MagnumServer Pages: Improvements and Extensions to JavaServer Pages

Patrick J Margey, B.Sc.

Master of Science (M.Sc.)

Letterkenny Institute of Technology

Supervisor: Jonathan Campbell, B.A., B.A.I., D.Phil.

Submitted to the Higher Education and Training Awards Council February 2005

Acknowledgements

I would like to thank my supervisor, Dr. Jon Campbell, to whom I am will be forever grateful for his ideas, insights, encouragement and especially this friendship during the duration of this masters.

I would also like to give a resounding thanks to my parents for their never-ending love, support and encouragement.

Also thanks must be given to my three brothers for their strength, advice and encouragement. *"All for one, and one for all"*.

Finally, a deep heartfelt thanks goes to Sharon for just being herself along the away.

Declaration

I hereby declare that with effect from the date on which the dissertation is deposited in the Library of Letterkenny Institute of Technology I permit the Librarian of the Institute to allow the dissertation to be copied in whole or in part without reference to me on the understanding that such authority applies to the provision of single copies made for study purposes or for inclusion within the stock of another library. This restriction does not apply to the copying or publication of the title and abstract of the dissertation.

IT IS A CONDITION OF USE OF THIS DISSERATION THAT ANYONE WHO CONSULTS IT MUST RECOGNISE THAT THE COPYRIGHT RESTS WITH THE AUTHOR AND THAT NO QUOTATION FROM THE DISSERTATION AND NO INFORMATION DERIVED FROM IT MAY BE PUBLISHED UNLESS THE SOURCE IS PROPERLY ACKNOWLEDGED

Abstract

Today vast amounts of services and information are provided by the WWW. By its very nature, the information involved is changeable; hence static web pages are no longer adequate and methods of coping with dynamic information are needed. One such technology from Sun Microsystems is called JavaServer Pages (JSP).

JSP is an integral component of J2EE and can be viewed as a simplified and augmented version of its parent technology Java servlets. JSP provides businesses with a means to rapidly develop robust large-scale web applications, as it offers programmers the ability to work parallel with web designers and provides a mechanism to easily integrate Java code with static HTML.

However JSP technology does have weaknesses; for example there is no standard design approach, no caching or compression mechanisms to improved presentation speed, automated testing is difficult and there are a number of known security vulnerabilities. As a result the industry has recognised these weaknesses and have started to develop new servlet frameworks / template engines that supply them with the ability to develop maintainable and cost effective web applications. Hence developers are now burdened with an indulgence of complex Java frameworks that require a steep learning curve to master.

The overall aim of this dissertation is to analyse, design, implement and evaluate a new improved Java web based technology (that we call MagnumServer Pages) and its corresponding novel servlet design framework. The new design will ultimately simplify the development process into easily understood components that resolve the issues surrounding JSP. The results of a detailed evaluation and benchmarking indicates that the new design is a flexible framework that provides reduced coupling, increased presentation speed, support for automated testing and a seamless development process.

Table of Figures

Page

Table of Tables

Page

Table of contents

1 Introduction

1.1 Purpose and Scope

This document is a thesis submitted in part fulfilment of the requirements for the degree of Master of Science at Letterkenny Institute of Technology Department of Computing. The topic is that of constructing a new robust Java web based technology, which will resolve some fundamental problems surrounding JavaServer Pages (JSP). The new technology will provide support for additional competing technologies, increased presentation speed and finally decoupled application code that can be easily unit tested.

1.2 Background and Overview

In recent years the WWW has changed significantly in terms of serving HTML content to clients Therefore over the course of time the range of dynamic web-based technologies (for example, CGI, PHP and Java servlets) has grown.

However, these dynamic web-based technologies have their own shortcoming such as scalability, performance, maintainability and cost of development. Specifically, in the case of Java servlets, there is no separation of programming logic and HTML processing; this results in costly development and maintenance difficulties. Hence JavaServer Pages (JSP) was created as an extension of Java servlets and quickly became the standard Java solution to dynamic HTML. The reason for this was that it offered developers a simplified way to create and maintain servlet style code that still contained the full power of its parent technology (that is, Java servlets).

Although JSP is the standard Java web solution, it is not the only solution. Currently, software houses have recognised some limitations with JSP, particularly in the areas of design, performance, testability and security. Therefore some software houses have started to develop servlet frameworks and template engines (for example, Apache Struts and Tapestry), which try to solve the limitations of JSP by applying new design patterns

and coding approaches. However there are costs associated with these new frameworks, such as poor documentation and high complexity for average programmers.

Thus the dissertation will present a new solution / approach for creating dynamic content implemented in Java. This solution will increase responsivity, security, testability and provide developers with a more intuitive and flexible design framework.

The objectives of this dissertation are as follows:

1. Conduct a detail literature review to investigate the nature of JSP and it's competing technologies;
2. Investigate the nature of JSP performance, security, error handling, debugging, ad-hoc design and the weakness of separation of presentation from business logic.
3. Follow software development best practices (for example, using object oriented design patterns and UML design processes) to design, implement and evaluate the new solution;
4. Discuss and suggestion future enhancements.

1.3 Outline of Document

This dissertation is divided into seven chapters, the first chapter aims to outline the scope and the main objectives for the dissertation.

Chapter two provides a review of the available literature in the context of JavaServer Pages (JSP). That is, the history of dynamic web technology is examined, alternative technologies are explained, Java Enterprise Edition (J2EE) model is explored in the context of the WWW, Java servlets are discussed and in particular JSP are explained in detail.

Chapter three examines the problems of JSP in the context of design, performance, testability and security.

Chapter four describes a plan to resolve the problems of JSP in the context of design, performance, testability and security.

Chapter five describes how the new design was implemented, that is explaining the architecture and construction in detail.

Chapter six evaluates the newly implemented design against competing Java web frameworks / technologies.

Chapter seven summarises the overall findings of the project and outlines possible future work.

2 Literature Review

2.1 Introduction

Currently there are many competing dynamic web page technologies, such as PHP and CGI, which offer their own unique advantages and disadvantages for building web applications in terms of design, performance, security and testability. Even though this chapter discusses and highlights the strengths and weaknesses of some of these competing technologies, the fundamental purpose of this chapter is to discuss dynamic web page technology in the context of the standard Java solution for producing dynamic HTML, namely JavaServer Pages (JSP).

JSP is an extension of the Java servlet architecture [Sun, 2001]; both JSP and servlets are server-side Java technologies that are supported on the majority of today's application servers (in the context of J2EE, "application server" can be defined as a web server that provides the mechanism to serve dynamic content). These technologies provide a platform independent language that offers an extensive library of predefined classes for developing dynamic HTML content. In conjunction with the existing Java standard development kit (JSDK) class libraries, the predefined servlet and JSP Java class libraries can be used to build enterprise scale Java web applications. The servlet and JSP Java class libraries are particularly powerful since they offer additional functionality support from the more traditional JSDK support (ranging from database connectivity to multithreaded network processing) [Sun, 2002]. Although similar, JSP differs from servlet technology in that it is a web-scripting language that attempts to separate static content (HTML) from dynamic presentation (servlet code).

The following sections will discuss JSP under the when, what, where, why, and how; that is, the following sections will provide detailed answers to the following questions:

- Why did dynamic web technology arise?
- Where does JSP fit into the overall Java model?
- What are Java servlets?
- Why did JSP technology occur?

- What are JSP?
- How does JSP work?
- What are the advantages of JSP?
- What exactly is JSP relationship with servlets?
- How does JSP help developers?
- How is JSP different from competing technologies?
- What other Java technologies can be used in tandem with JSP?
- What are JSP competing technologies?

2.2 The evolution of dynamic web technology

When the WWW was created, its primary purpose was to serve static HTML pages. However this primary purpose changed when people started to use the WWW as not only a means to find static information but as a tool to perform daily tasks, for example, banking and shopping. To perform these daily tasks the WWW started to serve dynamic content [Kassem et al, 2002]. The serving of dynamic content occurs when a client's browser submits an HTTP request for a particular web page (typically implemented by a scripting language or technology such as ASP, PHP, CGI or JSP) on a web server [Brown et al, 2001], the web server would locate the dynamic page, execute its program and retrieve the page result as HTML through a corresponding HTTP response [Fields et al, 2000].

To further this discussion, we must discuss some of the dynamic web technologies alternative to JSP, particularly in the context of different language implementations and what advantages and disadvantages they bring.

2.2.1 Common Gateway Interface

Common Gateway Interface (CGI) was one of the first technologies to be used for building dynamic HTML [Fields et al, 2000]. CGI by itself is not a programming language; it is a standard lightweight interface that is based on the same model that the web server uses to serve static HTML files [Birznieks et al ,2000]. That is, the web server reads an incoming HTTP request from a URL and identifies a server side CGI resource file (that is, an interface file denoted by `.cgi` extension). Sequentially,

the web server executes CGI resource file, waits until the CGI process has finished and sends the resultant HTML output as response back to the client [Christiansen et al, 1998]. CGI code can be written in most languages [NSCA, 98] and while a CGI file is executed, the application code is sequential executed to print out one large textual string (The textual string is a combination of intermixed static HTML and dynamic functionality).

2.2.1.1 Advantages of CGI

The following are the advantages associated with using CGI:

a) Program languages

CGI applications can be written in most programming languages, for example, Perl, Python, Visual Basic, C/C++, Unix shell scripts, and even COBOL.

b) Large range of robust utility libraries

Depending on the programming language that you use for your CGI file, for example, Perl or C / C++, your CGI code would have access to a large set of built-in libraries, which would provide extra functionality to developers so they can quickly develop applications with the minimum effort.

c) Learning curve

Since CGI can use any one of a wide range of programming languages for its coding (which most developers and students have used at least one in their work/studies) and CGI is extensively documented with workable examples. Therefore most developers can become highly productive without huge effort.

2.2.1.2 Disadvantages of CGI

The following are the disadvantages associated with using CGI:

a) Use of interpreted languages

In most cases, the programming language you use to build CGI applications is not compiled, for example, Perl and Python. Once a CGI script / program is called the interpreter is loaded, the script is checked for errors at run-time, then executed as a single process on the server. This process is slow to execute and has large memory footprint [Wu et al, 2000] because the CGI file has to

be interpreted for every single HTTP request and external resources have to be held in memory until termination of CGI process.

b) Scalability

This single process execution does not provide support for threading, for example in the use with database and object pooling. CGI applications have an increased load time to connect to external resources such as databases and shared libraries as these external connections need to be created and reloaded each time the CGI code is executed. Therefore this process has a detrimental effect on the performance of the web server as it uses valuable CPU memory in a processor inefficient manner [Wu et al, 2000].

c) Performance

No matter which programming language you use, CGI cannot save user session data in memory. The reason for this is that upon every request for a CGI resource file, a single process is executed and then terminated on completion. Therefore memory allocation must be reinitialised for every request. Some of the programming languages that can be used with CGI, for example, Perl uses a combination of text file manipulation (reading and writing to a file) and databases for data persistence.

2.2.2 PHP (PHP Hypertext Processor)

PHP is an open source platform independent server side scripting language.
It is an interpreted language that is best described as a cross between C/C++ and Perl. PHP was designed to simplify manipulation of databases and provide a set of reusable libraries that could be used to build dynamic content for the WWW [Fields et al, 2000]. The PHP scriptlet is embedded into HTML and then executed to give dynamic functionality [Bakken et al, 2003].

It was created in 1994 by Rasmus Lerdorf as a way to track users entering his website. Lerdorf originally named PHP (Personal Home Programming) but throughout the years the language has become more generally accepted and is now adopted by the

GNU community. Currently over 11 million domains use PHP as the main server side scripting language to render their web pages.

The following subsections describe some of the advantages and disadvantages that are currently associated with PHP.

2.2.2.1 Advantages of PHP

The following are the advantages associated with using PHP:

a) Large range of robust utility libraries

Since PHP is an open source technology that was built primary for web development, it comes which an array of built-in libraries, for example, Java and `.NET` plug-ins, XML and database libraries, which hide mundane tasks from developers so they can quickly develop applications with the minimum effort [Bakken et al, 2003] [Welling et al, 2001].

b) Database integration

PHP has many easy to use predefined libraries to connect and interact with many industrial standard Databases, for example, `MySQL`, `PostgresSQL`, `mSQL`, `Oracale`, `Sybase` and `SQL server` etc. This functionality provides low configuration and start up time to building robust database driven web systems [Bakken et al, 2003] [Welling et al, 2001].

c) Free to the public

There is no licensing or cost associated with PHP. It is freely available on the web and is supported by most major Internet Service Providers (ISP)

d) Learning curve

PHP is very similar to Perl, C and C++ (which most developers and students have used in their work/studies) and the PHP language is extensively documented with workable examples. This means that most Perl, C and C++ skilled developers can become highly productive without huge effort [Welling et al, 2001].

e) Platform independent

The PHP language can run on any UNIX systems such as Linux, Solaris, etc. and any Windows based platform.

2.2.2.2 Disadvantages of PHP

The following are the disadvantages associated with using PHP:

a) Weak abstraction for databases

PHP comes with a large array of database libraries that use different method calls to connect to and interact with specific databases. This results in maintenance difficulty for developers to switch databases within their applications [Wu et al, 2000]. For example, some of the database connection functions for PHP are:

`mysql_connect()` - establishes a connection to a `MySQL` server;

`ifx_connect()` - establishes a connection to an `Informix` server;

`sybase_connect()` - establishes a connection to a `Sybase` server.

b) Interpreted language

Like all interpreted languages, PHP is not compiled. As stated earlier, once a script is called the interpreter is loaded, the script is checked for errors at run-time, then executed via a single process on the server; hence this process is slow too execute and has large memory footprint [Wu et al, 2000].

c) Scalability

This single process execution does not provide support for threading, for example in the use with database and object pooling. PHP applications have an increased load time to connect to external resources (databases and files) and internal components need to be built each time etc which all have a decreasing effect on the performance of the web server [Wu et al, 2000].

2.2.3 ASP.NET

Active Server Pages .NET (ASP.NET) was created by Microsoft as a core sub component of the .NET framework. The specific purpose of ASP.NET is to provide an event driven development approach to building dynamic web pages. In the case of

event driven, we mean that ASP.NET intentions is to provide a high level Application Programmer Interface (API) (which is part of the .NET Framework Class Library (FCL)), which a developer can use to implement the minimal amount of code for separating the background engine code from the user interface portions of a dynamic web page. ASP.NET can be implemented using any of the five languages for the .NET framework. That is, C#, VB, C++, JScript and J++ [Kalani, 2003].

The following are the some of the advantages and disadvantages that are currently associated with ASP.NET.

2.2.3.1 Advantages of ASP.NET

The following are the advantages associated with using ASP.NET:

a) Large range of robust utility libraries

As ASP.NET is a part of the overall .NET Framework Class Library (FCL), it can use any predefined classes from the FCL to support basically any functionality such as web services to file manipulation. These predefined class libraries offer programmers more power to develop large-scale reusable components that can form enterprise solutions to large organisations [Kalani, 2003].

b) Performance

Compared to competing technologies (for example, PHP, CGI and JSP), an application's overall performance can be improved when the application has been developed in ASP.NET. This improvement can occur in two ways which most of the competing technologies do not implement, firstly any dynamic web page developed using ASP.NET is compiled before it is executed (therefore saving time on interpreting the source code) and secondly the compiled version of the dynamic web page is cached (therefore saving time on recompilation) [Kalani, 2003].

c) Scalability

An application developed in ASP.NET can be distributed across several machines or several processes of the same machine. Therefore a web

application can scale smoothly as the number of users increases [Kalani, 2003].

d) Learning curve

Programmers can become productive at an early stage since ASP.NET can be developed in a number of languages (for example, C# and VB) and is extensively documented.

2.2.3.2 Disadvantages of ASP.NET

The following are the disadvantages associated with using ASP.NET:

a) Platform dependent

Even though ASP.NET source code is compiled into Microsoft Intermediate Language (MSIL) (which is platform independent native code). One drawback of ASP.NET is that any web application developed with this framework needs to be deployed on a Microsoft specific web server such as IIS because the MSIL has yet to be embraced by other operating systems.

2.2.4 Conclusions

This subsection has offered an insight on the inception and growth of dynamic web technology and in particular covered some of the more important JSP alternatives. Therefore we must now investigate and discuss the role of JSP technology in terms of the overall Java model.

2.3 JSP compatibility with the Java Enterprise Edition model

The purpose of this section is to define the role that the JSP architecture plays in the overall scheme of the Java Enterprise Edition (J2EE) model; one must understand that J2EE is an architecture that consists of many technologies such as Enterprise JavaBeans (EJB), CORBA, XML, servlets, JSP and Web services. Under J2EE these technologies can be categorised into three distinct groups: *component*, *service* and *communication* [Kassem et al, 2002].

Due to the vastness of these three J2EE technology groups, this section shall only examine a subsection of the *component* group called the *Web tier*. The *Web tier*

covers all the fundamental components in relation to JSP and will explain the processes involved in deploying a Java based application onto the web.

As highlighted previously, the J2EE platform isn't a single entity. J2EE is amalgamation of many Java technologies [Kassem et al, 2002] and before discussing the J2EE *Web tier*; we must explain in detail about three tiered architectures. That is, J2EE's primary focus is to provide the technologies that produce such software architectures.

Before continuing, we must define "business logic"; which is used throughout this dissertation; it refers to some context, that is, software component operations; that make data relevant for an application. Basically *business logic* refers to the logic rather than the view / representaion of that data. That is, it refers to the manipulation of data [McLaughlin, 2002].

2.3.1 Three Tiered Architecture

A *three tiered architecture* describes the situation in which an application is broken into a three tier distributed client / server design; the purpose of these tiers is to provide a loosely coupled architecture that can be developed in parallel (see Figure 2.1).

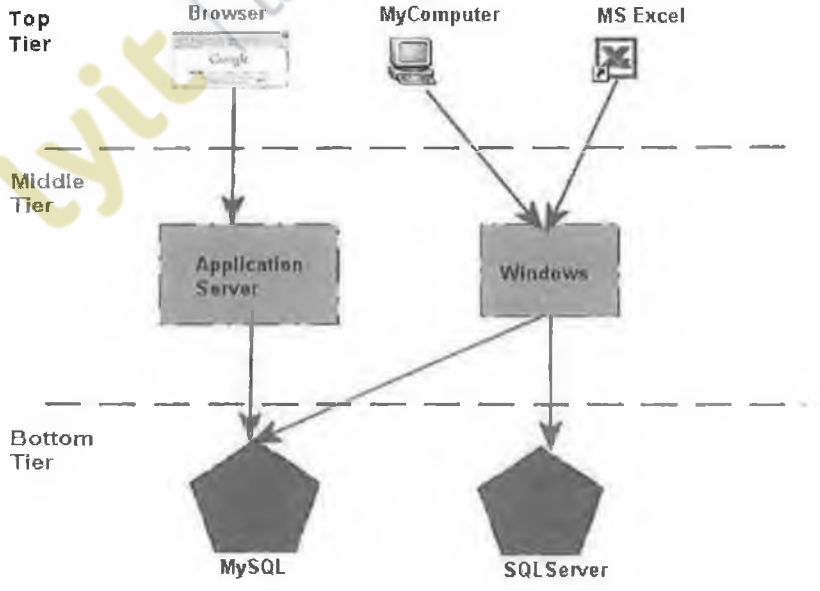


Figure 2.1: Example of Three Tiered Architecture

(Please note: although the image of a browser in Figure 2.1 displays the index page for the Google search engine, it is only signifying a basic internet browser in the context of the overall diagram. Also this browser image will be used throughout the rest of dissertation and the same significance will apply to all diagrams)

Top Tier

The top tier is the entry point of the system, typically a client user interface that provides services such as logon, session, data input and display.

