names</Title>
        <Abstract>Minor civil divisions for Itasca County.(annotation only)
        </Abstract>
      </Layer>
    </Layer>
    <Layer queryable="0" opaque="0" cascaded="0">
        <Name>twprgpy3</Name>
        <Title>Township Boundaries</Title>
        <Abstract>Pulic Land Survey (PLS) township boundaries for Itasca
County. See http://deli.dnr.state.mn.us/metadata/full/twprgne2.html for
more information.
        </Abstract>
    </Layer>
    <Layer queryable="1" opaque="0" cascaded="0">
        <Name>lakespy2</Name>
        <Title>Lakes and Rivers</Title>
        <Abstract>DLG lake and river polygons for Itasca County.
See http://deli.dnr.state.mn.us/metadata/full/dlglkpy2.html for
more information.
        </Abstract>
    </Layer>
    <Layer queryable="1" opaque="0" cascaded="0">
        <Name>dlgstln2</Name>
        <Title>Streams</Title>
        <Abstract>DLG streams for Itasca County.
See http://deli.dnr.state.mn.us/metadata/full/dlgstln2.html for
more information.
        </Abstract>
    </Layer>
    <Layer>
      <Name>roads</Name>
      <Title>roads</Title>
```

```
      <Layer queryable="0" opaque="0" cascaded="0">
        <Name>ctyrdln3</Name>
        <Title>County Roads</Title>
        <Abstract>County roads. (lines only)  Derived from MNDOT roads
layer, see http://deli.dnr.state.mn.us/metadata/full/dotrdln2.html for
more information.
        </Abstract>
      </Layer>
      <Layer queryable="0" opaque="0" cascaded="0">
        <Name>ctyrdln3_anno</Name>
        <Title>County Roads Names</Title>
        <Abstract>County roads. (shields only)  Derived from MNDOT roads
layer, see http://deli.dnr.state.mn.us/metadata/full/dotrdln2.html for
more information.
        </Abstract>
      </Layer>
      <Layer queryable="0" opaque="0" cascaded="0">
        <Name>majrdln3</Name>
        <Title>Highways</Title>
        <Abstract>Highways- state, US and interstate.(lines only)
Derived from MNDOT roads layer,
see http://deli.dnr.state.mn.us/metadata/full/dotrdln2.html for
more information.
        </Abstract>
      </Layer>
      <Layer queryable="0" opaque="0" cascaded="0">
        <Name>majrdln3_anno</Name>
        <Title>Highways Names</Title>
        <Abstract>Highways- state, US and interstate.(shields only)
Derived from MNDOT roads layer,
see http://deli.dnr.state.mn.us/metadata/full/dotrdln2.html for
more information.
        </Abstract>
      </Layer>
    </Layer>
```