Middle Tier

The middle tier is usually a set of software components that provide the processing power to handle events triggered by the top tier user interface. When handling events, the middle tier conducts server side application logic, which could be in the form of business logic execution, file input/output, transactions and connectivity to the bottom tier. The middle tier is flexible in that it provides the ability to add additional software components without disrupting the majority of the underlining code base; therefore providing an extensible system that controls the communication flow between top and bottom tiers.

Bottom Tier

The bottom tier can be recognized as the database management tier; otherwise known as the physical database. This tier provides data consistency and replication to ensure secure interaction with the middle tier. The communication with the database tier usually takes the form of a middle tier database driver or service such as JDBC or EJB respectively.

Architects and developers must make standard design decisions when deciding the interoperability of these technologies within a three tiered architecture, this in turn requires a higher understanding of what technologies drives the platform and the trade-offs involved in applying specific design decisions to a specific application problem [Kassem et al, 2002].

2.3.2 J2EE Web Tier architecture

In terms of web development involving Java, the J2EE platform tries to address many problems that arise in standard development such as:

a) Productivity

Currently, development for the WWW means that a programmer needs to be skilled in multiple technologies, as they not only provide for dynamic content but also transactional database processing, distributed systems and mail clients, etc. Since many different technology bases can be use within system architectures, these architectures can often become convoluted and as a result, system and work productivity diminish [Kassem et al, 2002];

b) Legacy system connectivity

Since most of the data that are used throughout corporations is housed on legacy systems, such as mainframes, it has become a problem for web applications to access and reuse this data. Developers need a common and consistent approach in bridging the gap between their WWW technologies, legacy middle tier and backend services [Kassem et al, 2002];

c) Scalability

In today's climate, users demand instant responses to the GUI based queries, which means WWW applications have to handled multiple requests and scale efficiently in terms of performance [Kassem et al, 2002];

d) Security

When an application consists of many tiers (for example, the three tiered architecture) and software components, it is clearly recognizable that there is a need for an overall security model. That is, there are multiple entry points (top, middle and bottom tiers) throughout the system, which can be utilised by hackers to exploit the system [Kassem et al, 2002].

The J2EE platform offers a suggested architecture solution to these problems called the *Web tier* [Kassem et al, 2002] (see Figure 2.2). The purpose of the *Web tier* is to manage the interaction between its external web clients and an application's business logic. The *Web tier* typically takes an incoming HTTP request and manages the resultant interaction between itself and its business logic (such as EJB or JDBC respectively) to form a result, which is then generated as a HTTP response that will serve particular content (such as HTML or XML) back to the external web clients [Kassem et al, 2002].

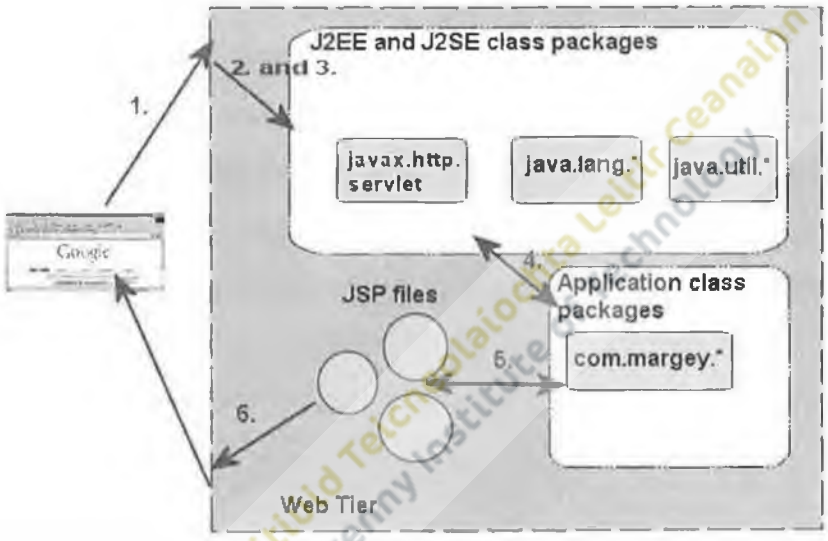


Figure 2.2: Diagram of J2EE Web tier functionality

The *Web tier* architecture offers developers the following functionality (see Figure 2.2):

1. Translates HTTP GET and POST methods so that they can be processed by the business logic classes, that is, the respective back end logical classes that conduct processing for an individual HTTP event;

2. Provides the plumbing to manage the interaction between a HTTP browser and core Java classes;

3. Manages individual user sessions by maintaining state connected with the processing of HTTP requests;

4. Has the option of implementing workflow business logic that is needed to generate dynamic content;

5. Controls the workflow (application flow between different business logic events) which flows between each rendered HTML page;

6. Generates dynamic content to be displayed on a HTTP browser [Kassem et al, 2002].

An application employing the *Web tier* architecture can be implemented using servlets, JSP or a combination of both and built from several suggested J2EE blueprint designs, for example, *page-centric* and Model View Controller (MVC) (a full explanation can be found in section 3.2.2 and 3.2.3 respectively). Although the *Web tier* architecture is essentially composed of one or more of the J2EE WWW based technologies (for example, servlets, JSP or JSP with JavaBeans), it must be realised that there are many more components involved in deploying a Java based web application onto the WWW. Therefore the following subsection will describe these components.

2.3.3 Definition of Web Tier components

The objective of this subsection is to highlight and explain many of the external components that have to be in place for the deployment of JSP. The following definitions of J2EE components and processes will provide clarification on common terms that will be used throughout the rest of this dissertation (see Figure 2.3).

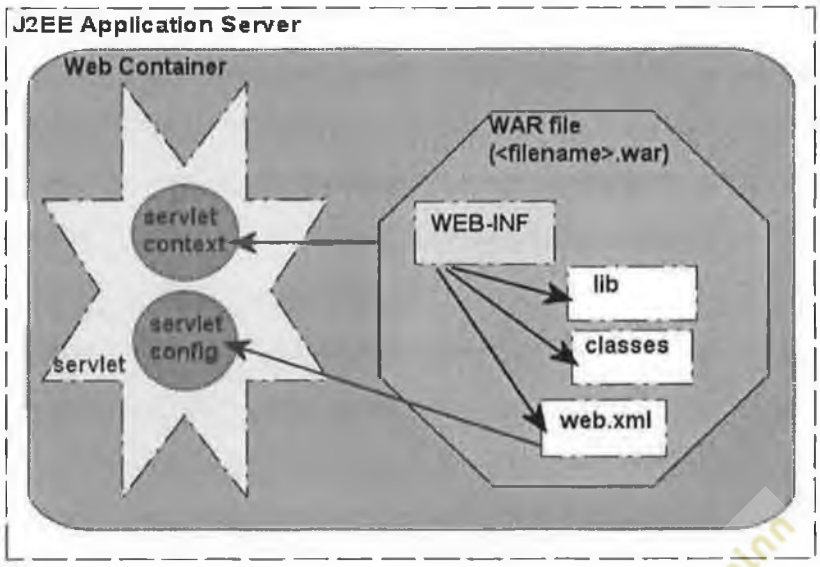


Figure 2.3: Diagram of Web tier architecture

J2EE Application server

An application server is a particular server, for example, Tomcat or Websphere that serves both static HTML and dynamic content through a web container [Sun, 2002];

Web container

A web container is used to serve dynamic content (written in either JSP and/or servlet technologies) as responses to request clients. [Sun, 2002];

WAR file (Web Application aRchive)

A web application archive is a collection/folder of `.html` files, `.jsp` files, images, property files, servlets and applets. These files are then zipped up into a `.war` file (not unlike a `.jar` file) to provide a single web application. Once the WAR file is deployed to a web container; the file unzips and the application servlets / JSPs are initialised and ready to serve to web clients [Sun, 2002];

Servlet context

Once a WAR file is unzipped through the deployment process, a top-level folder is created with a name based on the filename of the WAR file. This folder exists as a context that contains all the servlets and JSPs for a particular web application, and is commonly termed as the servlet context. Also once the WAR file has been unzipped,

a Java object of type `javax.servlet.ServletContext` is instantiated inside the Java Virtual Machine (JVM). Basically this object holds information on what JSP and servlets exist for a web application;

Servlet config

The *Servlet config* is a Java object of type `javax.servlet.ServletConfig` that initialises once your WAR file is deployed on the web container. This object contains application specific configuration information, which is read from the `web.xml` file. Basically this object contains all the initialisation information to start up an application servlet context and subsequent servlets and JSP files.

WEB-INF

`WEB-INF` is a directory that exists inside every WAR file. This folder contains a `web.xml` file that provides the Java object of type `javax.servlet.ServletConfig` initialisation parameters in the form of a Java object of type `javax.servlet.ServletConfig`.

`web.xml` (Web application deployment descriptor)

The `web.xml` file, which is stored under the `WEB-INF` directory, is a listing of servlet context's servlet or JSP information and their initialisation parameters. Once the WAR file is deployed, the XML file is read into the web container and any specified servlets will be loaded. It can also contain information on the *servlet context*, for example, session timeout, welcome page etc.

An overview of some of the XML tags contained in the `web.xml` is as follows [Goodwill, 2001] (see Figure 2.4):

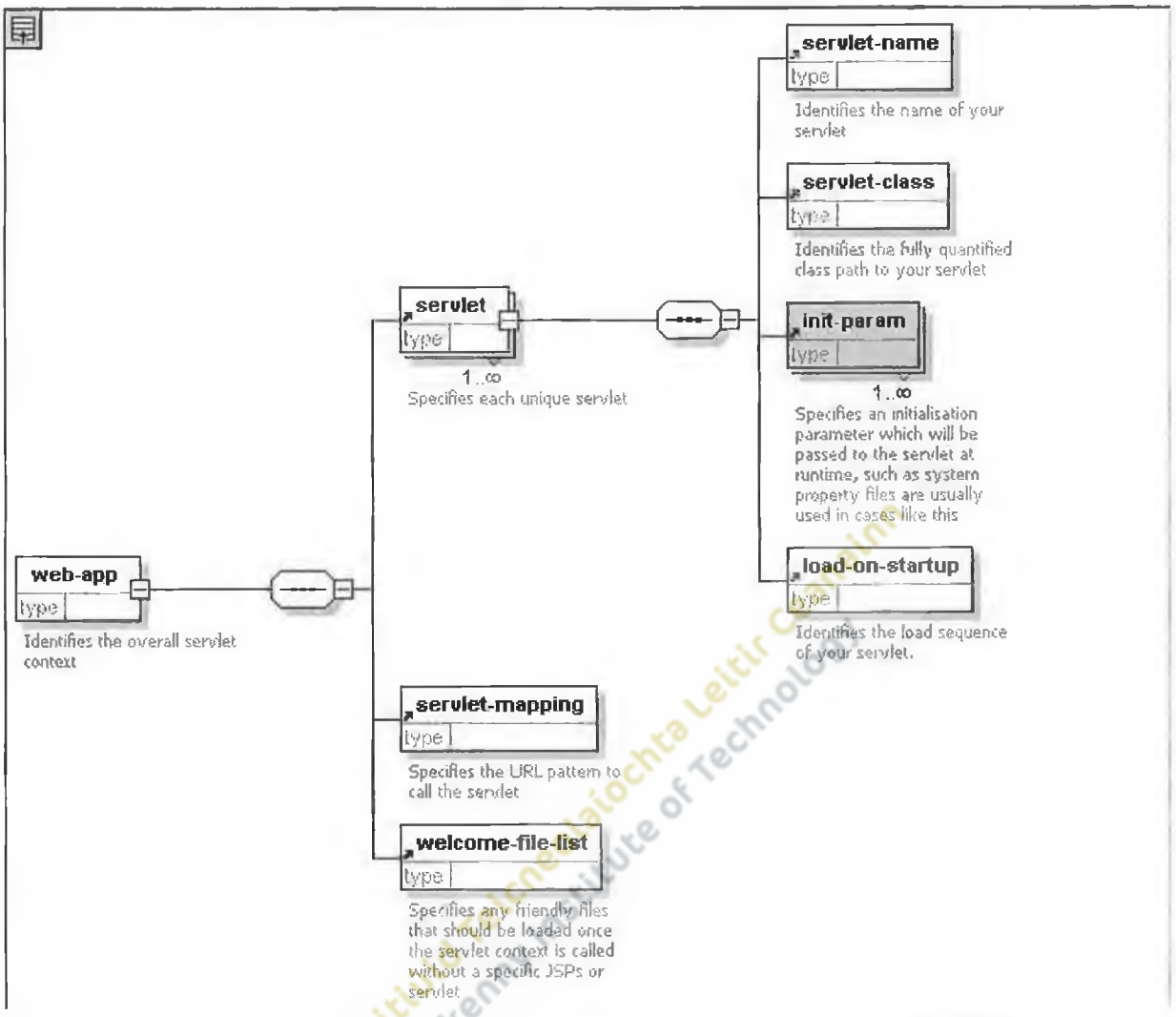


Figure 2.4: Diagram of web.xml file structure (extract taken from XML spy)

For more clear definition, there is an example listing of an actually `web.xml` below.

```
<web-app>
   <servlet>
      <servlet-name>myServlet</servlet-name>
      <servlet-class>com.margey.framework.servlet.Java_DispatcherServlet</servlet-
class>
      <init-param>
         <param-name>PROPERTY_FILENAME_AND_PATH</param-name>
         <param-value>C:\\propertyFiles\\Config\\Start_Up.properties</param-value>
      </init-param>
      <load-on-startup>1</load-on-startup>
   </servlet>
   <servlet-mapping>
      <servlet-name>myServlet</servlet-name>
      <url-pattern>/s</url-pattern>
   </servlet-mapping>
   <welcome-file-list>
      <welcome-file>index.jsp</welcome-file>
   </welcome-file-list>
</web-app>
```

`lib` directory

The `lib` directory, which is stored under the `WEB-INF` directory, is a directory which contains any and all `JAR` files (`.jar` file extension) that are directly related to running of the web application. Once the `WAR` file is deployed these `.jar` files are loaded into memory and live under the web application servlet context. Typically these `.jar` files would store Database Drivers (MySQL, Sybase) JavaMail, XML and/or the direct web application Java source and compiled runtime code.

`classes` directory

The `classes` directory, which is stored under the `WEB-INF` directory, is a directory with can contain all your web application's compiled Java code, for example, JavaBeans and Java classes, that are needed in the successful running of an web application's servlet and JSPs. Typically this method is used if the developer does not wish to contain their code in a `JAR` file. Once the `WAR` file is deployed these class files are loaded into memory and live under the web application servlet context.

2.3.4 Conclusions

This subsection has offered an insight into the role of JSP technology in terms of the J2EE *Web tier* architecture. The subsection has also defined the context of where JSP belongs in the physical makeup of a J2EE web application. Therefore the origins of JSP (That is, its parent technology Java servlets) must now be investigated and discussed.

2.4 Java servlets

As mentioned in section 2.1, JSP extends the Java servlet architecture therefore we must digress and investigate the origins of JSP in the form of Java servlets.

Since the introduction of the Java language, a core feature of the language called Java applets were used to promote the overall development capabilities of the language. A Java applet is client-side program; which is downloaded from an HTTP server via the WWW onto a client's browser. Once downloaded the applet executes on the client local machine to perform some action. The applets of yesteryear were normally heavy graphic oriented presentations or utilities. However since the WWW changed primary

focus from being a research tool to a business medium, it was noticed that these client side programs offered little regard to solving enterprise business problems, such as online banking, insurance and shopping. So a new strategy was formed to counteract the Personal Computer (PC) client side program execution that applets offered. The new strategy was a fundamental change back to application server side program execution and in terms of Java technology it was called Java servlets.

2.4.1 What are Java servlets?

Java servlets are a server side mechanism for executing business logic with the view of displaying dynamic HTML via the WWW. Java servlets provide a simple and robust API (that is, the API consists of around 10 classes and 10 interfaces) that supports HTTP Protocol requests and responses.

Servlets are programs that run on a J2EE application server's web container, their primary function is to deal with incoming client browser HTTP GET/POST requests and generate an appropriate HTTP response, which contains specific content such as HTML or XML.

2.4.2 The Servlet Hierarchy

The server side servlet programs that traffic HTTP request and response between client PC and application server are made up of a architecture that contains three distinct components (see Figure 2.5):

`Servlet Interface`

This Java interface provides a contract to all other servlets, it tells any newly developed servlet that implements this interface that they must implement an `init()`, `destroy()` and `service()` methods (a full explanation can be found in section 2.3.3). All servlets developed by programmers will implement this interface directly, or alternatively the most common way is to inherit from a Java class that implements the contract suggested by the Servlet interface [Goodwill, 2000], [Sun, 2001].

`GenericServlet class`

This is an abstract Java class that provides a developer with an implementation for most of the contractual methods supplied by the Java Servlet Interface. However it doesn't provide an implementation of the `service()` method, therefore if class inheritance occurs then an enforcement of this particular method is needed to run a Servlet [Goodwill, 2000], [Sun, 2001].

`HttpServlet class`

This is an abstract Java class that is most commonly inherited from when a programmer develops a Java servlet. It provides an implementation of all the methods suggested from the Servlet interface and offers methods such as `doPost()` and `doGet()` for dealing with HTTP POST and GET requests. A programmer can provide their own processing of these HTTP requests by overriding these methods [Goodwill, 2000], [Sun, 2001].

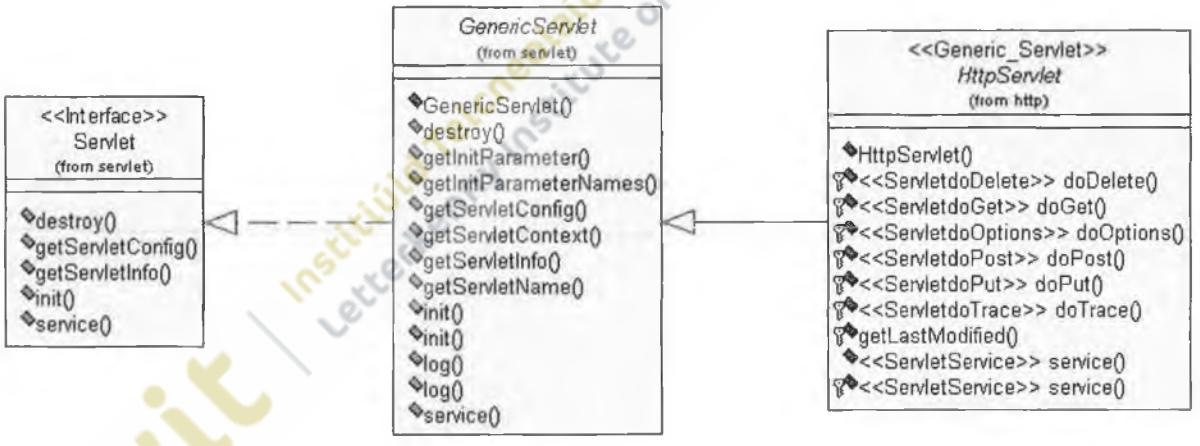


Figure 2.5: UML class diagram of the servlet hierarchy

2.4.3 The Servlet Lifecycle

The lifecycle of a Java servlet is quite simple. A servlet is loaded once and then persists in memory; once loading has completed the servlet initialises any and all specific system resources. From this point the servlet then services incoming HTTP requests and then performs it own house keeping (that is, any unnecessary system resources / process are shutdown). [Sun, 2003] [Zeiger, 1999] [MageLang, 1999]

We shall now look at the significant methods of the servlets in more detail (see Figure 2.6).