```
    <Layer queryable="1" opaque="0" cascaded="0">
        <Name>airports</Name>
        <Title>Airports</Title>
        <Abstract>Airport runways for Itasca County.</Abstract>
    </Layer>
  </Layer>
</Capability>
</WMT_MS_Capabilities>
```

Appendix B

Sample GML Document presenting Feature Information

This document describes geographic feature information provided by the UMN MapServer demonstration application. The features of two layers are presented: lake and airport.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<msGMLOutput xmlns:gml="http://www.opengis.net/gml"
             xmlns:xlink="http://www.w3.org/1999/xlink"
             xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance">
<lakespy2_layer>
  <lakespy2_feature>
    <gid>422</gid>
    <area>1576600.89769</area>
    <perimeter>12743.5631</perimeter>
    <usclass>421</usclass>
    <dowlknum>69093900</dowlknum>
    <dow_verif>1</dow_verif>
    <lake_name>STURGEONWEST</lake_name>
    <lake_class>10</lake_class>
    <elevation>1371</elevation>
    <acres>389.578</acres>
    <perfeet>41811.631</perfeet>
    <gml:boundedBy>
```

```
    <gml:Box srsName="EPSG:26915">
      <gml:coordinates>
        494267.830461,5278432.894177 495357.223954,5282092.221817
      </gml:coordinates>
    </gml:Box>
  </gml:boundedBy>
  <gml:MultiPolygon srsName="EPSG:26915">
    <gml:Polygon>
      <gml:outerBoundaryIs>
        <gml:LinearRing>
          <gml:coordinates>
            494330.752194,5278593.021148 494315.566150,5278648.895884
            494318.192261,5278699.770187 494325.818703,5278768.269135
            494407.194914,5278869.765625 494447.883074,5278923.138836
            494432.696575,5278958.763820 494353.947219,5278941.016231
            494290.448176,5278946.142936 494277.761244,5278963.893072
            494277.762447,5279017.267417 494313.452045,5279131.515021
            494305.890676,5279177.264670 494267.830461,5279256.014767
            494290.770286,5279372.887692 494318.835092,5279492.135442
            494463.646310,5279522.506012 494499.208725,5279540.254797
            494550.020420,5279535.128436 494590.581688,5279504.627677
            494664.203648,5279413.126748 494702.265463,5279405.500778
            494753.076938,5279390.249545 494760.701270,5279364.874647
            494755.512256,5279293.750674 494704.635165,5279169.253639
            494689.321737,5279118.504695 494712.132064,5279034.630098
            494778.130168,5278988.878827 494813.628455,5278933.003532
            494861.876576,5278877.127883 494933.000240,5278859.251127
            495032.061599,5278866.748278 495103.186221,5278892.120982
            495151.499815,5278970.743653 495151.563155,5279008.868174
            495121.127349,5279067.368290 495027.192617,5279135.995052
            495024.694110,5279201.994298 495067.944710,5279255.367423
            495075.570112,5279278.241924 495052.759098,5279331.616894
            495055.322124,5279356.991505 495111.197099,5279389.989530
            495159.446054,5279372.113404 495215.320133,5279364.486937
            495264.086336,5279391.919508 495244.762814,5278835.100412
```

```
          495240.307955,5278825.992996 495141.182750,5278757.496609
          495113.182054,5278709.247991 495115.680060,5278620.374033
          495047.056125,5278628.000842 495003.868345,5278589.877515
          494968.306705,5278607.753279 494937.997977,5278760.127223
          494917.686082,5278775.377597 494882.124053,5278775.503584
          494838.873660,5278732.255323 494838.809581,5278661.131209
          494846.370460,5278592.631850 494838.745051,5278569.757346
          494726.931690,5278465.761735 494676.119961,5278470.763080
          494625.307897,5278460.764612 494472.871748,5278432.894177
          494429.748449,5278483.769746 494419.625119,5278552.269184