`public void init(javax.servlet.ServletConfig servletConfig) throws javax.servlet.ServletException`

The `init()` method is where a servlet's life begins. This method is invoked straight after servlet object instantiation. The `init()` method is used to initialize any system resources such as CORBA, RMI and JDBC, which the servlet will need when processing incoming HTTP requests. The method input parameter is of type `javax.servlet.ServletConfig` interface, this object provides information on the servlet that has be gathered from the deployed WAR file's `web.xml` file. [Goodwill, 2000] [Hunter et al, 1998];

`public void service(ServletRequest req, ServletResponse res)throws ServletException, IOException`

This method handles all the incoming HTTP requests by determining the request type (HTTP GET or POST) and calling the additional appropriate servlet method, such as `doPost()` or `doGet()`. This method input parameters are of type `javax.servlet.ServletRequest` interface (provides information supplied by a client) and a `javax.servlet.ServletResponse` interface (offers a response back to the client) [Goodwill, 2000] [Hunter et al, 1998];

`protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, java.io.IOException`

This method handles all HTTP GET requests, which mean browser based URL queries. The method input parameters are of type `javax.servlet.http.HttpServletRequest` interface (provides HTTP information supplied by a client) and `javax.servlet.http.HttpServletResponse` interface (offers a HTTP response back to the client) [Hunter et al, 1998];

`protected void doPost(HttpServletRequest servletRequest, HttpServletResponse servletResponse) throws ServletException, java.io.IOException`

This method handles all HTTP POST requests, for example HTML Form based queries. The method input parameters are of type `javax.servlet.ServletRequest` interface (provides HTTP information supplied by a client) and `javax.servlet.ServletResponse` interface (offers a HTTP response back to the client) [Hunter et al, 1998];

`public void destroy()`

This method is invoked when it is time to end the servlet life cycle. When an application server is shutting down, it will execute this method. The destroy method should do the exact reverse of the `init()` method, that is, close down any and all system resources that was open by the `init()` method. [Goodwill, 2000] [Hunter et al, 1998].

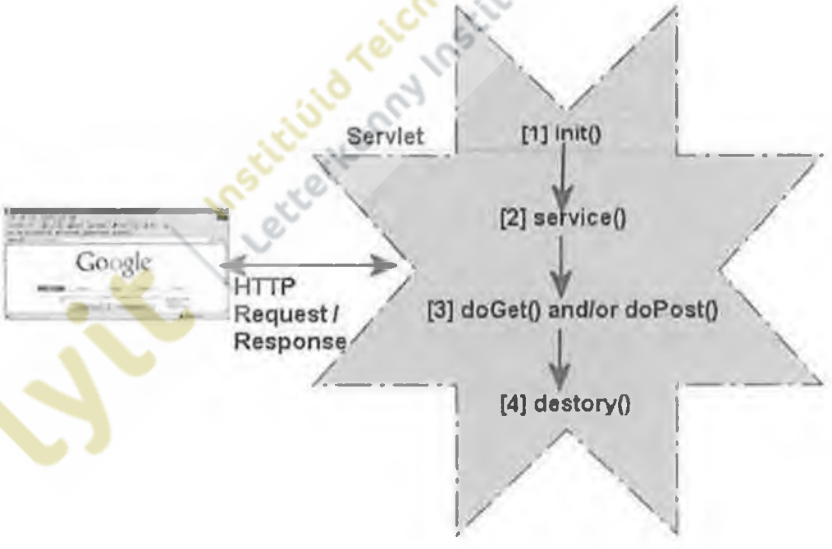


Figure 2.6: Servlet lifecycle diagram

2.4.4 Advantages of servlets over alternative technologies

Some of the key benefits of using the servlet architecture over alternative technologies are as follows:

a) Platform independent

Since servlets are written in the Java language, they are completely platform independent (write once run anywhere). That means a developer can write a servlet on a Windows operating system and run it on a UNIX flavoured platform [Hunter et al, 1998] [Hall, 2002]. However this only is viable if the application server that the servlet is deployed on is implemented in Java (for example, Microsoft's IIS server cannot run servlets as the server was not implemented in Java).

b) Extensive class library support

Servlets have a simplified API (that is, 10 classes and interfaces) however their true power is that they can leverage the extensive Java API. The Java API comes with a huge library of predefined classes that support everything from networking, database and file manipulation etc. Also in recent years Sun Microsystems have added to their existing libraries with J2EE (Java Enterprise Edition) that caters for CORBA, messaging, mail and XML services. Since servlets are a component of the J2EE model, these libraries (Java packages) gives developers programming servlets more power to develop large-scale reusable components that can form enterprise solutions to large corporations for example, purchasing a product, handling credit card facilities etc [Hunter et al, 1998] [Hall, 2002].

c) Application/Web servers

The majority of industry standard application/web servers support JSP (for example, IBM Websphere, Apache/Tomcat web server and BEA weblogic) while its main rivals Microsoft's ActiveServer Pages (ASP) and .NET are currently only supported by IIS.

d) Memory and process efficiency

Once a servlet is deployed onto an application server, it is instantiated as a single Java object in memory. Now when a web client starts sending over HTTP requests for the servlet to handle, the servlet can handle the request straight away, as it does not need to start an interpreter or spawn another operating system process. [Hunter et al, 1998] [Hall, 2002]

e) Endurance

Since a servlet stays loaded in memory, it can maintain state and hold on to external resources such as JDBC database connections, sockets etc which would normally take a few seconds to load. [Hunter et al, 1998] [Hall, 2002]

f) Free to the public.

There is no licensing or cost associated with Java servlets. It is freely available on the web and is supported by most major ISP (Internet Service Providers).

2.4.5 Why is JSP needed?

JSP is needed simply because web developers need something that is easier than servlets to develop; note that many web developers would have difficulty developing robust large scale web applications with servlets because of their inherent problem of intermixing business logic and HTML based presentation. The example below shows how a servlet class was often written.

```
public class HelloWorldServlet extends HttpServlet {

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

         response.setContentType("text/html");
         PrintWriter out = response.getWriter(  );

         out.println("<html>");
         out.println("  <head>");
         out.println("    <title> Hello World Page </title>");
```

```
out.println("  </head>");
out.println("  <body>");
out.println("    <h1>Hello World</h1>");
out.println("  </body>");
out.println("</html>");
}
```


Therefore these convoluted servlets restrict organised development in the following ways:

a) Every web page's HTML content has to be generated through the use of writing excessive amounts of `println()` methods which are associated with Java's `OutputStream` or `PrintWriter` classes. Also every piece of HTML content that required a quotation had to be delimited by a backslash as Java code recognises a quotation as the end of a literal string. For example, a servlet would handle the following code in bold as a literal string and complain of a compilation error.

```
out.println("<form name="form1" method="post"
action="">");
```

Therefore the following code manipulation would have to occur to resolve any literal string problems.

```
out.println("<form name=\"form1\" method=\"post\"
action=\"\">");
```

This action resulted in huge human effort in terms of maintenance through the updating and recompilation of the servlet implementation code [Bergsten, 2003] [Hall, 2002] [Hunter et al, 1998] [Hunter, 2000];

b) Web designer and Java programmer have to work very closely to complete any and all content changes, as both parties did not have the necessary skills to complete each other's work. Therefore this process consumed precious project schedule time [Bergsten, 2003] [Hunter, 2000];

c) Servlets cannot harness the power of web WYSIWYG development tools such as Macromedia Dreamweaver, as a developer must manually embed HTML

into servlet code, therefore creating a development process that is error prone and time-consuming [Bergsten, 2003].

2.5 JavaServer Pages (JSP)

JSP technology is a component of the industry standard J2EE *Web tier* model (see section 2.3.2) and with respect to servlets, JSP can be viewed as a simplified version of the servlet API. The reason for this is that JSP is built upon the existing Java servlet infrastructure and wraps many of the mundane tasks of servlet programming into an API that is easier to use but still offers a developer the full power of the servlets.

Basically JSPs are standalone programs that offer programmers the ability to develop server side Java programs easily, as they overcome the fundamental problems with servlets (see section 2.4.5).That is, JSP are easily maintained and cleanly separate project development roles. Therefore, JSP have become the standard Java solution for producing dynamic HTML. Like JSP's parent technology (servlets), JSP is a platform independent server side scripting language for building robust enterprise standard dynamic websites. However the visual difference between JSP and servlet technology is in the JSP server scripting language (that is, JSP scriptlet).

This server scripting language can be intermixed with HTML to generate a flat text file which is known as a JSP document (denoted by `.jsp` file extension). This JSP scriptlet code provides a mechanism to separate a developer from his program. That is, hiding the developer from writing code that will generate / print HTML tags while performing dynamic actions (which can be costly in both terms of time and maintenance).

2.5.1 How does JSP work?

An explanation of the execution of a traditional JavaServer Page (JSP) is as follows (see Figure 2.7):

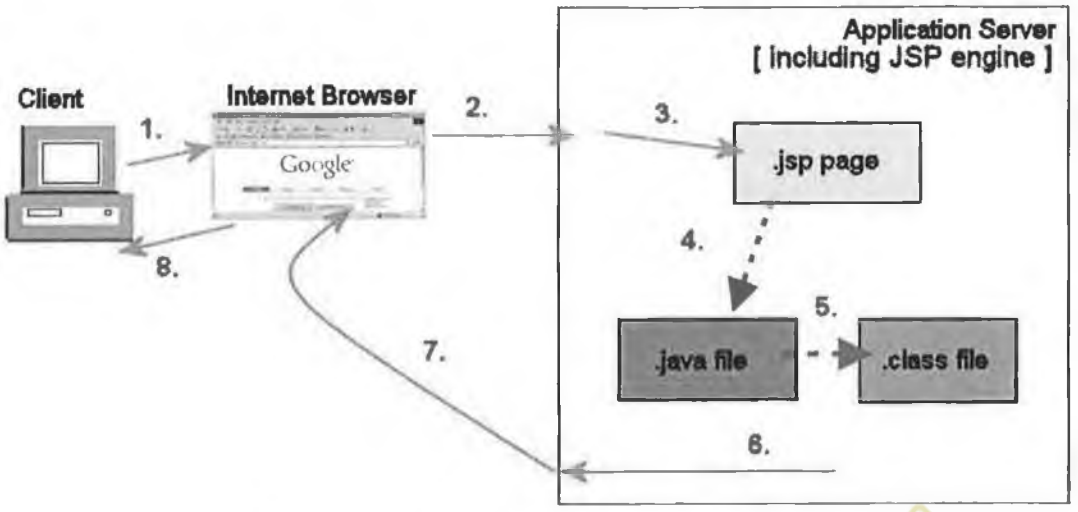


Figure 2.7: Diagram of traditional JSP lifecycle

1. A Client enters a URL into the address bar of their browser;

2. The browser then sends a HTTP Request via GET or POST method to the application server (for example, Apache Tomcat, IBM Websphere etc);

3. The application server now retrieves the requested `.jsp` file;

4. The JSP engine, for example, Apache Tomcat's Jasper parses the `.jsp` file and creates a `.java` source file. The `.java` file will hold generated class code that extends `javax.servlet.http.HttpServlet` and contains the code which will generate the contents to be displayed to the screen (typically the content is made from a combination of Java code and HTML) ;

5. The JSP engine then compiles the `.java` source file into a `.class` file which contains the class's compiled byte code;

6. A JSP engine will then initialise the servlet class into its configuration. The class file is then executed and resultant text (for example, HTML and XML) is created;

7. The resultant text stream is then pulled back to the browser via writing the text to an instance of `ServletOutputStream`. The instance of `ServletOutputStream` can be retrieved from executing the method `getOutputStream()` on a interface type of `javax.servlet.http.HttpServletResponse`;

8. The browser displays the result of the process to the client.

2.5.2 What are the advantages of JSP?

The JSP architecture offers developers many benefits over servlets; some of which are as follows:

a) Reduces development time;
For a JSP program to execute, developers no longer need to consume their time implementing code that inherits from the servlet base class `javax.http.HttpServlet`. Since a JSP file is a combination of HTML and JSP scriptlet code that ultimately will be generated into a servlet (through server parsing and compiling).

b) Reduces development maintenance;
JSP reduces its parent technology (servlet) mundane approach to writing and modifying HTML (for example, writing inline `println` statements into servlets for generating HTML), as JSP auto generates these `println` statements once the `.jsp` file is parsed by a JSP engine.

c) Separation of developer roles;
JSP facilitates the separation of roles within a team context. It clearly defines that roles between graphic designers and developers, that is the ability to work on creative front ends and dynamic areas respectively.

2.6 Conclusions

The subsections covered in this chapter have not only plotted the origins of JSP (that is, from the initial birth of dynamic web technologies, right through to JSP's parent technology servlets), but also discussed in detail the JSP technology itself (that is what it is, right through to its advantages). Therefore since we now know the when, what, where, why, and how of the subject matter, we will now proceed to investigate the wrongs of the technology.

3 JSP Problems

3.1 Introduction

Even though JSP is the standard Java solution for the production of dynamic HTML it does have limitations. Therefore the objective of this section is to highlight and explain some of the fundamental problems that are currently associated with JSP technology. The problems will be discussed in the context of the following areas:

1. Design

   JSP has no standard design approach, this can lead to difficulties with integrating application business logic with JSP script [McLaughlin, 2000] [Altendorf et al, 2002] (a full explanation can be found in section 3.2);

2. Performance
   a. JSP code requires separate interpretation in addition to Java byte code interpretation [Hunter, 2000];
   b. JSP provides no server side caching of dynamic and static content, which leads to increase memory usage from web and application servers [Datta et al, 2002] [Iyengar et al, 2000] [Datta et al, 2002b] (a full explanation can be found in section 3.3.3);
   c. JSP doesn't provide functionality to compress outgoing data [Hall, 2001] (a full explanation can be found in section 3.3.4).

3. Testability

   There is no real unit testing tool at present, only interfaces to existing technologies [Pipka, 2002], which cannot accommodate the existing J2EE architecture [Massol, 2003]; therefore this type of situation encourages the well known affliction of testing, that is, testing is done after the completion of development code [Peeters, 2001]; also JSP error handling provides non-intuitive debug information therefore making the testing process more difficult [Hunter, 2000]; (a full explanation can be found in section 3.4)

4. Security

   Today's application server which host JSP pages have serious internal vulnerabilities that expose the source code in a JSP source file (`.jsp`). For example, since the JSP source file (`.jsp`) exists on the server it can be exploited through hacking [Dimov, 2002] [Raykov, 2002] (a full explanation can be found in section 3.5).

3.2 Design

It has been acknowledged that the current standalone design of JSP (which intermixes HTML and JSP scriptlet code) is not very maintainable or reusable for enterprise solutions, therefore programmers were allowed to impose their concepts of design and in the early days of the technology many programmers encountered the following problematic areas:

a) Programmers started to demand more functionality / services from the JSP / servlet technology. For example, rendering different formats of dynamic content (XML and HTML etc.) [Dai et al, 2000] was incorporated into the technology to make use of the other interoperable Java API's for example, JavaMail, JavaBeans and JDBC;

b) Each programmer has full access to the `javax.servlet.http.HttpServletRequest` object, which causes the following problems [Dai et al, 2000].
   i. They were developing at the low level HTTP protocol;
   ii. Low level programming and business logic became blurred;
   iii. Coding and naming inconsistencies became the norm.

However as JSP gained more recognition within the development community, there has been a debate over what is proper JSP object oriented design. As result, JSP design techniques continued to evolve, now there are many different proposed solutions with their own advantages and disadvantages. These solutions often lead to developer confusion and implementation errors due to inexperience and design complexity.

Before we digress further, a discussion on the composition of a web-based system will be made.

3.2.1 Composition of a traditional web application

Web systems are typically designed into three tiered architectures (a full explanation can be found in section 2.3.1), with the middle tier typically composed of three logical tiers [Kaewkasi et al, 2002] [Altendorf et al, 2002] (see Figure 3.1).

1. Business logic

   The business logic layer has no knowledge of the corresponding workflow (see workflow control in this section) or presentation areas (see presentation layer in this section). The sole purpose is to communicate with external systems (for example, CORBA and Database) and execute logical calculations, such as adding, deleting and updating prices in a shopping cart application or performing file manipulations [McLaughlin, 2002].

2. Presentation layer

   This layer takes the final results of a particular page's business logic processing and displays them in readable formatted text, for example, HTML and XML [McLaughlin, 2002].

3. Workflow control

   This area implements decisional processing based on a user interface decisions that are triggered by a user during their individual session visit, that is a user unique viewing of possible logical workflow within a website. The workflow control handles all incoming HTTP requests in terms of a switching mechanism (*if-else*) and passes them to the business logic layer that will do all necessary page specific processing before passing the results to the presentation layer. The presentation layer in turn builds the page and hands the resultant text back to the workflow control to dispatch as a HTTP Response. The workflow control could be viewed as a multi-channel switch, which directs HTTP requests to the correct area for business logic processing. For

example, a user logging into a system could be directed to a logon error page or the index page of website depending on the choices that they make [McLaughlin, 2002].

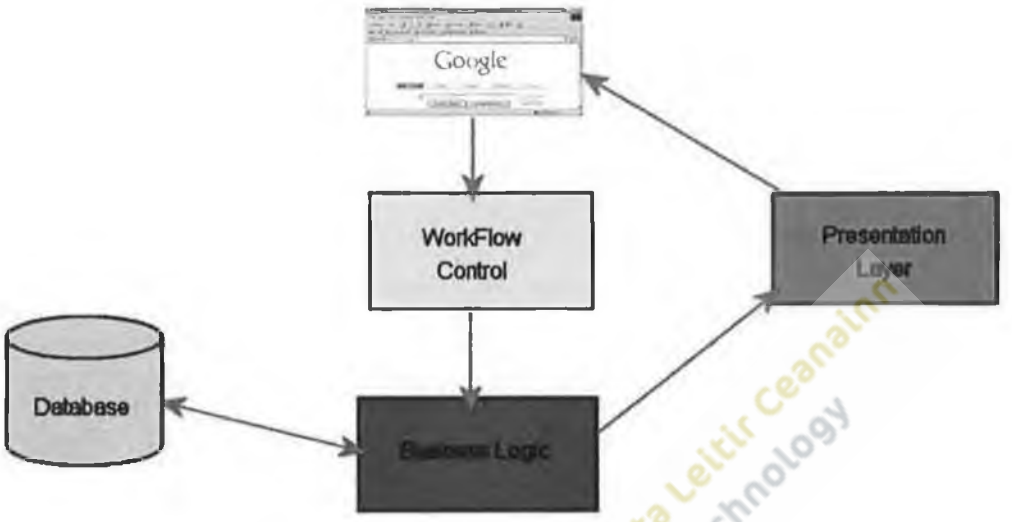


Figure 3.1: Composition view of a web application
(The database in this diagram is an example of an external system, which the business logic communicate with)

In the following sections, two industry standard designs for JSP web-based systems will be discussed along with their inherent problems, these design solutions are *page-centric* and Model View Controller (MVC / Model 2) [Brown et al, 2001] [Kassem et al, 2002].

3.2.2 Page-centric (Model 1)

In this model, the application is built solely from interlinked dynamic web pages, which incorporate the following components into each page:

- Connectivity to external resources. (Database, CORBA services etc.);
- Implementation of model specification;
- Performing calculations;
- Dynamic formatting of results;
- Hard coded hyperlinks.

This approach tightly couples the traditional three-tier architecture (a full explanation can be found in section 2.3.1). It is best suited for small to medium sized web applications [Brown et al, 2001] because the application page flow is usually predefined and the overall structure of the application is simple [Kassem et al, 2002]. The *page-centric* design can be implemented by using the *page-view* or *page-view with bean* design approaches.

3.2.2.1 Page-view

With this approach the JSP page is solely responsible for processing all incoming HTTP requests and offering HTTP responses in return. It combines the business logic, presentation layer and workflow control into one entity, which is the `.jsp` file. The JSP page stands as a single entity that handles, maintains and processes incoming requests, application state, business logic and presentation. This approach often leads to a significant amount of JSP scriptlet code embedded within the JSP page [Hunter, 2000] [Brown et al, 2001].

How does the Page-view design work?

The Page View model works in a JSP web application as follows (see Figure 3.2):

1. The HTTP browser requests a specific user requested JSP page;

2. The JSP page in question loads as Java servlet; once initialized, the servlet / JSP page will then run JSP defined scriptlets which will invoke pure Java objects to fulfill business logic. Once business processing has completed, the dynamic content will be presented as straight HTML;

3. The HTML content is now sent back to the browser as a HTTP response.

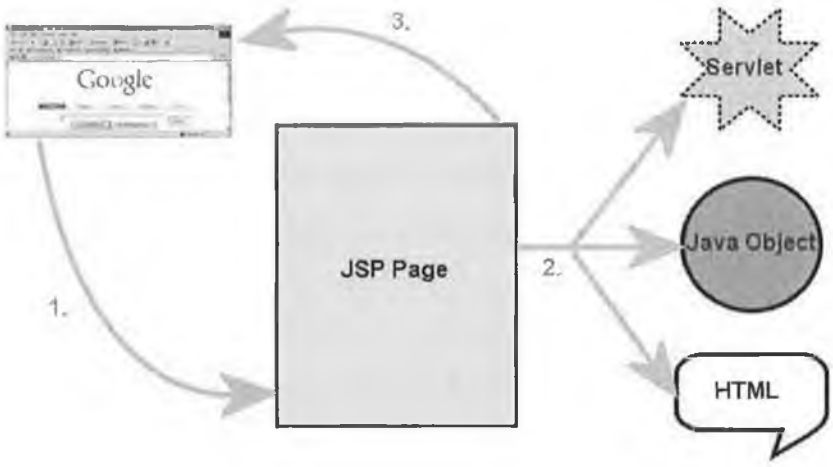


Figure 3.2: Page-view working diagram

3.2.2.2 Page-view with Bean

With this design strategy, an existing Java technology concept was introduced to help reduce the amount of embedded JSP scriptlet code in a JSP page, and in terms of Java, this technology solution was called JavaBeans.

JavaBeans

Basically JavaBeans are Java classes that can be used as the building blocks to form other larger components or full applications [DeSoto, 1997]. A JavaBean is a Java class that fully conforms to the JavaBeans specification. The specification states that three simple rules must be adhered to by any Java class before the class can become a JavaBean (that is, a portable, platform-independent software component model [Sun, 1997]). The following three rules are:

a) The class must implement the interface `java.io.Serializable`.
   Upon object instantiation, the realisation of the `Serializable` interface permits a class to compose itself into a streams of bytes [Hall, 2001] [Johnson, 1997] [Sun, 1997]. The stream of bytes offers an `Serializable` object with the following functionality:
   i. The bytes can be transmitted over a network via socket calls;
   ii. The bytes can be saved to a hard disk via a flat file, this actions allows the present state of the object to be stored for later restoration. That is, the object state can be used as a session variable.

b) The class must implement a no-argument constructor.
The following rule must be implemented because when an external technology such as JSP wishes to instantiate a JavaBean, a process called introspection is performed. Introspection is a runtime process that determines the methods, properties and constructor of a given bean. This process makes heavy use of `java.lang.reflect` reflection mechanism and a number of JavaBeans naming conventions [Flanagan, 1999] [Sun, 1997]. Therefore during instantiation of a bean, introspection will take the class type name of the bean and through reflection, object creation will occur by using the non-argument constructor [Hall, 2001];

c) A class must provide *getter* and *setter* methods to access its properties.
JavaBean properties (attributes) must be implemented as private instance variables, therefore to gain public access to these variables a class accessor (getter) and mutator (setter) methods must be implemented. These method names must adhere to a particular naming convention, which states that each method name mimics the property name with the get or set prefixed to it [Brown et al, 2001]. Also the initial character of the property name in the method name must be uppercase. For example, if a JavaBean called `Person` contained one property called `name`, then the JavaBean methods would be `getName()` and `setName()`.

```
public class Person implements java.io.Serializable{
      private String name;

      public Person(){
      }

      public String getName(){
            return this.name;
      }


      public void setName(String newName){
            this.name = newName;
      }
}
```

The reason for following these naming conventions is simple. Through the use

of Java *introspection*, a list of properties supported by the JavaBean can be determined by scanning the class for methods that have the right names and signatures to be `getXXX` and `setXXX` property methods [Brown et al, 2001] [Sun, 1997].

The introduction of the new JavaBean entity causes a significant intuitive design change from the previous stated *page-centric* design called "Page View" (a full explanation can be found in section 3.2.2.1), as most if not all of the business logic from each JSP page entity is removed and placed into JavaBeans. This offers a clearer design by defining a clearer separation of presentation from content [Brown et al, 2001] [Pipka, 2002].

The architecture works on the basis that the JSP file will now be responsible for the workflow, maintaining state and rendering presentation while delegating all business logic to its companion JavaBeans. The Beans will then act out all calculations and interface with external resources and then return the results to the JSP page for dynamic formatting.

How does the Page View with Bean design work?

The *Page View with Bean* model works in a JSP web application as follows (see Figure 3.3):

1. The HTTP browser requests a specific user requested JSP page;

2. The JSP Page in question loads as Java servlet, once initialized the servlet / JSP page will then run JSP scriptlet code or JSP JavaBean tags which will invoke JavaBeans to fulfill business logic. The JavaBean tags in question are special JSP tags, which use JavaBean *introspection* to instantiate a particular JavaBean (for example, `<jsp:useBean>`) and / or invoke *getter* and *setter* property methods (for example, `<jsp:setProperty>` and `<jsp:getProperty>`) on a particular JavaBean. Once business processing has completed, the dynamic content will be combined with straight HTML;

3. The HTML content is now sent back to the browser as a HTTP response.

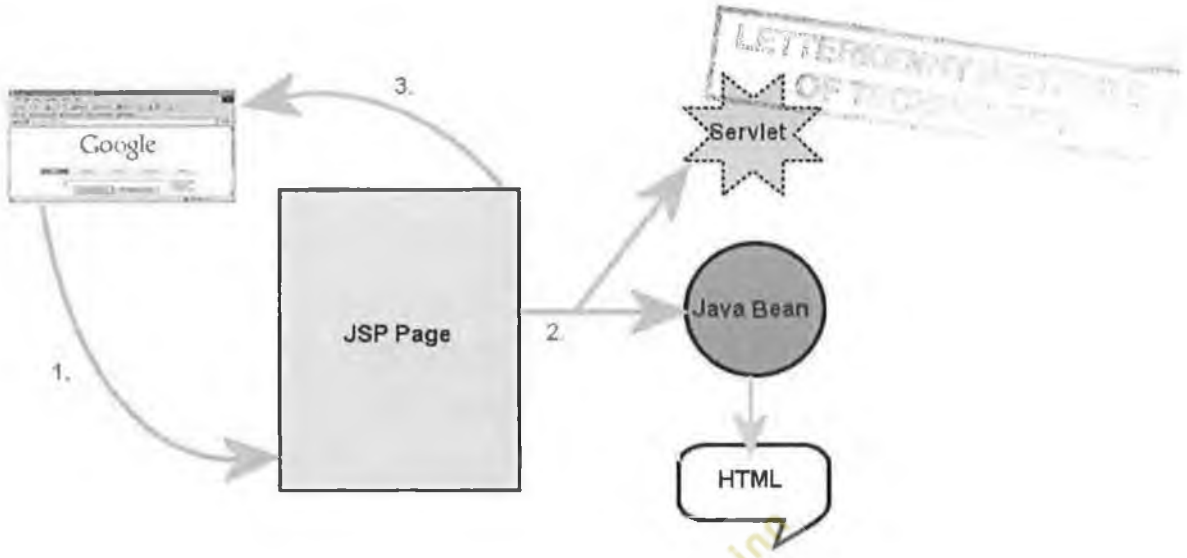


Figure 3.3: Page View with JavaBean working diagram

(Although this figure is very similar to Figure 3.2, there is a subtle different in that the Java object in Figure 3.2 has now become a JavaBean and the JSP page in Figure 3.2 has now moved control of HTML presentation to the JavaBean. That is, the JavaBean reduces the amount of JSP scriptlet code inside the JSP page, which in turn makes the JSP page more readable and maintainable for developers.)

3.2.2.3 Disadvantages with page-centric design

The following are the fundamental problems associate with the *page-centric* approach (That is, both the page-view and page-view with bean approaches):

a) Maintainability.

The degree of maintaining an application built with this approach is enormous. Reusability would basically be non-existent for other applications. Design changes could have major time impact on delivery of code. (SQL table change – could mean an update for all SQL queries in pages). There would be a significant impact on the fundamental intuitive logic that each page represents as the JSP scriptlet code and HTML are firmly blurred [Seshadri, 1999] [Mclaughlin, 2000] [Unger, 2000] [Pipka, 2002] [Kassem et al, 2002].

b) Work flow.

Every single JSP page implemented using a *page-centric* design approach

stands on its own merit, that is, there is no outside influence guiding the overall page to page logical flow for the complete web application. Therefore there is a diminished intuitiveness to these standalone JSPs, since any developer would find the page to page logical flow quite difficult to follow as each page has hard coded links to other dynamic pages. It is not advisable to place a new step in the workflow / logical flow as every page is uniquely tied to each other. That is, specific HTTP request and session variables which are set on a JSP are uniquely used on the following JSP logical flow [Seshadri, 1999] [Hunter, 2000] [Mclaughlin, 2000] [Unger, 2000] [Pipka, 2002] [Kassem et al, 2002] (see Figure 3.4).

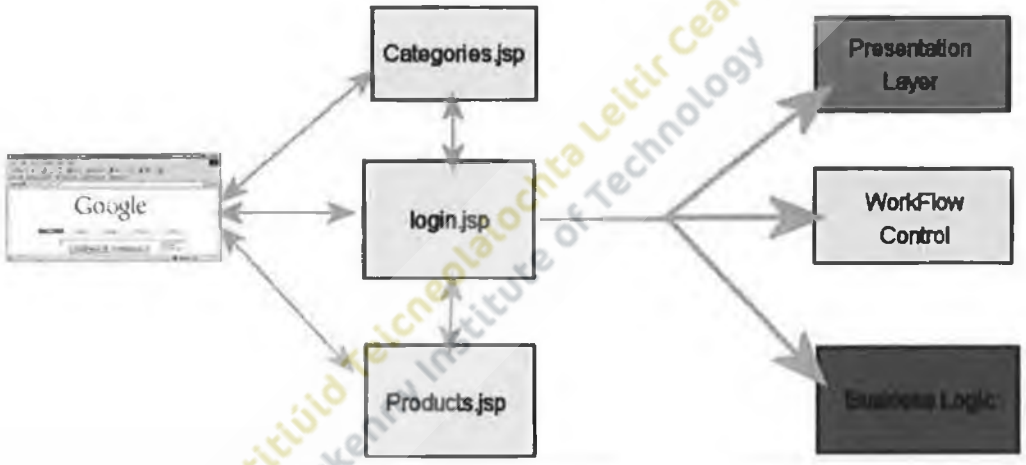


Figure 3.4: JSP workflow complexity

(The `.jsp` pages in this diagram are examples of how JSP communicates with the traditional web application layers.)

3.2.3 Model View Controller (MVC) or Model 2

The core difference between the *page-centric* and MVC design approach is that the responsibility of HTTP request processing has been removed from the JSP file.

The MVC model provides developers with isolated components that are easier to understand and maintain (See Appendix B.2 – Apache Struts framework for

rationale). It is clear separation of an application (be it web or GUI) into three unique parts:

a) Model;
b) View;
c) Controller.

These components are further explained in section 3.2.3.2, we note that this design pattern originated in the Smalltalk-80 system to promote a layered approach to developing graphical user interfaces (GUI) [Fowler, 2003] [Knight et al, 2002]. The MVC is based on the Observer / Observable design pattern (which is the basis of all modern day GUI design).

3.2.3.1 Observer / Observable design pattern

The objective of the Observer / Observable design paradigm is to clearly separate an application's business logic from its presentation view. That is, the design pattern supplies a means where components (both GUI and application driver code) are loosely coupled; therefore promoting component reuse in other applications. This loose binding of components is achieved through indirect referencing of each other (presentation view and application code). For example, the application business logic can be reused in other applications as it loosely coupled from the presentation view. That is, the business logic has no knowledge of what type of view will present its results. Therefore for the presentation view to display the results of a business logic action, it must watch for an event to be triggered by the business logic. Therefore when an event is triggered by the business logic, all subsequent presentation views check to see if the event had any specific meaning and therefore carries out an action. Basically the application data and presentation view do not know that the other exists, but they behave as the do [Gamma et al, 1994].

This pattern is made up of two distinct parts (see Figure 3.5)

a) Observer

   Any class that implements this interface, has a mechanism to update itself once its present viewing observable object state changes [Gamma et al, 1994].

b) Observable/Subject class

This class is unaware of how many observers are watching it, these observers who can attached / detached themselves at any time and are notified when the state of the observable object is changed [Gamma et al, 1994].

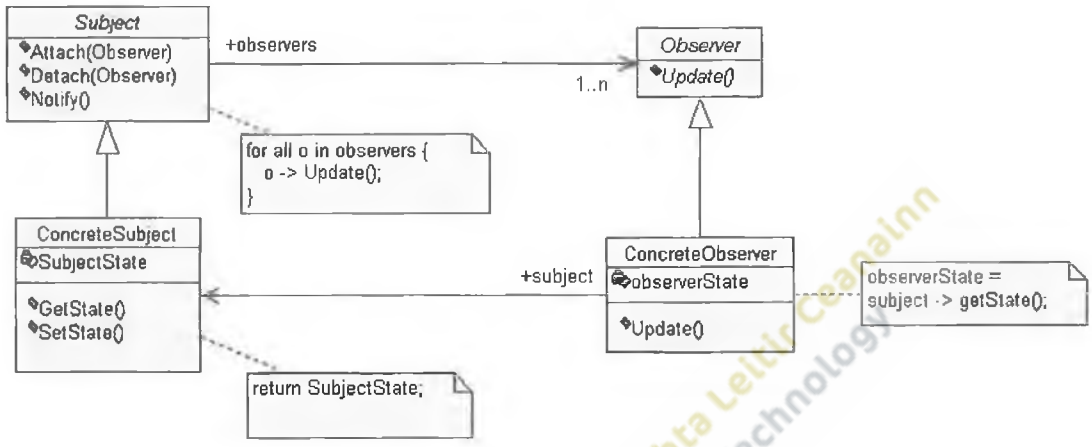


Figure 3.5: Observer Design Pattern [Rose, 2000]

To gain a more real world understanding of the Observer / Observable design pattern the following analogy will be made:

"All of a sudden a man (Observer) from his house window spots (attach method) a particular movie star, for example, Tom Cruise (Observable) walking down a deserted street. Unknown to Tom Cruise that he is in fact being watched, he cries "I am the best movie star in the world" at the top of his voice (notify method). The man laughs to himself (update method) because he realises that Tom Cruise never won an Oscar. The man then watches Tom Cruise exit the deserted street (detach method)".

The MVC builds on the Observer design pattern, in the fact that view components (observable) are clearly separated from their model components (observer).

3.2.3.2 Components of MVC

The following are the fundamental components of the MVC architecture:

a) **Model**

This component deals exclusively with application business logic (that is, the server side logic processing user HTTP request) [Althammer et al, 1999]. The *model* layer suggests that all data objects, for example, JavaBeans will be processed in this layer before handing the results back to the *controller* layer, which in turn directs the results to the appropriate *view* layer. The *model* layer might also touch upon external resources such as JDBC connections, CORBA services and Enterprise Java Beans (EJB). Any objects belonging to this layer should be able to run with a command line driver [Krasner et al, 1988] [Kassem et al, 2002] [Knight et al, 2002].

Using a *model* layer in an application promotes the following:

i) Reusability, since the objects processed in the *model* layer are independent of the *controller* and *view* layers (that is, they should be executable through the command line) then there is no reason for why they cannot be reused in other applications with similar functionality [Kassem et al, 2002];

ii) Separation of developer roles, as a developer working on this layer shouldn't necessary have web development skills [Kassem et al, 2002];

iii) Database portability, since updates to JSP pages containing embedded SQL commands (that is, JSPs using the *page-centric* design) would mean high maintenance costs for a project. Therefore if database querying is not exclusive tied to a particular database, a *model* layer could negate this problem because subsequent changes to objects (for example, changes to embedded SQL, calling new stored procedures and / or the usage of a new JDBC driver) on this layer would not adversely affect the view layer as then are loosely coupled [Kassem et al, 2002].

b) **View**

No business logic is conducted in this layer as its only responsibility is to display presentation items. For example, static and dynamic HTML, Applets and images. Usually JSP takes on this responsibility as it offers developers the opportunity to interact with the *model* layer by requesting information from JavaBeans and then render their pages with static and dynamic content [Krasner et al, 1988] [Kassem et al, 2002] [Knight et al, 2002].

Using a *view* layer in an application promotes the following:

i) Reusability as the *view* components, that is JSP pages can broken into templates with subsections. That is, server side includes which gives developers the ability to use common page elements through an application [Kassem et al, 2002];

ii) Separation of developer roles, as a developer (graphic design) working on this layer shouldn't necessary have Java development skills [Kassem et al, 2002].

c) **Controller**

The controller's only function is to maintain application state and delegate user requests to the appropriate *model* and *view* layers, where request processing and presentation rendering can be made respectively.

Using a *controller* layer in an application promotes the separation of developer roles as the controller acts as an interface to both the *view* and *model* layers therefore decoupling these two components so that graphic designer and developer can work separately [Kassem et al, 2002].