          494330.752194,5278593.021148
        </gml:coordinates>
      </gml:LinearRing>
    </gml:outerBoundaryIs>
  </gml:Polygon>
  <gml:Polygon>
    <gml:outerBoundaryIs>
      <gml:LinearRing>
        <gml:coordinates>
          495265.721227,5279439.029870 495174.824707,5279547.360777
          495027.578338,5279621.113982 494979.392565,5279669.489729
          494948.956774,5279727.864855 494936.270630,5279781.239546
          494946.458159,5279788.864164 494997.207349,5279783.737803
          495185.077028,5279659.109088 495271.450792,5279656.481698
          495273.277334,5279656.763726 495265.721227,5279439.029870
        </gml:coordinates>
      </gml:LinearRing>
    </gml:outerBoundaryIs>
  </gml:Polygon>
  <gml:Polygon>
    <gml:outerBoundaryIs>
      <gml:LinearRing>
        <gml:coordinates>
          495296.229158,5280318.134542 495205.902546,5280263.600915
          495096.653627,5280243.354254 495017.967454,5280253.606347
```

```
              494885.971499,5280352.733837 494825.036438,5280423.984673
              494746.415937,5280576.360003 494657.735590,5280840.609237
              494576.551754,5280952.485158 494449.617678,5281023.737888
              494363.308137,5281099.989405 494315.123449,5281193.989620
              494317.749326,5281234.614045 494348.249795,5281275.237675
              494414.311731,5281292.985570 494541.310786,5281333.481445
              494620.059748,5281338.479134 494708.997135,5281391.725933
              494808.058411,5281401.847977 494871.620937,5281444.970622
              495001.438551,5281762.337938 495128.564854,5281907.082474
              495222.565625,5282005.953529 495298.815542,5282054.200728
              495357.223954,5282092.221817 495352.374997,5281951.950479
              495346.246180,5281773.577907 495309.856639,5280713.702279
              495224.100740,5280761.469144 495127.666410,5280842.845860
              495046.547117,5281048.595588 494769.803774,5281168.226977
              494751.991193,5281152.977675 494746.928482,5281140.227978
              494960.172802,5281023.223364 494985.417775,5280812.475281
              495053.914943,5280728.599386 495165.662789,5280703.096536
              495221.473375,5280652.220592 495254.471788,5280601.345297
              495292.533526,5280591.219346 495302.720937,5280593.719026
              495306.057688,5280603.054703 495300.561409,5280442.970985
              495296.229158,5280318.134542
            </gml:coordinates>
          </gml:LinearRing>
        </gml:outerBoundaryIs>
      </gml:Polygon>
    </gml:MultiPolygon>
  </lakespy2_feature>
</lakespy2_layer>

<airports_layer>
  <airports_feature>
    <gid>4</gid>
    <name>Christenson Point Seaplane Base</name>
    <lat>47.6692</lat>
    <lon>-93.0544</lon>
```

```
    <elevation>1372</elevation>
    <quadname>Side Lake</quadname>
    <gml:boundedBy>
      <gml:Box srsName="EPSG:26915">
        <gml:coordinates>
          495913.000000,5279532.000000 495913.000000,5279532.000000
        </gml:coordinates>
      </gml:Box>
    </gml:boundedBy>
    <gml:Point srsName="EPSG:26915">
      <gml:coordinates>495913.000000,5279532.000000</gml:coordinates>
    </gml:Point>
  </airports_feature>

  <airports_feature>
    <gid>10</gid>
    <name>Sixberrys Landing Seaplane Base</name>
    <lat>47.6775</lat>
    <lon>-93.0481</lon>
    <elevation>1372</elevation>
    <quadname>Side Lake</quadname>
    <gml:boundedBy>
      <gml:Box srsName="EPSG:26915">
        <gml:coordinates>
          496393.000000,5280458.000000 496393.000000,5280458.000000
        </gml:coordinates>
      </gml:Box>
    </gml:boundedBy>
    <gml:Point srsName="EPSG:26915">
      <gml:coordinates>496393.000000,5280458.000000</gml:coordinates>
    </gml:Point>
  </airports_feature>
</airports_layer>

</msGMLOutput>
```