3.2.3.3 How MVC operates in servlet web applications?

The MVC model works in a servlet web application as follows [Ping, 2003] (see Figure 3.6):

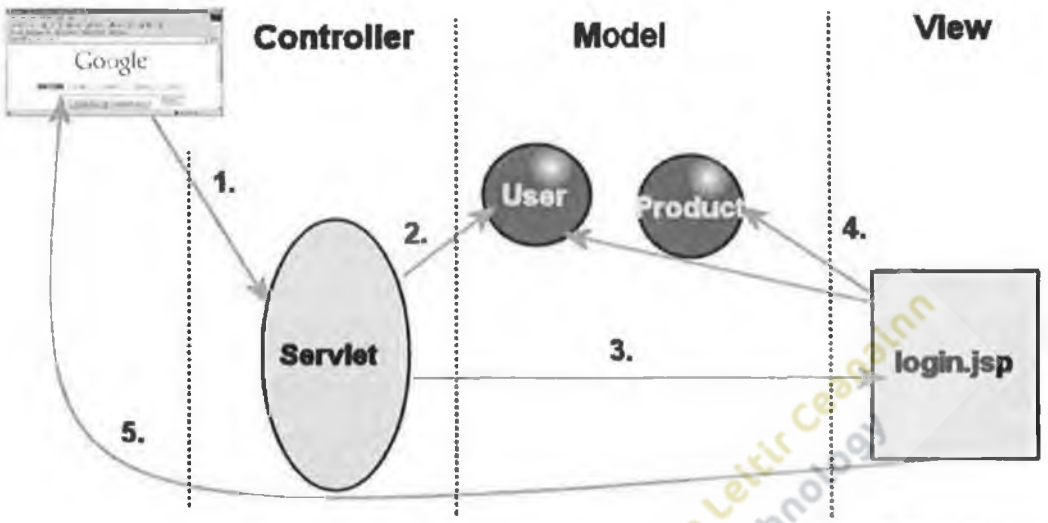


Figure 3.6: MVC working diagram

1. The browser sends a HTTP request to the Controller servlet. The servlet then checks the HTTP request for a specific HTTP field-value string parameter, for example, `nextPage=login`. For clarity, a field-value string is a name-value pair that can attached to any HTTP POST or GET method to signify that a HTML form or URL contains additional information. For example, a logon screen contains two HTML form fields (username and password respectively) therefore once the form has been submitted two field-value strings will be sent using HTTP POST and might contain the values `username=margey` and `password=mypassword` respectively. The value of HTML form field can be gather calling the method `getParameter(String fieldName)` on the interface `HttpServletRequest`. After the servlet retrieves the field-value string, the servlet will then interpret this field-value as a way to direct the HTTP request to the specific page for further processing.

2. Since the servlet has interpreted the field-value string it can redirect to the specific page. However before the redirection is made, the servlet could check the state of certain business *Model* objects (JavaBeans). This would ensure

that application state is consistent before allowing further business logic processing. For example, if a user wished to view/track their current order from a bookstore, then the servlet must check that the user is actually logged in before processing.

3. If application state is consistent, the servlet passes the workflow to the specific page for processing.

4. The page in question will then proceed to build/process business logic objects to update their state and run specific behaviour. The result of this process will then be fused within the page text to form dynamic presentation behaviour.

5. At the end of this process the text be it HTML, XML etc. will then be flushed/sent back to the browser by the means of a HTTP response.

3.2.3.4 Problems with MVC

When properly followed the MVC design pattern enforces a well controlled and structured web application. However it does have the following disadvantages:

a) Unnecessary updates.
A fundamental problem with this design approach is that each component of the *view* layer (`.jsp` file) must update whenever the model (business logic) changes, even if the component doesn't need to update. Basically if a JSP file (*view*) is broken into subcomponents and a *model* object changes then a request will be made to all subcomponents of the JSP page in question [Zhao et al, 2002] [Althammer et al, 2003] .

b) *Model* and *controller* are tightly coupled.
The application logic is not clearly separated between the *controller* and *model* layers, in the sense that the *controller* and *model* still needs to maintain / share session state between each other [Dai et al, 2000] [Unger, 2000].

c) Difficulty.

The MVC design pattern is quite difficult to comprehend and implement for developers who are not well versed in the internals of the Java API. This can lead to a fragile solution that fails to clearly separate the important parts of the system and as a result it is hard to implement and maintain [Althammer et al, 2003] .

3.3 Performance

As highlighted in the previous section, choosing a standard JSP application design (that is, between *page-centric* and MVC) leads to a host of disadvantages, however that is not the only JSP limitation, another substantial limitation is performance.

The main difference between static and dynamic content is server side processing time. Static content is served to a client browser in the following manner. Once an incoming HTTP request is received, the server determines that a particular static page has been requested and processing starts on a server. Then the web server or container finds the aforementioned page and serves it back to the client browser.

While serving static content is a relatively low performance drain on a web container or server, dynamic content is entirely a different matter. JSP performance degrades under the following pressures:

3.3.1 Connectivity to external resources

Depending on the complexity of an application, dynamic content served via JSP can require various amounts of processing. Processing of this sort could take the form of connectivity to various external resources, such as databases or CORBA services etc. which involves significant periods of time to complete therefore reducing response speed to a client's browser [Iyengar et al, 2002].

3.3.2 Thread management of Server Side Includes (SSI)

Once a JSP file is actually parsed and compiled into a single servlet (which occurs during the initial execution of the JSP page), the web container has to manage an individual servlet thread process for each HTTP of the JSP / servlet [Iyengar et al, 2002]. Therefore, using server side includes (that is, the JSP include statement `<%@ include file="<FILENAME>" %>`) to fragmentize a dynamic JSP page with a view to using common fragments throughout a website (for example, page header and footer elements) would result in an increase of servlet thread processes that the web container had to manage [Hunter, 2000] (see Figure 3.7).

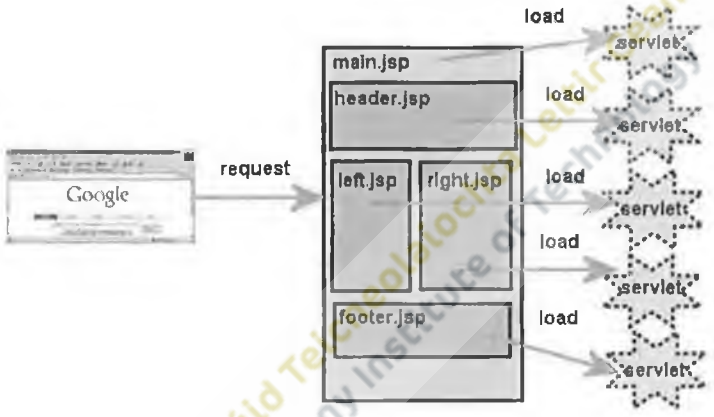


Figure 3.7: JSP include fragment diagram

3.3.3 Caching

Section 3.3.2 suggests a more significant problem in JSP, namely there is no facility to cache server side static or dynamic page fragments. If a programmer had the ability to store server side JSP dynamic or static fragments as a single process in memory or serialized on disk, then the web container would not have to manage these JSP fragments as multiple servlet instances. A JSP caching mechanism would significantly reduce the load on both web server and container and increase the overall performance of an application [Challenger et al, 2000] [Knystautas, 2001].

```
application = pageContext.getServletContext();
config = pageContext.getServletConfig();
session = pageContext.getSession();
out = pageContext.getOut();
```

A possible workaround to this problem (although not feasible in the private sector and very time consuming) is to customise the JSP engine in order to get a `GZIPOutputStream` instead of the `JspWriter`. Developers could do this with at least one JSP Engine (Tomcat) because it has an open source code base.

3.4 Testability

A substantial JSP limitation is in its testability, as developers wishing to perform a line-by-line debug walkthrough of their JSP scriptlet find themselves with a quite taxing task compared to normal Java applications. The reasons for this are actual simple:

a) Pure Java based application GUIs developed using Swing or AWT exclusively deal with native Java objects. Hence the code can be debugged through a traditional integrated development environment (IDE). However JSPs have outside interlinking component variables, which are hard to simulate in an IDE. Examples of these are the HTTP protocol, web browsers, web containers, web servers, session management etc. [Brown et al, 2001] [Dai et al, 2000] [Hieatt et al, 2002];

b) The overall design of a web application consisting of JSP can often lead to an increase in application complexity and developer's confusion since the developers have a wide choice in their implementation methods, for example *page-centric* and MVC designs [Brown et al, 2001];

c) The non-intuitive way in which JSP deals with error handling. When JSPs throw an exception, that exception is based on the generated Java source file as opposed to the JSP file itself, therefore inexperienced developers try to match the error line number to the JSP file and not the actually source Java class file. Basically developers must debug compiled machine code without

using high level language symbols and structures, and this is difficult especially for the inexperienced [Brown et al, 2001] [Hunter, 2000];

d) During the parsing and compilation of a JSP source file (`.jsp` extension) to a Java servlet class, the base JSP scriptlet code is not checked for warnings (for example, of type checked / unchecked exceptions and possible null pointers) and depreciated methods before / during compilation therefore the JSP code is more vulnerable to runtime errors as oppose to native java code [Dudney et al, 2003] [Hunter, 2000];

e) Currently there is a servlet / JSP application server non-standardisation towards reporting JSP errors, as each vendor offers their own interpretation / implementation of a JSP error handling mechanism. Therefore this non-standardisation can lead to programmers spending more time on routine bug fixing and learning the internal workings of a specific application server [Brown et al, 2001].

Many programmers try to overcome these JSP testing problems by using conventional testing methods. These methods can form two categories namely console and IDE based testing.

3.4.1 Console based testing

This method of testing consists of systematically entering Java `System.out.println()` statements throughout a code base, with a view of examining the results when the web application is executed.

It is fair to assume that this method is relatively easy to implement and has the advantage that you don't have to create additional classes in dealing with outputting to the console, however there are significant drawbacks.

a) The code based increases in size and method intuitiveness is lost thought the clutter of `System.out.println()` statements [Dudney et al, 2003];

b) It is laboursome and somewhat mundane process that increases `String` object creation in the application server's Java Virtual Machine (JVM), therefore overall application performance can degrade significantly.

Example:

```
System.out.println(
"[debug info: for counter := " + i + "]");
```

One might think that in theory that the overloaded operator "+" only creates one `String` object where the `String` grows in length, however Java strings are immutable. That is, "+" creates a new `String` object the size of the right `String` plus the left `String`, therefore in reality a third `String` object is created [Sun, 2002b] [Brown et al, 2001];

c) Since the results of the debug messages must be manually examined, incorrect results could be inferred due to human error [Dudney et al, 2003].

3.4.2 IDE debugger based testing and profiling

Currently there is a large choice of tools for debugging and profiling Java applications, which can perform breakpoint code walkthroughs, variable watches and threading support. These tools can speed up development and reduce the number of application bugs found in a production environment. Even though these tools provide major advantages, they can be out of reach from small businesses and students as they are very expensive to purchase.

3.5 Security

The Java language is judged as being secure as it is strongly typed language (that is, in Java every variable or class has a type and therefore during compilation and runtime execution, if the value type and object type do not match then a new value cannot be assign to an object. Thus hackers cannot introduce foreign entities into system which could masquerade themselves as normal entities) and the Java language contains cryptography / security API components. However in terms of the WWW,

application-level security vulnerabilities are inherent in a Web application's code, regardless of the technology in which the application is implemented or the security of the Web server and backend database on which it is built [Scott et al, 2002].

JSP is no exception to the above, since its primary function is to render dynamic web content over the WWW, it is explicitly exposed to many different security problems that fall under two categories namely *application level* and *application server* vulnerabilities.

3.5.1 Application level vulnerabilities

General security vulnerabilities at JSP application level can contribute to two problems, one is third party components; these components might be full of security holes and developers who are integrating them into their systems have no control over their problems. The second general problem is that code is buggy as developers generally overlook the identification of security related code as they are under projects time constraints / commitments [Scott et al, 2002].

The most significant JSP application level security breaches can fall under three methods, that is HTTP form modification, Cross-Site Scripting and JavaBean exploitation.

3.5.1.1 HTTP Form modification

This attack takes the form of saving an outputted dynamic or static HTML page from a browser and manipulating an embedded HTML form before submitting it via the WWW [Dimov, 2002].

For example, a user could have a back account creation form where they enter in their personal details, that are name, address, age etc. However the page could have a hidden form field that states that the overall balance is zero.

```
<form name="form1" method="post" action="http://www.bank.com">
      <table border="1" cellspacing="0" cellpadding="0">
        <tr>
          <td>Name</td>
```

```
        <td><input type="text" name="name"></td>
      </tr>
      <tr>
        <td>Address</td>
        <td><input type="text" name="address"></td>
      </tr>
      <tr>
        <td><input type="submit" name="Submit" value="Submit"></td>
        <td> </td>
      </tr>
    </table>
    <input name="balance" type="hidden" value="0">
  </form>
```

Now if a user saved this output and changed the following HTML text from **`<input name="balance" type="hidden" value="0">`** to **`<input name="balance" type="hidden" value="1000000">`** then the user would be a million pounds richer once he/she submitted the page!

It is extremely difficult to combat this attack, as it requires server side JSP scripting, which is tedious, time-consuming and error prone task that is rarely undertaken in practice [Scott et al, 2002]. For example, instead of client side Javascript validation; the validation is now moved to the server side JSP code base – this means that a HTTP request must be sent to the application server and dealt with there as oppose to using Javascript. Javascript can check for errors before a HTML form is submitted and therefore lessen the amount of HTTP requests that are sent to the application server.

3.5.1.2 Cross-Site Scripting (XSS)

This technique is the most common attack method used by hackers. It is where a hacker wishes to steal a client's details (by manipulating their cookies, which contain passwords and usernames) by embedding malicious JavaScript or HTML into JSP dynamic page generation output.

For example, take the friendly URL `http://www.mysite.com/index.jsp?message=Patrick` which will take name – value pair of message and display the users name on the screen.

A XSS attack could be to embed a malicious URL into the message name – value pair, which when clicked would bring a friendly user to a new website that exposes the user's sensitive information (that is, cookies etc.) [Dimov, 2002] [Klein, 2003] [Scott et al, 2002].

For example,

```
http://www.mysite.com/index.jsp?message=<a
href="http://www.evilsite.com">Patrick</a>
```


3.5.1.3 JavaBean exploitation

As specified in the JSP specification, JSP can modularize certain business logic areas into workable reusable components using JavaBean technology (a full explanation can be found in section 3.2.2.2).

A JavaBean primary function is to provide an encapsulation of data properties and provide easy access to these data properties by using *getter* and *setter* methods [Sun, 2001]. In JSP these *setter* methods can be abbreviated through the use of JSP bean tags, for example `<jsp:setProperty name="JavaBean_Name" property="name"/>`

Instead of a developer using multiple JSP bean tags to set multiple JavaBean properties, a developer can use of the wild card character "*" (for example, `<jsp:setProperty name="JavaBean_Name" property="*"/>`) [Sun, 2001], which provides a shorthand JSP Bean tag notation to set all properties of a JavaBean.

However the usage of the wild card character exposes a large security hole, as there is nothing stopping a user from manipulating a HTTP POST/GET URL (that is, by appending additional name-value pairs) to set additional properties on a JavaBean [Dimov, 2002].

For example, a HTML account setup form (see Figure 3.8) that contains two form fields, say *name* and *address*, will be submitted to an *Account* bean class which

contains three data properties *name*, *address* and *balance*. Therefore upon the HTML form submission, only *name* and *address* properties will be set on the *Account* bean. However if a user appended a name–value pair to the end of the HTML form designated JSP page, for example, form.jsp?balance=1000000. Then they could rightly initialise a user bank account balance to a million pounds as oppose to zero pounds if the JSP code looked like the following.

```
<jsp:useBean id="account" class="AccountBean">
    <jsp:setProperty name=" account" property="*"/>
 <jsp:useBean>
```

Figure 3.8: Account HTML setup form

3.5.2 Application Server vulnerabilities

Like any other software, application server software can be shipped with deficiencies. There are many reported cases of where vendors have shipped their JSP implementations (Tomcat, Websphere etc.) with software bugs in the form of security vulnerabilities.

For example, an early version of Tomcat had a problem in that it exposed a requested JSP file source code by replacing the file extension `.jsp` with `.js%2570`. The problem is that the characters `%25` is an URL encoded "%", and `70` is the hexadecimal value for "p". Thus application server doesn't invoke the JSP page (since the URL does not end in "`.jsp`"), however it does invoke a static version of the file (since the URL ends in "`.js%p`"), which displays the file source code [Dimov, 2002] [Huseby, 2001] [Scott et al, 2002].

Also versions of Tomcat and Websphere had an exploitation of source code problem by appending the default servlet implementation `org.apache.catalina.servlets.DefaultServlet` and `servlet/file/` respectively to the beginning of the requested JSP page. For example, if hackers wish to gain the source code to a JSP file called `index.jsp` then all they had to do was enter the following text as a URL in a browser. `http://www.<websitename>.com/org.apache.catalina.servlets.DefaultServlet/index.jsp` [Rayvok, 2002] or `http://www.<websitename>.com/servlet/file/index.jsp` [Shah et al, 2000].

3.6 Conclusions

The areas highlighted in this chapter, such as design, performance, testability and security have demonstrated the limitations of JSP. The problems outlined should be carefully considered as they could cause lateness, instability and quality degradation within a JSP web development project.

4 Proposed Solution (MagnumServer Pages)

4.1 Introduction

The overall objective of this chapter is to present a new architecture design for developing web applications in Java. We call this new architecture MagnumServer Pages, which will provide solutions to the main fundamental problems that are currently associated with JSP technology, thus the suggested solutions identified in this chapter are organised according to the JSP problems areas identified in section 3. Hence the new architecture will be discussed under the categories of design, performance, testability and security.

4.2 Design

The new architecture is based on an enhancement of the Model-View-Controller (MVC) (see section 3.2.3).

4.2.1 Enhancement of MVC

The proposed design alternative will leverage and enhance the MVC tiers (Model, View and Controller) in the following manner:

a) The Controller servlet will only have one responsibility, that is, to re*model* the HTTP request object as a pure Java object and dispatch it for business processing [Alur et al, 2003] [Ball, 2001];

b) Each programmer does not have access to the `javax.servlet.http.HttpServletRequest` object directly. They are dealing with a pure Java transport request object which means that the following will occur:

   i) They are developing at a high level, where the HTTP protocol has been hidden in favour of a pure Java object which acts as a full request/response mechanism between the *model* and the *view* layers [Alur et al, 2003];

ii) Low level programming and business logic will be clearly defined and separated, that is, the proposed framework will handle all low level aspects of web development (for example, session management) and the business logic can follow a Unified Modelling Language (UML) [Booch et al, 1998] use case format (for example, follow logical business processing steps) [Alur et al, 2003];

iii) The proposed alternative will impose strict rules on coding application solutions. Thus these standards will present a set of guidelines that rule out inconsistencies when developing a web application with JSP (for example, a developer can design and implement their web application using any approach they wish. However this can lead to problematic situations).

Since developers have so many decisions to make in view of JSP object oriented design (see section 3.2), which in turn offer their own problems, for example with respect to JSP design there can be less intuitiveness and tightly coupled layers. The proposed design alternative will present a means to decrease developer's confusion and reduce implementation errors, as it will offer a flexible and intuitive design that experienced developers can use.

In this section a discussion on the proposed design solution for building Java web-based systems will be made. The design will offer plausible solutions over the presented MVC design problems.