Appendix C

Sample Mapfile for UMN MapServer

This is the mapfile for the UMN MapServer with demonstration dataset for Itasca County located in north central Minnesota.

```
#
# Start of map file
#
NAME DEMO
SIZE 600 600
EXTENT 388013.643812817 5200395.13465842 500802.348432817 5313156.99196842

#
# Projection definition, consult the PROJ.4 documentation for
# parameter discussion
#
PROJECTION:
   "init=epsg:26915"
END

#
# Start of web interface definition (including WMS enabling metadata)
#
WEB
```

```
  TEMPLATE demo.html
  METADATA
    WMS_TITLE "UMN MapServer Itasca Demo"
    WMS_ABSTRACT "This is the UMN MapServer demonstration application for
Itasca County located in north central Minnesota."
    WMS_ACCESSCONSTRAINTS none
    WMS_ONLINERESOURCE "http://localhost:8080/cgi-bin/mapserv?"
    WMS_SRS "EPSG:26915"
  END
END

#
# Start of symbol definitions (we're only using a few)
#
SYMBOL
  NAME 'circle'
  TYPE ELLIPSE
  POINTS 1 1 END
  FILLED TRUE
END

#
# Start of layer definitions
#
LAYER
  NAME ctybdpy2
  CONNECTIONTYPE postgis
  CONNECTION "user=hdi12 dbname=mygisdb host=localhost port=5432"
  DATA "the_geom from ctybdpy2"
  TYPE POLYGON
  STATUS off
  CLASSITEM 'cty_name'
  CLASS
    EXPRESSION 'Itasca'
    OUTLINECOLOR 128 128 128
```

```
    COLOR 225 225 185
  END
  CLASS # every other county in the state
    EXPRESSION /./
    OUTLINECOLOR 128 128 128
    COLOR 255 255 255
  END
  METADATA
    WMS_TITLE "County Boundary"
    WMS_ABSTRACT "Itasca County boundary shapefile. See
http://deli.dnr.state.mn.us/metadata/full/ctybdne2.html for more
information."
  END
END # county boundary

LAYER
  NAME mcd90py2
  CONNECTIONTYPE postgis
  CONNECTION "user=hdi12 dbname=mygisdb host=localhost port=5432"
  DATA "the_geom from mcd90py2"
  TYPE POLYGON
  STATUS OFF
  GROUP cities
  CLASSITEM city_name
  CLASS
    NAME "Cities & Towns"
    EXPRESSION /./
    COLOR 255 225 90
    TEMPLATE "mcd90py2.html"
  END
  DUMP TRUE # allow GML export
  METADATA
    WMS_TITLE "Minor Civil Divisions"
    WMS_GROUP_TITLE "cities"
    WMS_ABSTRACT "Minor civil divisions for Itasca County.(boundaries only)"
```

```
  END
END citys

LAYER
  NAME mcd90py2_anno
  CONNECTIONTYPE postgis
  CONNECTION "user=hdi12 dbname=mygisdb host=localhost port=5432"
  DATA "the_geom from mcd90py2"
  TYPE ANNOTATION
  STATUS OFF
  GROUP cities
  LABELITEM "city_name"
  CLASSITEM "city_name"
  CLASS
    EXPRESSION /./
    COLOR -1 -1 -1
    LABEL
      COLOR 0 0 0
      SHADOWCOLOR 218 218 218
      SHADOWSIZE 2 2
      TYPE BITMAP
      SIZE MEDIUM
      POSITION CC
      PARTIALS FALSE
      BUFFER 2
    END
  END
  METADATA
    WMS_TITLE "Minor Civil Divisions Names"
    WMS_ABSTRACT "Minor civil divisions for Itasca County.(annotation only)"
  END
END # city annotation

LAYER
  NAME "twprgpy3"
```

```
  CONNECTIONTYPE postgis
  CONNECTION "user=hdi12 dbname=mygisdb host=localhost port=5432"
  DATA "the_geom from twprgpy3"
  TYPE POLYGON
  STATUS OFF
  CLASS
    SYMBOL 'circle'
    SIZE 2
    NAME 'Townships'
    OUTLINECOLOR 181 181 145
  END
  METADATA
    WMS_TITLE "Township Boundaries"
    WMS_ABSTRACT "Pulic Land Survey (PLS) township boundaries for Itasca
County. See http://deli.dnr.state.mn.us/metadata/full/twprgne2.html for
more information."
  END
END # township

LAYER
  NAME lakespy2
  CONNECTIONTYPE postgis
  CONNECTION "user=hdi12 dbname=mygisdb host=localhost port=5432"
  TYPE POLYGON
  STATUS OFF
  DATA "the_geom from lakespy2"
  CLASS
    NAME 'Lakes & Rivers'
    TEMPLATE "lakespy2.html"
    COLOR 49 117 185
  END
  DUMP TRUE # allow GML export
  METADATA
    WMS_TITLE "Lakes and Rivers"
    WMS_ABSTRACT "DLG lake and river polygons for Itasca County.
```

```