4.2.2 Components of alternative MagnumServer Pages design

For the design of MagnumServer Pages, the application will use a *three-tiered* architecture; in turn the middle tier will use a *three-layered* approach. The layers will provide programmers with independent and hidden components that are easy to implement and invoke. Similar to the MVC design pattern, the new design will be separated into three distinct parts that will offer loosely coupled and more intuitive design.

a) **Model**

Firstly, the *model* layer will be completely independent of the HTTP protocol. That is, if the *model* implements the Command design pattern [Gamma et al, 1994], it will only deal with pure Java objects (object creation and setting mutable attributes) and will strictly adhere to the basic and alternative flow of a UML use case [Booch et al, 1998]. It will be perfectly feasible to run the *model* element with a command-line driver. For example, a shopping basket checkout use case could be executed as a separate stand-alone entity.

Using the proposed *model* layer in an application contributes to the following:

i) Promotion of reusable classes as the objects used are common throughout an application and they are loosely coupled since they support the Chain of Responsibility pattern (that is, the *model* layer is not aware of were a request was sent from) [Alur et al, 2003] [Gamma et al, 1994];

ii) Since the *model* represents a use case and it is independent of the HTTP protocol, it offers developers the ability to easily unit test their logical units of work through the use of a flexible test framework such as *Apache JUnit* because it can represent a standalone entity (that is, separate from the *controller* and *view* layers) that can be tested by using a command-line driver [Alur et al, 2003];

iii) The proposed *model* layer will be very easy to implement and to understand, as there are no contributing components such as the HTTP protocol. Therefore other developers may easily pick up another person's *model* unit and continue to work with it with minimal overhead [Gamma et al, 1994];

iv) The *model* layer allows a clear separation of developer roles, as a developer working on this layer will not need web development skills.

b) **View**

Again this layer will be completely independent of its counterparts, that is the

model and *controller*. No business processing will be conducted in this layer, as its sole task is to take a single native Java request object and use it to display dynamic presentation items [Fowler, 2003]. The *view* layer will also have the ability to use any presentation rendering style, for example JSP, XSL, and HTML.

Using a *view* layer in an application promotes the following:

i) The ability to use the best suitable rendering strategy to display results without worrying about using a new Java framework or refactoring code to incorporate a new technology. A developer is free to use a combination of rendering strategies within their application, which offers unlimited opportunities in developing web applications [Alur et al, 2003] [Gamma et al, 1994].

ii) Like the proposed architecture's predecessor (MVC) there is huge scope for reusability. For example, a page broken into page subsections, which reduces the overall implementation time of dynamic pages as these subsections can be reused [Alur et al, 2003].

iii) Separation of developer roles, as a developer (graphic designer) working on this layer shouldn't necessary have Java development skills [Fowler, 2003].

c) **Controller**

It is proposed that this layer should use a thin servlet that acts a single point of entry for an application. The layer only functions are to separate the HTTP protocol from the Java request, dispatch the request for business processing and then delegate the request for appropriate visual rendering [Fowler, 2003].

Using a *controller* layer in an application promotes the following:

i) Through the use of a Factory pattern (that is, a class that can create an abstract class so that it can be perform polymorphic behaviour throughout the rest of an application), the *controller* will cleanly separate the HTTP protocol from an incoming request, which promotes

loosely coupled interaction between the *model* and *view* layers [Fowler, 2003];

ii) The *controller* layer will have a clean internal design for dispatching a pure Java request object. That is, as opposed to the normal MVC decision design mechanism, which uses nested `if else` or `switch` statements. The alternative *controller* layer will incorporate the Dispatcher design pattern, which in turn uses Java reflection to decide how to direct the request to its appropriate *model* unit. Thus eliminating decision code maintenance from an application [Alur et al, 2003] [Ball, 2001] [Fowler, 2003];

iii) At runtime the *controller* will also delegate the request object (after business logic execution) to an appropriate rendering Strategy pattern [Gamma et al, 1994] (that is, JSP, XML etc.). This runtime binding will promote the use of interchangeable presentation styles therefore offering developers with the best possible choice to display results [Alur et al, 2003].

4.2.3 How does the alternative design work at run-time?

The proposed alternative design solution will work in a servlet web application as follows (see Figure 4.1)

1. The browser sends a HTTP request to the *controller* servlet. The servlet first gathers the `javax.servlet.http.HttpServletRequest` object and then checks the HTTP request for a specific field-value string parameter, that is Action=login. The servlet will later interpret this field-value as a way to direct the HTTP request to the specific page for further processing;

2. The servlet now proceeds to grant independence to the object of type `javax.servlet.http.HttpServletRequest` by calling on a Factory [Gamma et al, 1994] to convert the object to a pure Java object. That is, the Factory strips out the parameter values, cookies and bytes from the

object of type `javax.servlet.http.HttpServletRequest` and inserts these values into a native Java object. This action permits loose coupling between the *controller*, *model* and *view* layers;

3. Through the use of the Dispatcher object (which uses reflection) [Ball, 2001] [Cymerman, 1999] [Cymerman, 2000] the servlet now creates a *model* domain object by using the field-value string parameter gathered in step 1. After the *model* domain object creation occurs, the servlet passes the pure Java request object to the *model* for execution (execution follows a UML use case basic and alternative paths) [Fowler, 2003];

4. If application state and *model* business logic are consistent, the servlet passes the *model* unit results (stored in the single pure Java request object) to runtime rendering strategy (JSP, HTML etc.) for page processing. The pure Java object in question will then proceed to be fused within the page text to form dynamic presentation behaviour [Fowler, 2003];

5. At the end of this process the text (be it HTML, XML etc.) will then be flushed/sent back to the browser by the means of an appropriate rendering strategy [Fowler, 2003].

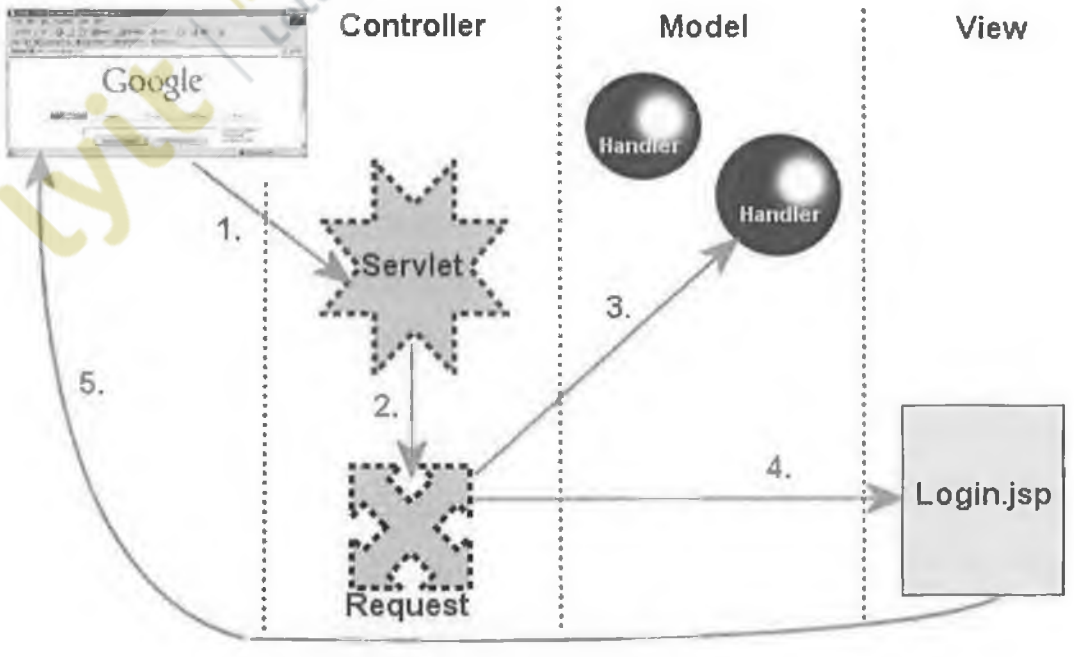


Figure 4.1: Proposed framework design working diagram

4.2.4 Advantages of the new MagnumServer Pages design

The design offers developers the following solutions to the problems associated with the *page-centric* approach (see section 3.2.2):

a) Maintainability

Maintaining an application built with the proposed framework will be very manageable. The design will offer a huge amount of reusability in the sense that existing *model* use case classes can be inherited from, that is the functionality can be extended. Design changes (both visual and logical) will have minimal impact on the delivery of code (that is, text / image changes can be performed by a graphical designer, while changes to database schema would result in a programmer simply updating the SQL in the *model* classes) [Alur et al, 2003].

b) Workflow

Each page will be clearly separated in *model* (use case class) and *view* (JSP, XSL) tiers. Therefore the workflow is quite easy to follow since it adheres very closely to a UML use case. A programmer can easily add and remove workflow steps from an application due to the above *model* - *view* separation and also that the framework hides the low level HTTP request and session variables [Alur et al, 2003] [Gamma et al, 1994].

Furthermore, even though the framework design uses ideas from the MVC design paradigm, one would think the design would encounter the same problems outlined with its predecessor. However that is not the case for the following reasons:

a) Unnecessary updates

During the execution of the *model* use case unit, the new design's native Java request object is filled with the actual results. This object will act as a single *model* results carrier to be fused with the appropriate *view* layer. Since the rendering strategy will interact only with the single pure Java object as opposed to the many JavaBeans in the MVC design (see section 3.2.3), this will reduce the problem of unnecessary updates of the *view* layer (`.jsp` file) whenever the *model* (business logic) changes in the MVC architecture.

b) *Model* and *controller* are loosely coupled

In terms of a web application; the logic will be clearly separated in the context of the *controller* and *model* layers. The *model* and *controller* are clearly independent as they do not share and maintain session state between one another [Alur et al, 2003] [Gamma et al, 1994].

c) Easier to understand and to use

The proposed framework will be very easy to understand and use, as a developer wishing to develop a dynamic web page will have to follow a simplified development process. For example, a developer only has to develop a *model* use case unit (to perform business logic) and its subsequent page (for rendering). The development process uses pure Java based classes with no added HTTP technology layer, therefore simplifying usage for non-web developers [Alur et al, 2003].

4.2.5 Summary

To summarise, lets contrast the traditional MVC design against the suggested alternative MVC design in terms of design (see Table 4.1).

Design Category	Traditional MVC Design	New MVC Design
Protocol	HTTP	None
Controller decisional process	If /else statements Switch statements	Reflection
Controller-*model* dependency	Tightly coupled	Loosely coupled
Application data transfer vehicle	`HttpServletRequest`	Pure Java object (Not tied to servlet API)
Domain *model*	JavaBean	Pure Java object
model-view dependency	Tightly coupled	Loosely coupled
Session management	Manual	Automatic
Rendering strategy	Static	Dynamic

Table 4.1: Design contrast between traditional and alternative MVC architectures

4.3 Performance

As previously discussed in section 3.3, JSP performance is affected by four fundamental problems, namely, (i) connectivity to external resources, (ii) thread management of SSI, (iii) caching and (iv) the lack of compression for HTML content. Although these problems affect overall JSP application performance significantly, they are not insoluble. The following section will outline how the new design proposes to overcome three of these performance problems. One problematic JSP area *caching* is out of scope of this thesis because it is too vast to provide a workable solution.

4.3.1 Connectivity to external resources

Connection to a database via JDBC, for example, can turn out to be a tremendously expensive operation for JSP; it is expensive in terms of both CPU cycles and memory footprint. JDBC connections involve significant set-up, execute and shutdown; all this leads to slower response times and increased server load, which in turn further slows response.

A proposed solution to this problem is to create a database pooling mechanism [Alur et al, 2003]. Upon servlet deployment and initialisation, a substantial number of JDBC connections are created within the pool. These connections are then handed out in a round robin manner to every pure Java request object created by the proposed architectural design approach. A single connection can be later used for each individual execution of a *model* use case unit (business processing) and replaced back into the database pool for later reuse.

4.3.2 Thread management of Server Side Includes (SSI)

Due to JSP ability to fragment common components of a dynamic JSP page, a web container's servlet thread load can increase significantly (that is, including the main JSP, each JSP fragment is a servlet itself). The effect of this can cause the overall performance degradation of a web container. Again, this performance problem cannot be resolved easily, as it is uniquely tied to the overall design of JSP. However a proposed remedy to this problem is the creation of a new Java based dynamic page

technology called MagnumServer Pages (MSP) (see section 5.5 for a full explanation), one where servlet threads are eliminated all together from the opposed technology.

Thus new design will allow for this as the *controller* layer initially strips the HTTP protocol (that is, through the use of Factory class) from the incoming Java request object. Therefore upon creating a dynamic web page, servlet thread activity ceases as MSP instantiates a native Java object (which in turn is maintained by the JVM). This native Java object in turn will take on the servlet's responsibility for building dynamic content. A full explanation can be found in section 5.5.

4.3.3 No provision for compression of HTML content

In section 3.3.4, we outlined how the design of JSP technology was limited in producing compressed data. Therefore we shall be introducing a process to compress the HTML content (that is, we shall retrieve an object of type `java.io.OutputStream` from a `javax.servlet.http.HttpServletResponse` object. This `OutputStream` object will then be wrapped by a `GZIPOutputStream` class, which then writes and flushes the compressed dynamic string back to the browser). Therefore for the new process to work, a new dynamic page technology will be implemented (That is, MSP see section 5.5) to return a full dynamic content string so that it can be compressed (that is, oppose to a JSP page writing the dynamic content string in sizeable segments through usage of its inherent `JspWriter` object - a full explanation can be found in section 5.5).

4.3.4 Summary

To summarise, Table 4.2 contrasts the traditional MVC design (using JSP technology) against the suggested alternative MVC design (using the new server page technology [MSP]).

Performance Category	Traditional MVC Design	New MVC Design
Connectively to external resources	None (must be manually implemented)	Yes (Database pooling built in)
Thread management of Server Side Includes	None (fragmentize JSP creates more servlet threads)	Yes (addition object creation)
Provision for compression of HTML content	None	Yes (Provision built into the overall design)
Caching	None	None

Table 4.2: Performance contrast between traditional and alternative MVC architectures

4.4 Testability

As outlined in section 3.4, there are many reasons that contribute to the overall difficulty in JSP testability. However in this section, a discussion outlining how the new architectural design solves the current problems with JSP testing will be made. These solutions are the following:

a) Currently, JSP interlinking components such as the HTTP protocol are hard to create in an artificial environment, for example an integrated development environment (IDE). Although the current IDEs offer great debugging mechanisms for Java based applications developed using the standard Java application programmer interface (API). The new design provides a suitable non-artificial environment to debug an application's core business logic. The reason for this is simple; the design strips out the HTTP protocol before business logic processing and session management is hidden for the programmer.

b) Debugging JSP scriptlet code is difficult as it is combined with an extra layer of complexity such as HTML and JavaScript. However the new design will make use of MSP (a full explanation can be found in section 5.5). The primary objective of MSP is to perform the similar rendering duties of a JSP page, but without the JSP infrastructure overhead of servlets threads and HTTP protocol.

During compilation, this new technology will take the dynamic page source code (such as the scriptlet code and HTML) and convert it to a native Java class as oppose to JSP's method of converting to a Java servlet class. The native Java class essentially builds a `java.util.StringBuffer` object, which is a composition of appended static strings (HTML, Javascript) with standard Java code execution to form an overall text output in the form of a `java.lang.String` object.

Therefore upon using any standard integrated development environment (IDE), programmers can easily perform a debug walkthrough as they are exclusively dealing with a pure Java object as oppose to JSP's servlet thread with adjacent interlinking components;

c) The new strict design will decrease application complexity and developers confusion, because developers will not have to choose from a particular implementation method, for example *page-centric* (*model* 1) and MVC (*model* 2) designs [Brown et al, 2001]);

d) MSP will offer a more intuitive way in dealing with error handling reporting, opposed to JSP's exception handling which is based on the parsed Java class file from the source `.jsp` file. The new dynamic page technology (MSP) parses its source files into native Java classes; therefore all compilation errors are in the form of native Java API exceptions. Thus inexperienced developers can easily identify the source of the exceptions in contrast to the identification of JSP exceptions.

MSP will need a compiler in the form of a command-line application that runs through the Java virtual machine (JVM). The new compiler grants developers the provision to debug through the program to assess where there are compilation errors in the dynamic page source file;

e) As JSP scriptlet code is not checked during JSP source code compilation, the code is more susceptible to runtime errors. However MSP uses native Java code that is checked during code compilation by the Java virtual machine (JVM). Therefore the JVM is more inclined to indicate problematic runtime errors opposed to the JSP compiler;

f) As stated in section 3.4, there is a no standard JSP error handling reporting amongst application server vendors (Websphere, Tomcat etc.). However as discussed in this section, the new dynamic pages architecture compiles its pages into native Java classes therefore using the all standard reporting power of the Java virtual machine (JVM).

Since the new design offers programmers the ability to produce test friendly code (A developer has an non HTTP environment to debug in and can track compilation and runtime errors more easier). Developers are now more inclined drop some, if not all of JSP's so-called tried and tested methods of testing, such as console and integrated development environment (IDE) based testing, in favour of using regression testing frameworks such as Apache's JUnit.

To summarise, we contrast the traditional MVC design (using JSP technology) against the suggested alternative MVC design (using new server page technology) in terms of testability (see Table 4.3).

Testability Category	Traditional MVC Design	New MVC Design
Error handling	Not transparent, inexperienced developers find it hard to track down origin of JSP error	Transparent, developers find it easier to track down JVM error
Outside interlinking component	Hard to simulate	Clean separation of HTTP, provides for easier testing of components
Warnings / deprecated methods	Not checked	Checked

Table 4.3: Testability contrast between traditional and alternative MVC architectures

4.5 Security

As highlighted previously in section 3.5, current web applications developed using JSP technology are highly vulnerable to security breaches due to many factors such as:

a) JSP technology weaknesses.
   JavaBeans can be infiltrated due to the shorthand JSP Bean tag notation to set all properties, that is through the use of the "*" wildcard character.

b) Applications server implementations.
   Since JSP source code (which is contained in a file with `.jsp` extension) is deployed on an application server for execution. It is more susceptible to exposure from hackers who can break into these application servers. Also many of today's application servers that provide support for JSP (Tomcat, Websphere etc.), are shipped with serious security vulnerabilities that exposes the underling JSP source code.

c) Programmer awareness.
   Due to inexperience and time commitments, web developers often create simple security holes in their code that hackers will exploit in the form of using Cross-Site Scripting and HTTP Form modification.

As security is now a global concern in the WWW community, it is out of scope of this paper to try and resolve every security problem related to JSP. However in terms of building a new architectural design a few safeguards will be proposed.

Firstly, the dangers of deploying JSP source code onto an application server cannot be repaired simply. As the process of deploying JSP source code onto an application server for execution is one that is bound by the technology. However the proposed alternative design will offer a new dynamic page technology (MSP). Since it has been suggested to put forward the idea that this technology will parse its source files into native Java classes (by using a new Java dynamic page compiler). Then it would be reasonable to deploy only the compiled version of the dynamic page class as Java byte

code, which in turn could be added to a collection of other compiled dynamic pages and deployed as a single compressed Java archive (JAR) (`.jar` file extension). The result of this security suggestion is that a hacker would need to go to extraordinary lengths to expose the source code of an individual dynamic page. As they not only have to first break into the application server, but also decompress and open the archived collection (`.jar` file) of compiled page classes and then decompile each individual class. Furthermore, since the new MSP compiler wouldn't reside on the server (as dynamic page compilation would occur before deployment), the hacker would then have great difficulty in reengineering the unformatted text (HTML, Javascript etc.) contained in the `.java` source file as oppose to JSPs formatted text contained in the `.jsp` file.