See http://deli.dnr.state.mn.us/metadata/full/dlglkpy2.html for
more information."
  END
END # lakes

LAYER
  NAME dlgstln2
  CONNECTIONTYPE postgis
  CONNECTION "user=hdi12 dbname=mygisdb host=localhost port=5432"
  DATA "the_geom from dlgstln2"
  TYPE LINE
  STATUS OFF
  CLASS
    NAME "Streams"
    TEMPLATE "dlgstln2.html"
    COLOR 49 117 185
  END
  DUMP TRUE # allow GML export
  METADATA
    WMS_TITLE "Streams"
    WMS_ABSTRACT "DLG streams for Itasca County. See
http://deli.dnr.state.mn.us/metadata/full/dlgstln2.html for more
information."
  END
END

LAYER
  NAME ctyrdln3
  CONNECTIONTYPE postgis
  CONNECTION "user=hdi12 dbname=mygisdb host=localhost port=5432"
  DATA "the_geom from ctyrdln3"
  TYPE LINE
  STATUS OFF
  GROUP roads
  CLASS
```

```
    COLOR 0 0 0
  END
  METADATA
    WMS_TITLE "County Roads"
    WMS_GROUP_TITLE "roads"
    WMS_ABSTRACT "County roads.(lines only) Derived from MNDOT roads layer,
see http://deli.dnr.state.mn.us/metadata/full/dotrdln2.html for more
information."
  END
END # county roads

LAYER
  NAME ctyrdln3_anno
  CONNECTIONTYPE postgis
  CONNECTION "user=hdi12 dbname=mygisdb host=localhost port=5432"
  DATA "the_geom from ctyrdln3"
  TYPE ANNOTATION
  STATUS OFF
  LABELITEM "road_name"
  GROUP roads
  CLASS
    COLOR 255 255 255
    SYMBOL 'symbols/ctyhwy.gif'
    LABEL
      MINFEATURESIZE 40
      MINDISTANCE 150
      POSITION CC
      SIZE TINY
      COLOR 0 0 0
    END
  END
  METADATA
    WMS_TITLE "County Roads Names"
    WMS_ABSTRACT "County roads.(shields only) Derived from MNDOT roads
layer, see http://deli.dnr.state.mn.us/metadata/full/dotrdln2.html for
```

```
more information."
  END
END # county road annotation

LAYER
  NAME majrdln3
  CONNECTIONTYPE postgis
  CONNECTION "user=hdi12 dbname=mygisdb host=localhost port=5432"
  DATA "the_geom from majrdln3"
  TYPE LINE
  STATUS OFF
  GROUP roads
  CLASS
    NAME "Roads"
    COLOR 255 0 0
    SIZE 10
  END
  METADATA
    WMS_TITLE "Highways"
    WMS_ABSTRACT "Highways- state, US and interstate.(lines only)
Derived from MNDOT roads layer,
see http://deli.dnr.state.mn.us/metadata/full/dotrdln2.html for more
information."
  END
END # highways

LAYER
  NAME majrdln3_anno
  CONNECTIONTYPE postgis
  CONNECTION "user=hdi12 dbname=mygisdb host=localhost port=5432"
  DATA "the_geom from majrdln3"
  TYPE ANNOTATION
  STATUS OFF
  GROUP roads
  LABELITEM "road_num"
```

```
CLASSITEM "road_class"
CLASS
  EXPRESSION "3"
  COLOR 0 0 0 # dummy color
  SYMBOL 'symbols/sthwy.gif'
  LABEL
    MINFEATURESIZE 50
    MINDISTANCE 150
    POSITION CC
    SIZE TINY
    COLOR 0 0 0
  END
END
CLASS
  EXPRESSION "2"
  COLOR 0 0 0 # dummy color
  SYMBOL 'symbols/ushwy.gif'
  LABEL
    MINFEATURESIZE 50
    MINDISTANCE 150
    POSITION CC
    SIZE TINY
    COLOR 0 0 0
  END
END
CLASS
  EXPRESSION "1"
  COLOR 0 0 0 # dummy color
  SYMBOL 'symbols/interstate.gif'
  LABEL
    MINFEATURESIZE 50
    MINDISTANCE 150
    POSITION CC
    SIZE TINY
    COLOR 255 255 255
```

```
    END
  END
  METADATA
    WMS_TITLE "Highways Names"
    WMS_ABSTRACT "Highways- state, US and interstate.(shields only)
Derived from MNDOT roads layer,
see http://deli.dnr.state.mn.us/metadata/full/dotrdln2.html for more
information."
  END
END # highway annotation

LAYER
  NAME "airports"
  CONNECTIONTYPE postgis
  CONNECTION "user=hdi12 dbname=mygisdb host=localhost port=5432"
  DATA "the_geom from airports"
  TYPE POINT
  STATUS off
  CLASS
    NAME 'Airports'
    COLOR 0 0 0
    #COLOR 128 255 164
    SYMBOL 'circle'
    SIZE 7
    TEMPLATE "airports.html"
  END
  DUMP TRUE # allow GML export
  METADATA
    WMS_TITLE "Airports"
    WMS_ABSTRACT "Airport runways for Itasca County."
  END
END # ariports

END # Map File
```