Secondly, the new architecture suggests the use of native Java classes (HTTP protocol and session management are stripped out or hidden) throughout the design. Then the alternative dynamic page technology (MSP) will offer a non-reliance on JavaBeans in the hope of neutralizing the weakness of using JSP JavaBean tag notation.

To summarise, lets contrast the traditional MVC design (using JSP technology) against the suggested alternative MVC design (using new server page technology) in terms of security (see Table 4.4).

Security Category	Traditional MVC Design	New MVC Design
JavaBean exploitation	Yes	No
Cross-Site Scripting (XSS)	Yes	Yes
HTTP Form Modification	Yes	Yes
Application Server vulnerability	Yes	No (As there is a choice of Rendering strategies)

Table 4.4: Security contrast between traditional and alternative MVC architectures

4.6 Conclusions

The ideas discussed in this chapter have provided a plan for implementing an alternative design framework for resolving the problems surrounding JSP. Once implemented, the highlighted solutions will introduce efficiencies within a Java web development project.

5 Implementation

5.1 Introduction

Section 4 has outlined the architecture of our proposed solution. The following section will discuss the detailed implementation of its components, namely the *model*, the *view*, and the *controller*. The following functionality was highlighted in section 4.2 as the main responsibilities of the system (see Figure 5.1):

a) Instantiation of a thin servlet (*controller*) and setup any necessary configurations via a system properties file.

b) Handle the separation of the HTTP protocol from an incoming request by delegating to a Factory pattern class [Gamma et al, 1994], which in turn creates a plain Java request object that is native to the improved design code base.

c) Dispatch the newly created request object for business logic processing in the *model* layer. The servlet would delegate responsibility for the dispatching process to a Dispatcher pattern class [Fowler, 2003], which in turn loads the correct business logic handler and process the request object.

d) Render the results of business logic processing in the appropriate format by identifying the type of appropriate format and then delegating rendering responsibilities to a chain of responsibility pattern class in the *view* layer [Gamma et al, 1994].

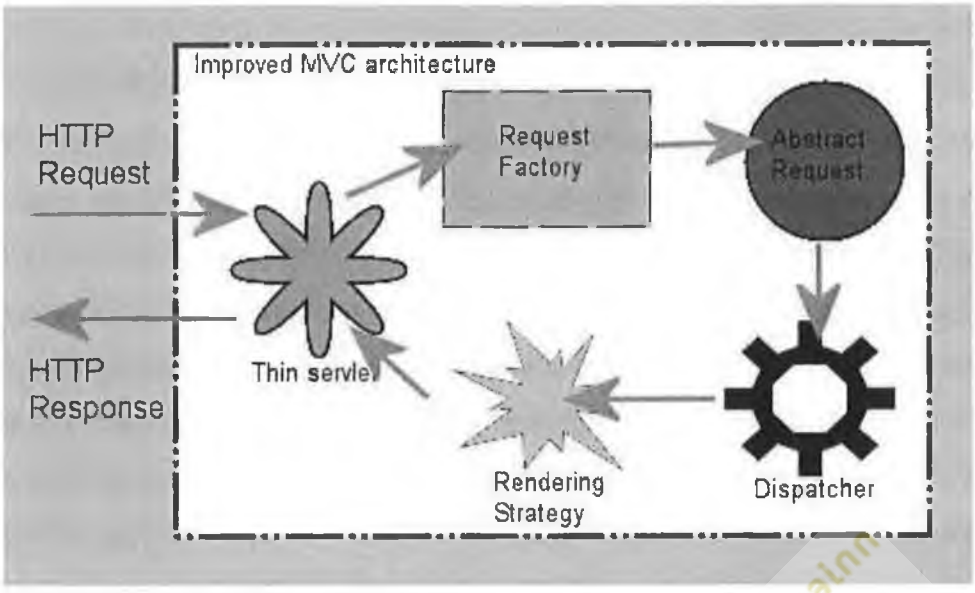


Figure 5.1: Overall functionality diagram

5.2 Controller

In the improved design, it has been proposed that there is a need for a thin *controller* servlet class, which will delegate the following core tasks to other sub components:

a) Separation of the HTTP protocol from an incoming request.

b) Dispatch the newly created request object for business logic processing in the *model* layer.

c) Render the results of business logic processing in the appropriate format by identifying the type of appropriate format.

5.2.1 Composition of controller

The thin *controller* servlet is the main component to the proposed solution as it delegates the processing of HTTP requests to three integral components. These components along with the main servlet can be identified as the following (see Figure 5.2):

a) `Java_DispatcherServlet` (thin *controller* servlet)

b) `RequestFactory` (Factory pattern class)

c) `Dispatcher` (Dispatcher pattern class)

d) `RenderingStrategy` (chain of responsibility interface)

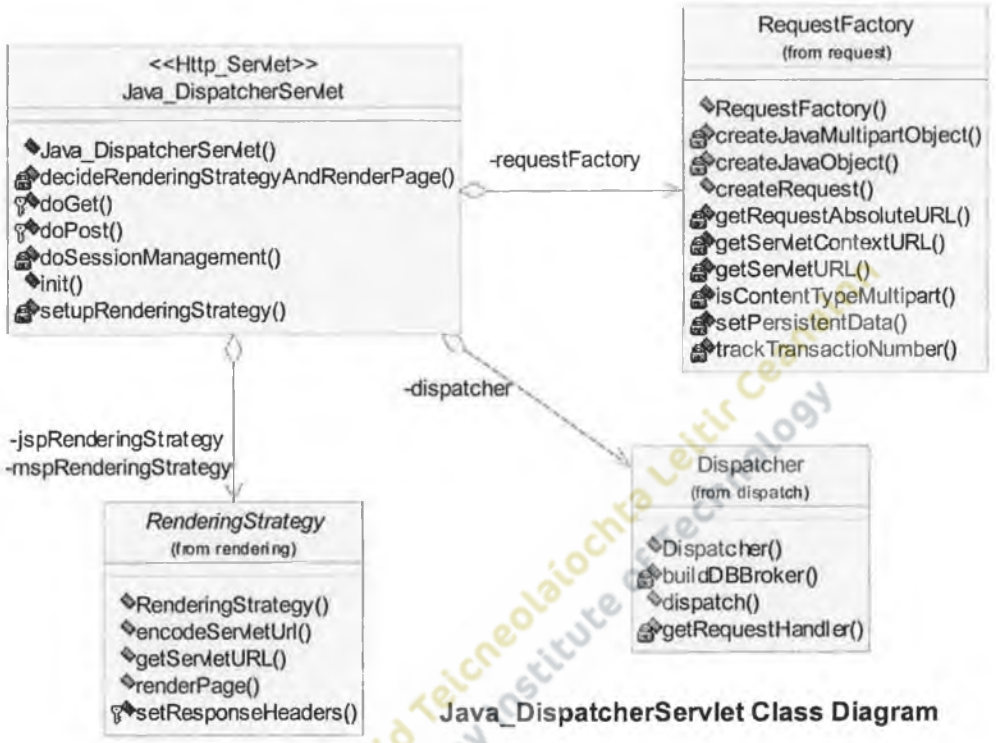


Figure 5.2: UML class diagram of the Controller layer

Since the main components that the *controller* servlet uses to process HTTP requests have been identified, it is now necessary to give an overview of the following method calls that are used within these components:

`init()`

Instantiation method of `Java_DispatcherServlet`, which then creates the three additional components (`RequestFactory`, `Dispatcher` and `RenderingStrategy`) for future delegation of tasks.

`doPost()` and `doGet()`

`Java_DispatcherServlet` method to handle incoming HTTP GET and POST requests.

`createRequest()`

`RequestFactory` method for creating a plain Java request object that is not tied to the HTTP protocol.

`dispatch()`

`Dispatcher` method to identified the proper business logic handler, which in turn processes the plain Java request.

`decideRenderingStrategyAndRenderPage()`

`Java_DispatcherServlet` method to decide the appropriate presentation strategy and pass responsibility to that strategy

`renderPage()`

Any class which implements from `RenderingStrategy` must realise the method to print the final result.

Although the main composition (that is, associated classes and methods) of the *controller* layer has been discussed, there is a need to fully discuss another important entity that is created in *controller* servlet. That is the plain Java object, which is created during HTTP protocol separation.

5.2.2 HTTP protocol separation

Although the normal JSP / servlet architecture uses the implicit request / response objects of type `javax.servlet.http.HttpServletRequest` and `javax.servlet.http.HttpServletResponse` to receive and respond to client demands. It was decided to mimic the functionality of these implicit objects as a single pure Java object in the new implementation. As long as there is a JVM present, developers would have the ability to use a pure fully functional Java request that has no limitations on platform, web application container or even front-end technologies such as JSP, .NET, PHP etc.

5.2.2.1 Composition of HTTP protocol separation

It was decided that the `RequestFactory` instance method `createRequest()` should return an abstract class of base type `AbstractRequest`. The `AbstractRequest` class holds all the state information posted from the browser, any state changes conducted during business logic processing and maintains application state in the session. This is an abstraction of the main primary data transfer container between the *controller* servlet, *model* and *view* layers. This abstract class would enable the new implementation to use polymorphism throughout the code base, which in turn allows the improved design to be scalable and rich in plug-and-play component architecture.

Before the abstract class was implemented, it was analysed that an interface of type `RenderableObject` must be first created. This interface is contractual bound to the `AbstractRequest` class to implement similar `HttpServletRequest` functionality.

An instance that inherits from `AbstractRequest` could be best described as a cross between `javax.http.HttpServletRequest` and a JavaBean. That is, any classes that inherited from `AbstractRequest` would have similar functionality to `HttpServletRequest` and would act as the data transfer container between the *controller*, *model* and *view* layers.

Therefore it was concluded to implement two types of requests; `JavaRequest` and `JavaMultipartRequest`. These classes would handle normal HTTP GET/POST submissions and HTTP file uploads respectively.

Therefore since all associated classes have been created to successfully satisfy the needs of the HTTP protocol separation a composite view is shown (see Figure 5.3).

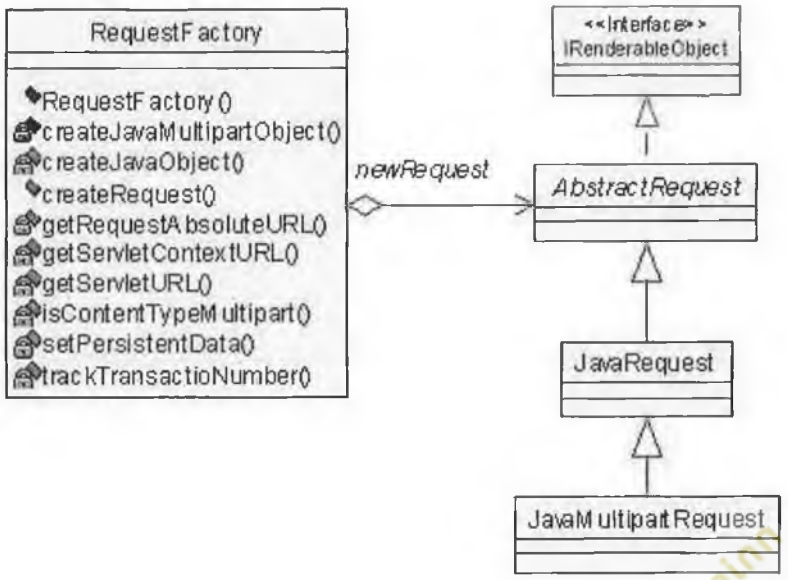


Figure 5.3: UML class diagram of the HTTP protocol separation

5.2.2.2 RequestFactory

Based on the gang of four's Factory design pattern [Gamma et al, 1994]. This class builds polymorphic objects of type `AbstractRequest` (for example, `JavaRequest` and `JavaMultiPartRequest`) by deconstructing objects of `HttpServletRequest`, which are sent via the HTTP Protocol. Ultimately, this Factory class allows developers to discard the `HttpServletRequest` early on in the request process; therefore the improved implementation is less of a reliance on servlet and JSP technology.

5.2.2.3 AbstractRequest

The `AbstractRequest` class responsibilities are the following:

a) Hold the *Action* name-value string pair that identifies which *model* domain object to instantiate later;

b) Hold the *Type* name-value string pair that identifies which *view* rendering strategy to run later;

c) Hold all HTTP POST and/or GET data, which is form fields, cookies, headers, and session objects;

d) Retain persistent data that is submitted via the model layer (business logic processing) so that it is available throughout the lifetime of a client's session;

e) Retain transient data only for the duration of the HTTP POST or GET;

f) Know the next page name that will be rendered.

Therefore to actually fulfill the similar responsibilities of `HttpServletRequest` the following `AbstractRequest` actions have been identified (see Table 5.1):

a) The `AbstractRequest` needs an ability to return a particular HTTP GET or POST value string based on a name string;

b) Returns a particular HTTP GET or POST value string array based on a name string;

c) Retrieve a pure Java object from a web clients overall session;

d) Retrieve all Java objects (in the form of a `java.util.Hashtable`) from a web clients overall session;

e) Retrieve a particular pure Java object from a web clients page session / scope;

f) Removal of a particular pure Java object from a web clients overall session by supplying the object identification string name;

g) Set HTTP GET or POST name-value string pair into the scope (lifetime) of the request;

h) Set an array of `String` objects containing all of the values that the given HTTP GET or POST request parameter has;

Index	AbstractRequest	Java servlet API
(a)	String getFieldValue(String name)	Javax.servlet.ServletRequest getParameter(String name)
(b)	String[] getFieldValues(String name)	Javax.servlet.ServletRequest getParameterValues(String name)
(c)	Object getPersistentObject(String persistentObjectName)	Javax.servlet.http.HttpSession getAttribute(java.lang.String name)
(d)	java.util.Hashtable getPersistentObjects()	N/A
(e)	Object getTransientObject(String transientName)	Javax.servlet.jsp.PageContext getAttribute(java.lang.String name)
(f)	removePersistentObject(Strin g param)	Javax.servlet.http.HttpSession removeAttribute(java.lang.String n ame)
(g)	setFieldValue(String field, String value)	Javax.servlet.ServletRequest setAttribute(java.lang.String name , java.lang.Object o)
(h)	setFieldValues(String field, String[ ] values)	Javax.servlet.ServletRequest setAttribute(java.lang.String name , java.lang.Object o)
(i)	setPersistentObject(String persistentObjectName, Object persistentObject)	Javax.servlet.http.HttpSession setAttribute((java.lang.String nam e, java.lang.Object value)
(j)	setPersistentObjects(java.ut il.Hashtable table)	N/A
(k)	SetRepsonse(javax.servlet.ht tp.HttpServletResponse response)	N/A
(l)	setTransientObject(String transientName, Object obj)	Javax.servlet.jsp.PageContext setAttribute (java.lang.String name, java.lang.Object attribute)

Table 5.1: contrast between new implementation and Java servlet API

i) Placement of a single pure Java object into a web clients overall session;

j) Placement of all Java objects (in the form of a `java.util.Hashtable`) into a web clients overall session;

k) Attachment of an `HttpServletResponse` object for the purpose of using its output stream at a later stage;

l) Placement of a pure Java object into a web clients page session / scope.

To fulfill `AbstractRequest` other duties, that is the containment of the *Action* and *Type* name-value string pairs and the containment of the next page name that will be rendered, the following methods have been identified.

`getAction()`

Retrieves a predefined string *Action* parameter from the submitted HTTP GET or POST.

`getTYPE()`

Retrieves a predefined string *Type* parameter from the submitted HTTP GET or POST.

`getNextPageName()`

Get the next page for the class of base type `AbstractRequest` object to visit.

`setNextPageName(String pageName)`

Set the next page for the class of base type `AbstractRequest` object to visit

5.2.2.4 JavaRequest

This class is an example of a fully implemented data transfer container between the thin *controller* servlet, *model* and *view* layers. The `JavaRequest` extends from the contracted typed `AbstractRequest` (parent-child relationship) thus inheriting all implemented methods. While `JavaRequest` enjoys all the benefits from its parent,

it has also been retrofitted to enable the processing of additional work. For example, `JavaRequest` implements the `javax.servlet.http.HttpServletBindingListener` Interface, which supplies a mechanism to allow an instance of the class to know when it is bound/unbound to an overall `HttpSession`.

5.2.2.5 JavaMultipartRequest

`JavaMultipartRequest` performs exactly the same functionality as its parent `JavaRequest`, however since the servlet / JSP API does not provide any rich mechanism to deal with multipart HTTP requests, that is, HTTP file form uploading. It was best thought that this functionality should be built into the framework to prevent / lessen the workload on developers when developing web page that contain file uploads.

5.2.2.6 Accommodation of other technologies

While the new implementation only demonstrates two fully functional request classes, we cannot rule out the building of other classes. For example, `CGIRequest` (which could model Perl / CGI variables), `PHPRequest` (that could model a PHP request script), `VBRequest` (That could deal with a Visual Basic application front end), or even `DotNetRequest` (see Figure 5.4)

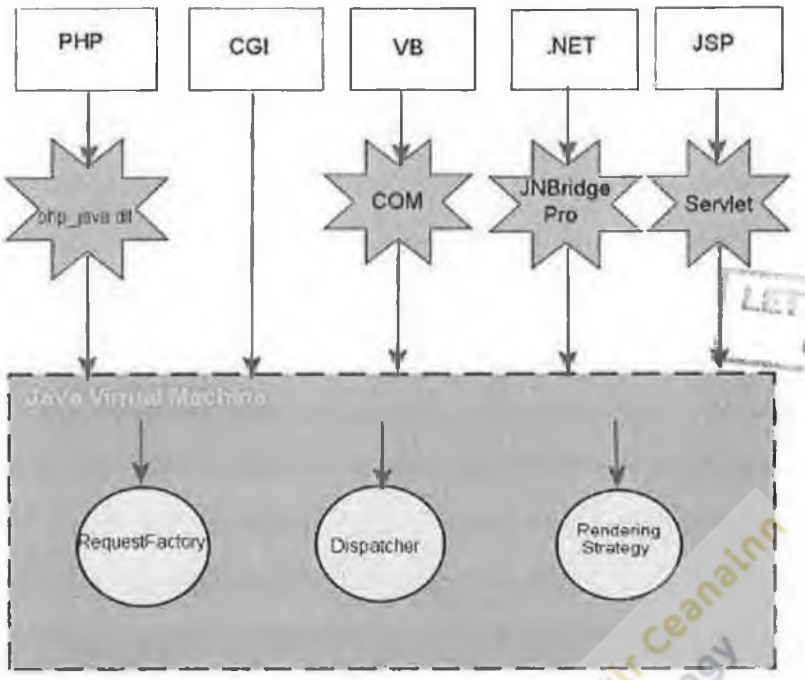


Figure 5.4: Multiple technologies diagram

5.2.3 Summary

To summarise, the following is an outline on how the new implementation will behave in the *controller* layer (see Figure 5.5):

1. Once the thin *controller* servlet is started on an application server, it will load all system properties into the application by reading from a designated property file;

   1.1. After the loading of system properties, the servlet will continue initialisation by instantiating objects of type `Dispatcher`, `RequestFactory` and `RenderingStrategy`;

2. A client's browser submits a HTTP GET / POST request to the *controller* layer;

3. The *controller* layer will delegate responsibility of separating the HTTP protocol from the request to the `RequestFactory` class by invoking the `RequestFactory`'s `createRequest()` method, which in turn creates a

polymorphic request object of type `AbstractRequest` (a full explanation can be found in section 5.2.2);

4. The *controller* layer will delegate responsibility of business logic '*model*' processing of the newly created request object to the `Dispatcher` class by invoking the `Dispatcher's dispatch()` method (a full explanation can be found in section 5.3);

5. After the model has executed its business logic, the *controller* layer will delegate responsibility of rendering the HTML page to the appropriate `RenderingStrategy` class by invoking the servlet's internal decideRenderingStrategyAndRenderPage() method (a full explanation can be found in section 5.4).