Appendix D

Source file *game.java*

```
/*
 ************************************
 * XMLC GENERATED CODE, DO NOT EDIT *
 ************************************
 */
import org.w3c.dom.*;
import org.enhydra.xml.xmlc.XMLCError;
import org.enhydra.xml.xmlc.XMLCUtil;
import org.enhydra.xml.xmlc.dom.XMLCDomFactory;

/**
 * XMLC Document class, generated from
 * game.html
 */
public class game extends org.enhydra.xml.xmlc.html.HTMLObjectImpl
    implements org.enhydra.xml.xmlc.XMLObject,
    org.enhydra.xml.xmlc.html.HTMLObject {

    private int $elementId_result = 15;
    private int $elementId_state = 8;
    private org.enhydra.xml.lazydom.html.LazyHTMLElement $element_Result;
    private org.enhydra.xml.lazydom.html.HTMLFontElementImpl $element_State;
```

```
/**
 * Field that is used to identify this as the XMLC generated class
 * in an inheritance chain. Contains a reference to the class object.
 */
public static final Class XMLC_GENERATED_CLASS = game.class;

/**
 * Field containing CLASSPATH relative name of the source file
 * that this class can be regenerated from.
 */
public static final String XMLC_SOURCE_FILE = "/game.html";

/**
 * XMLC DOM factory associated with this class.
 */
private static final org.enhydra.xml.xmlc.dom.XMLCDomFactory fDOMFactory =
    org.enhydra.xml.xmlc.dom.XMLCDomFactoryCache.getFactory(
    org.enhydra.xml.xmlc.dom.lazydom.LazyHTMLDomFactory.class);

/**
 * Options used to preformat the document when compiled
 */
private static final org.enhydra.xml.io.OutputOptions fPreFormatOutputOptions;

/**
 * Template document shared by all instances.
 */
private static final org.enhydra.xml.lazydom.TemplateDOM fTemplateDocument;

/**
 * Lazy DOM document
 */
private org.enhydra.xml.lazydom.LazyDocument lazyDocument;

/*
```

```
 * Class initializer.
 */
static {
    org.enhydra.xml.lazydom.html.LazyHTMLDocument doc =
        (org.enhydra.xml.lazydom.html.LazyHTMLDocument
         )fDOMFactory.createDocument(null, "HTML", null);
    buildTemplateSubDocument(doc, doc);
    fTemplateDocument = new org.enhydra.xml.lazydom.TemplateDOM(doc);
    fPreFormatOutputOptions = new org.enhydra.xml.io.OutputOptions();
    fPreFormatOutputOptions.setFormat(
        org.enhydra.xml.io.OutputOptions.FORMAT_AUTO);
    fPreFormatOutputOptions.setEncoding("ISO-8859-1");
    fPreFormatOutputOptions.setPrettyPrinting(false);
    fPreFormatOutputOptions.setIndentSize(4);
    fPreFormatOutputOptions.setPreserveSpace(true);
    fPreFormatOutputOptions.setOmitXMLHeader(false);
    fPreFormatOutputOptions.setOmitDocType(false);
    fPreFormatOutputOptions.setOmitEncoding(false);
    fPreFormatOutputOptions.setDropHtmlSpanIds(true);
    fPreFormatOutputOptions.setOmitAttributeCharEntityRefs(true);
    fPreFormatOutputOptions.setPublicId(null);
    fPreFormatOutputOptions.setSystemId(null);
    fPreFormatOutputOptions.setMIMEType(null);
    fPreFormatOutputOptions.markReadOnly();
}

/**
 * Default constructor.
 */
public game() {
    buildDocument();
}

/**
 * Constructor with optional building of the DOM.
```

```
 */
public game(boolean buildDOM) {
    if (buildDOM) {
        buildDocument();
    }
}

/**
 * Copy constructor.
 */
public game(game src) {
    setDocument((Document)src.getDocument().cloneNode(true),
                         src.getMIMEType(), src.getEncoding());
    syncAccessMethods();
}

/**
 * Create document as a DOM and initialize accessor method fields.
 */
public void buildDocument() {
    lazyDocument = (org.enhydra.xml.lazydom.html.LazyHTMLDocument)(
                    (org.enhydra.xml.xmlc.dom.lazydom.LazyDomFactory
                     )fDOMFactory).createDocument(fTemplateDocument);
    lazyDocument.setPreFormatOutputOptions(fPreFormatOutputOptions);
    setDocument(lazyDocument, "text/html", "ISO-8859-1");
}

/**
 * Create a subtree of the document.
 */
private static void buildTemplateSubDocument(
    org.enhydra.xml.lazydom.LazyDocument document,
    org.w3c.dom.Node parentNode) {

    Node $node0, $node1, $node2, $node3, $node4;
```

```
Element $elem0, $elem1, $elem2, $elem3;
Attr $attr0, $attr1, $attr2, $attr3;
Element $docElement = document.getDocumentElement();

$node1 = document.createTemplateComment(
         " Template html file game.html ", 1);
parentNode.insertBefore($node1, $docElement);

$elem1 = document.getDocumentElement();
((org.enhydra.xml.lazydom.LazyElement)$elem1).makeTemplateNode(2);
((org.enhydra.xml.lazydom.LazyElement)$elem1
 ).setPreFormattedText("<HTML>");

$elem2 = document.createTemplateElement("HEAD", 3, "<HEAD>");
$elem1.appendChild($elem2);

$elem3 = document.createTemplateElement("TITLE", 4, "<TITLE>");
$elem2.appendChild($elem3);

$node4 = document.createTemplateTextNode("game", 5, "game");
$elem3.appendChild($node4);

$elem2 = document.createTemplateElement("BODY", 6, "<BODY>");
$elem1.appendChild($elem2);

$node3 = document.createTemplateTextNode("It is ", 7, "It is ");
$elem2.appendChild($node3);

$elem3 = document.createTemplateElement(
 "FONT", 8, "<FONT color=\"red\" id=\"state\">");
$elem2.appendChild($elem3);
$attr3 = document.createTemplateAttribute("color", 9);
$elem3.setAttributeNode($attr3);

$node4 = document.createTemplateTextNode("red", 10, "red");
```

```
    $attr3.appendChild($node4);
    $attr3 = document.createTemplateAttribute("id", 11);
    $elem3.setAttributeNode($attr3);

    $node4 = document.createTemplateTextNode("state", 12, "state");
    $attr3.appendChild($node4);

    $node4 = document.createTemplateTextNode("good or bad", 13,
                                             "good or bad");
    $elem3.appendChild($node4);

    $node3 = document.createTemplateTextNode(" news! We ", 14,
                                             " news! We ");
    $elem2.appendChild($node3);

    $elem3 = document.createTemplateElement("SPAN", 15, "<SPAN>");
    $elem2.appendChild($elem3);
    $attr3 = document.createTemplateAttribute("id", 16);
    $elem3.setAttributeNode($attr3);

    $node4 = document.createTemplateTextNode("result", 17, "result");
    $attr3.appendChild($node4);

    $node4 = document.createTemplateTextNode("won or lost", 18,
                                             "won or lost");
    $elem3.appendChild($node4);

    $node3 = document.createTemplateTextNode(" the game.", 19,
                                             " the game.");
    $elem2.appendChild($node3);
}

/**
 * Clone the document.
 */
```

```
public Node cloneNode(boolean deep) {
    cloneDeepCheck(deep);
    return new game(this);
}

/**
 * Get the XMLC DOM factory associated with the class.
 */
protected final org.enhydra.xml.xmlc.dom.XMLCDomFactory getDomFactory() {
    return fDOMFactory;
}

/**
 * Get the element with id <CODE>result</CODE>.
 * @see org.w3c.dom.html.HTMLElement
 */
public org.w3c.dom.html.HTMLElement getElementResult() {
    if (($element_Result == null) && ($elementId_result >= 0)) {
        $element_Result = (org.enhydra.xml.lazydom.html.LazyHTMLElement
                   )lazyDocument.getNodeById($elementId_result);
    }
    return $element_Result;
}

/**
 * Get the element with id <CODE>state</CODE>.
 * @see org.w3c.dom.html.HTMLFontElement
 */
public org.w3c.dom.html.HTMLFontElement getElementState() {
    if (($element_State == null) && ($elementId_state >= 0)) {
        $element_State = (org.enhydra.xml.lazydom.html.HTMLFontElementImpl
                  )lazyDocument.getNodeById($elementId_state);
    }
    return $element_State;
}
```

```
/**
 * Get the value of text child of element <CODE>result</CODE>.
 * @see org.w3c.dom.Text
 */
public void setTextResult(String text) {
    if (($element_Result == null) && ($elementId_result >= 0)) {
        $element_Result = (org.enhydra.xml.lazydom.html.LazyHTMLElement
                    )lazyDocument.getNodeById($elementId_result);
    }
    doSetText($element_Result, text);
}

/**
 * Get the value of text child of element <CODE>state</CODE>.
 * @see org.w3c.dom.Text
 */
public void setTextState(String text) {
    if (($element_State == null) && ($elementId_state >= 0)) {
        $element_State = (org.enhydra.xml.lazydom.html.HTMLFontElementImpl
                   )lazyDocument.getNodeById($elementId_state);
    }
    doSetText($element_State, text);
}

/**
 * Recursize function to do set access method fields from the DOM.
 * Missing ids have fields set to null.
 */
protected void syncWithDocument(Node node) {
    if (node instanceof Element) {
        String id = ((Element)node).getAttribute("id");
        if (id.length() == 0) {
        } else if (id.equals("result")) {
            $elementId_result = 15;
```

```
                $element_Result =
    (org.enhydra.xml.lazydom.html.LazyHTMLElement)node;
            } else if (id.equals("state")) {
                $elementId_state = 8;
                $element_State =
    (org.enhydra.xml.lazydom.html.HTMLFontElementImpl)node;
            }
        }
        Node child = node.getFirstChild();
        while (child != null) {
            syncWithDocument(child);
            child = child.getNextSibling();
        }
    }

}
```

Bibliography

[1] *Apache Cocoon home page*. http://cocoon.apache.org/2.0/.

[2] *Apache HTTP Server home page*. http://httpd.apache.org.

[3] *Apache Struts home page*. http://jakarta.apache.org/struts.

[4] *Apache Velocity home page*. http://jakarta.apache.org/velocity/.

[5] *Common Gateway Interface*. http://www.w3.org/DOM/.

[6] CompuServe Incorporated. *GRAPHICS INTERCHANGE FORMAT(sm) Version 89a*, July 1990. http://www.dcs.ed.ac.uk/home/mxr/gfx/2d/GIF89a.txt.

[7] *CubeView Demo*. http://demo.cubewerx.com/demo/dubeview.

[8] *Data Access Object*. http://java.sun.com/blueprints/patterns/DAO.html.

[9] D.H.Young. *A Frient Game of Yug of War: XMLC vs JSP*, April 2002. http://www2.theserverside.com/resources/article.jsp?l=XMLCvsJSP.

[10] *Digital Earth Web Map Viewer*. http://viewer.digitalearth.gov.

[11] *Enhydra Barracuda home page*. http://barracuda.enhydra.org.

[12] *Enhydra XMLC home page*. http://xmlc.enhydra.org.

[13] *GD Graphics Library*. http://www.boutell.com/gd/.

[14] G.Seshadri. *Understanding JavaServer Pages Model 2 architecture*. http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-jspmvc_p.html.

[15] H.Sheil. *OSS MVC Application Framework Matrix Review.* http://www.jcorporate.com/html/products/expresso/matrix_compare.html.

[16] *InterGraph WMS Viewer.* http://www.wmsviewer.com.

[17] I.Singh, B.Stearns, M.Johnson, and the Enterprise Team. *Designing Enterprise Applications with the J2EE^{TM} Platform*, chapter 1. Addison-Wesley, 2nd edition, March 2002.

[18] *Jakarta Tomcat home page.* http://jakarta.apache.org/tomcat.

[19] *JavaServer Faces Technology.* http://java.sun.com/j2ee/javaserverfaces/.

[20] Inc. Lutris Technologies. *Barracuda - Framework Comparisons.* http://barracuda.enhydra.org/cvs_source/Barracuda/docs/barracuda_vs_struts.html.

[21] *MapServer home page.* http://mapserver.gis.umn.edu.

[22] *Model-View-Controller.* http://java.sun.com/blueprints/patterns/MVC.html.

[23] Open GIS Consortium Inc. *OpenGIS Simple Feature Specification For SQL. Version 1.1*, May 1999. Open GIS project document 99-049.

[24] Open GIS Consortium Inc. *Geography Markup Language (GML) 2.0*, February 2001. OGC Document Number: 01-029.

[25] Open GIS Consortium Inc. *Web Map Service Implementation Specification. Version 1.1.0*, June 2001. Open GIS project document: OGC 01-047r2.

[26] Open GIS Consortium Inc. *Web Feature Service Implementation Specification. Version 1.0.0*, September 2002. Open GIS project document: OGC 02-058.

[27] *PostGIS home page.* http://postgis.refractions.net.

[28] *PostgreSQL home page.* http://www.postgresql.org.

[29] *PROJ.4 - Cartographic Projections Library.* http://www.remotesensing.org/proj/.

[30] S.Bayern. *Introducing The JSP Standard Tag Library*, April 2002. http://www.theserverside.com/resources/articles/JSTL/article.html.

[31] Sun Microsystems Inc. *Java^{TM} Language Specification*, 2nd edition, 1996.

[32] Sun Microsystems Inc. *JavaBeans^{TM} Specification. Version 1.01*, August 1997.

[33] Sun Microsystems Inc. *JavaServer Pages^{TM} Specification. Version 1.2*, August 2001.

[34] Sun Microsystems Inc. *Java^{TM} Servlet API Specification Version: 2.3*, September 2001.

[35] Sun Microsystems Inc. *JavaServer Pages^{TM} Standard Tag Library (JSTL) Specification. Version 1.0*, March 2002.

[36] Sun Microsystems Inc. *Java^{TM} 2 Platform Enterprise Edition Specification, v1.4*, April 2003.

[37] W3C Recommendation. *PNG (Portable Network Graphics) Specification Version 1.0*, October 1996. http://www.w3.org/TR/PNG.

[38] W3C Recommendation. *XML Path Language (XPath) Version 1.0*, November 1999. http://www.w3.org/TR/1999/REC-xpath-19991116.

[39] W3C Recommendation. *XSL Transformations (XSLT) Version 1.0*, November 1999. http://www.w3.org/TR/xslt/.

[40] W3C Recommendation. *Extensible Markup Language (XML) 1.0*, 2nd edition, October 2000. http://www.w3.org/TR/2000/REC-xml-20001006.

[41] W3C Recommendation. *Scalable Vector Graphics (SVG) 1.0 Specification*, September 2001. http://www.w3.org/TR/SVG/.