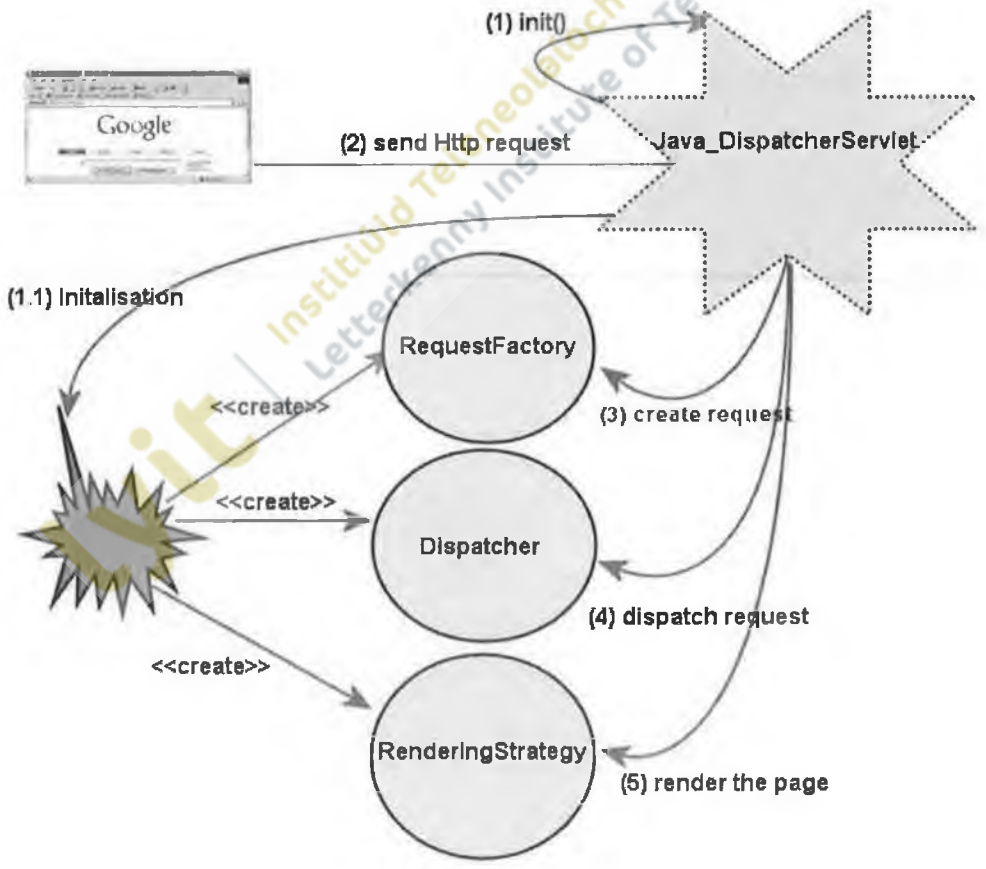


Figure 5.5: *Controller* layer outline behaviour diagram

5.3 Model

As identified in section 5.2.1, the *controller* thin servlet layer will delegate the business logic '*model*' processing of a newly created polymorphic request object (that is, of type `AbstractRequest`) to a Dispatcher pattern class [Ball, 2001] [Fowler, 2003]. Therefore the `Dispatcher` class will need to fulfill the following to satisfy the goals of the *model* layer (a full explanation can be found in section 4.2.2):

a) The *model* layer will be completely independent of the HTTP protocol; which is already implemented as the *model* layer is receiving an request object devoid of the HTTP protocol (`AbstractRequest`);

b) The *model* will represent a UML Use Case basic and alternative flows;

c) The business logic '*model*' processing can be run as a command-line application.

5.3.1 Composition of Model

As a natural consequence of separating the HTTP protocol from the Java request, the `Dispatcher` instance method `dispatch()` will take an input parameter of base type `AbstractRequest`. The `Dispatcher` will dispatch / forward the polymorphic `AbstractRequest` to an appropriate *model* domain object of type `RequestHandler` for processing.

The `RequestHandler` class is an abstraction of the main *model* domain and in turn provides an abstract implementation to handle the request object appropriately. Page specific UML use cases would inherit from `RequestHandler` and override its abstract implementation.

To overcome the JSP performance problem of connectivity to external resources, (for example, a database - a full explanation can be found in section 3.3.1) it was analysed that a database pooling component could be introduced to maximise the speed the business logic processing.

Since all associated classes have been identified to fulfill the characteristics of the improved *model* layer, a composite view is shown (see Figure 5.6).

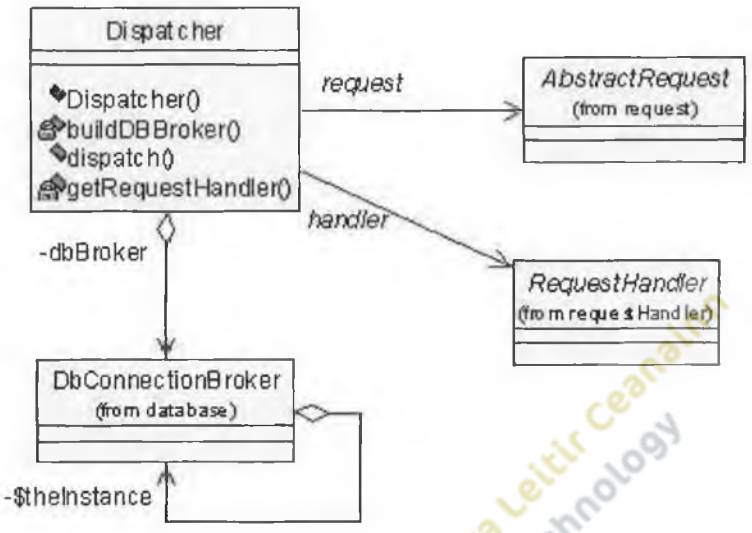


Figure 5.6: UML class diagram of the Model layer

5.3.2 Dispatcher

The `Dispatcher` class main responsibilities are as follows:

a) Initialise all external resources, such as database connection pooling;

b) Dispatch polymorphic request objects of type `AbstractRequest` (for example, `JavaRequest` and `JavaMultiPartRequest`) to *model* domain objects for business logic processing;

c) The `Dispatcher` is completely free (not coupled) to a servlet / JSP environment as it is not tied to the HTTP protocol. Therefore Apache JUnit test suites can be built to test the functionality of the system.

However how does the Dispatcher know which *model* domain object (that is, of type `RequestHandler`) to delegate the business logic processing to? The answer of this question is that the `Dispatcher` class will first call `getAction()` method on the polymorphic request object and then through the use of reflection create a

polymorphic *model* domain object instance (that is, of type `RequestHandler`) [Roschelle, 2000].

To fulfill the responsibilities the following `Dispatcher` methods have been identified:

`dispatch(AbstractRequest request)`

Delegate the incoming `AbstractRequest` to the correct model domain (that is, of type `RequestHandler`) for business logic processing. This is done by invoking the `getAction()` method on `AbstractRequest`, which gathers the *Action* name-value string. Through the use of reflection, this *Action* string is subsequently used to create an instance of base type `RequestHandler`.

`getRequestHandler(String action)`

This method uses reflection to create a the specific type of `RequestHandler` [Cymerman, 1999] [Roschelle, 2000].

5.3.3 RequestHandler

This is an abstract base class that developers will subclass to implement their own *model* layer specific functionality (which follows a UML use case). For example, if a system has a login page then it must also have a subclass of `RequestHandler` called `LoginHandler`, which in turn implements specific page business logic functionality.

This class is modelled on the Command design pattern [Gamma et al, 1994]. This design is well suited when endeavouring to break the normal JSP technology coupling between business logic and page scripting, that is, the relationship between Java beans and JSP script. The use of JavaBeans in JSP technology does not provide a clear distinction between what is business logic and page rendering data. Their primary use is maintaining the behaviour state changes to their class attributes / properties (that is, *getter* and *setter* methods), and not conducting page specific business logic that many miss sighted programmers implement.

However through use of the improved design implementation, the `RequestHandler` class focuses developers on conducting *model* business logic in a black box, which will aid developers in testing, performance and intuitive understanding of particular page *model* domain.

The responsibilities of the `RequestHandler` are as follows:

a) Execute business *model* logic, that is, each sub class of `RequestHandler` is responsible for the execution of a discrete use case (for example, find customer, update customer, get summary, etc);

b) Call out to JDBC Connections or any other external date source to collect/update data;

c) Decide the next page name that will determine the next appropriate *view* / page;

d) Add any persistent or transient objects to the polymorphic request object so that the *view* layer can retrieve data to use in rendering content.

To satisfy the responsibilities of `RequestHandler`, the following method calls have been identified:

`execute(AbstractRequest request)`

Each page subclass of must `RequestHandler` override the following method, with an implementation of specific *model* / business logic processing. For example, a `SearchForProductHandler` class would execute the following:

a) Retrieve the entered search string parameter from the object of type `AbstractRequest`. For example, "DVD = Lord Of the Rings";
b) Retrieve a JDBC `java.sql.Connection`;
c) Build the SQL query string to search the database with;
d) Search the database and build a product entity object. For example, instantiate a `Product` class;

e) Store the `Product` instance as a transient object in `AbstractRequest` for later use in page rendering.

`preExecute(AbstractRequest request)`

This method is called before `execute()`, if the developer wishes then each subclass can choose to override this method . An example use of this method would be to implement retrieval of a commonly used CORBA service;

`postExecute(AbstractRequest request)`

This method is called after `execute()`, if the developer wishes then each subclass can choose to override this method. An example use of this method would be to implement a reassignment of a commonly used CORBA service.

5.3.4 Summary

To summarise, the following is an outline on how the new implementation behaves in the *model* layer (see Figure 5.7):

1) In improved implementation, when a client's browser submits a HTTP GET / POST request to the *controller* layer, the new implementation will create a polymorphic object of base type `AbstractRequest`;

2) The *controller* layer will delegate responsibility of business logic '*model* ' processing of the newly created request object to the `Dispatcher` class by invoking the `Dispatcher`'s `dispatch()` method;

3) The `Dispatcher` will now ask the object of base type `AbstractRequest` for its *Action* parameter (HTTP name-value string pair, for example, Action=Login) by invoking the request object's `getAction()` method;

4) Once the *Action* parameter has been retrieved, the `Dispatcher` invokes an internal method `getRequestHandler()` to fetch the appropriate subclass of `RequestHandler`.

5) The `Dispatcher` will delegate responsibility of business logic '*model*' processing of the request object to the subclass of `RequestHandler` class by invoking the subclass's `execute()` method. As a result, all specific UML use case actions will be executed.

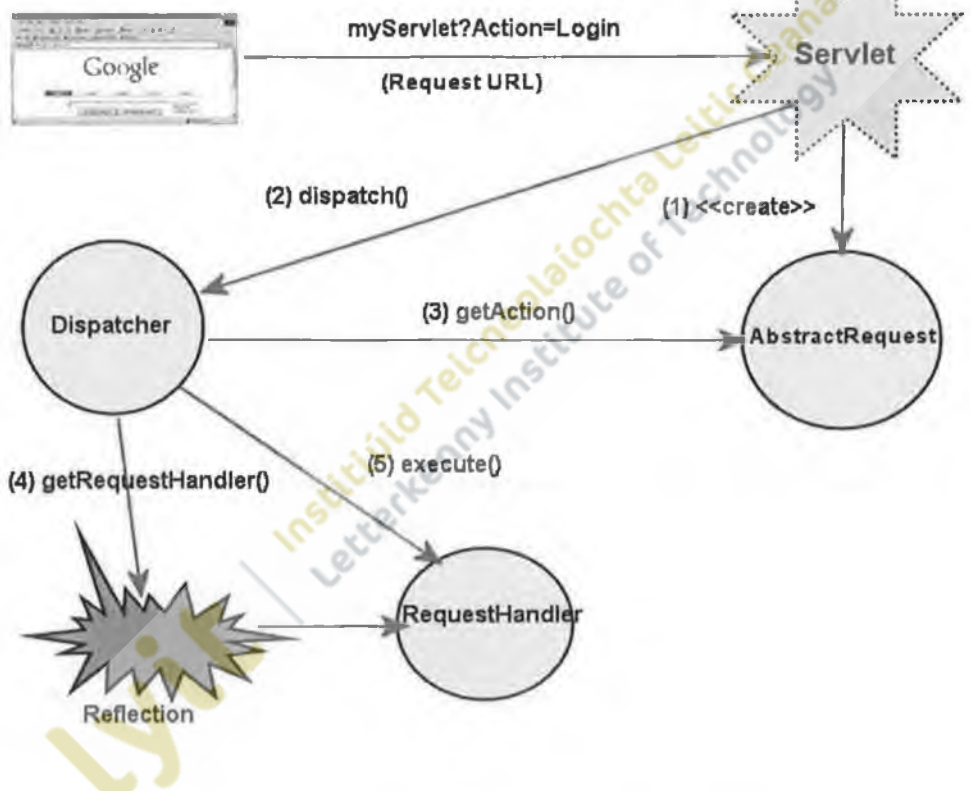


Figure 5.7: *Model* layer outline behaviour diagram

5.4 View

As identified in section 5.2.1, the *controller* thin servlet layer will delegate the results of business logic '*model*' processing (contained in `AbstractRequest`) to a specific rendering approach. Therefore the `RenderingStrategy` class will need to promote the following to satisfy the goals of the *view* layer (a full explanation can be found in section 4.2.2):

a) The new implementation will support multiple rendering approaches, which allows programmers a variety of choice in how they wish to present their data;

b) Separation of development roles, for example one developer can work on the *model* layer while another works on the *view*. Therefore promoting loose coupling between layers.

5.4.1 Composition of View

It was decided that the *controller* servlet will decide the type of rendering approach and forward the polymorphic `AbstractRequest` (which contains results of the *model* layer processing) to an appropriate *view* domain object of base type `RenderingStrategy` for content presentation.

The `RenderingStrategy` class is an abstraction of the *view* domain and in turn provides an abstract implementation to present the request object appropriately. More specific presentation approaches (for example, JSP or XSLT) would inherit from `RenderingStrategy` and override its abstract implementation.

Since all associated classes have been identified to fulfill the characteristics of the improved *view* layer, a composite view is shown (see Figure 5.8).

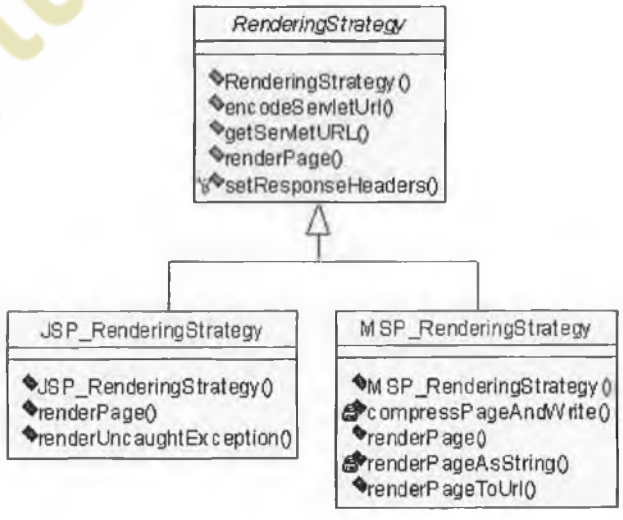


Figure 5.8: UML class diagram of the View layer

5.4.2 RenderingStrategy

This is an abstract base class that developers will subclass to implement their own *view layer* rendering functionality. For example, if a system decides to handle JSP technology then it must implement `JSP_RenderingStrategy` (which is a subclass of `RenderingStrategy`), which in turn implements JSP specific rendering functionality.

The overall responsibilities of the `RenderingStrategy` are as follows:

a) Provide an abstract implementation for presenting the results of the *model* layer;
b) Encode submitted servlet URL. This is a necessary step in the event that session tracking is done via URL rewriting (That is, URL rewriting occurs when a session created within a browser that has cookies turned off);
c) Retrieve a fully quantified submitted URL.

To fulfill the responsibilities of `RenderingStrategy`, the following method calls were identified.

`renderPage()`

An abstract method that implies rendering of a HTML page. All subclasses of `RenderingStrategy` must implement the method so they can render the page in their unique way. For example, a class of type `JSP_RenderingStrategy` would implement the method to handle JSP technology presentation;

`encodeServletUrl()`

Handle encoding URLs in the case of URL rewriting;

`getServletURL()`

Build a fully quantified URL that is made up of the following.

Get the scheme, which can be HTTP, HTTPS or FTP;

Get the hostname (eg) www.yahoo.com;

Get the path to the servlet;

Get the path after servlet and get the query string.

5.4.3 JSP_RenderingStrategy

Using the results of the *model* layer, this class provides an implementation for presenting JSP. The main responsibility of the class is to redirect the *controller* flow of control towards a particular JSP.

Therefore to fulfill the responsibilities of `JSP_RenderingStrategy`, the following method call were identified:

`renderPage()`

An implementation of its parent class (`RenderingStrategy`) abstract method. A JSP must be rendered in a particular fashion; the following describes the events that occur to satisfy the rendered JSP.

a) Firstly, add an object of type `AbstractRequest` to the `HttpSession`;
b) Retrieve the relative address URL to the JSP page. For example, /login.jsp;
c) Encode the relative address URL. For example, if there is a white space in the URL, encoding will change this to %20;
d) `javax.servlet.http.HttpServletResponse` is asked to send a redirect to the page.

5.4.4 MSP_RenderingStrategy

This class provides a mechanism to support the rendering of a new Java based dynamic page technology called MagnumServer Pages (MSP) (a full explanation can be found in section 5.5). The main responsibility of the class is to build the appropriate dynamic HTML content string by using the results of the *model* layer and then output the string to a client's browser. The class also provides functionality to compress the HTML content string before it is returned back to the browser, therefore solving the JSP performance limitation of no provision for compression of HTML content (a full explanation can be found in section 3.3.4).

Therefore to fulfill the responsibilities of `MSP_RenderingStrategy`, the following method calls were implemented:

`renderPage()`

An implementation of its parent class (`RenderingStrategy`) abstract method. A MSP must be rendered in a particular fashion; the following describes the events that occur to satisfy the rendering of MSP.

a) Firstly, add an object of type `AbstractRequest` to the `HttpSession`;
b) Encode the fully quantified servlet URL address;
c) Combined the results of the request object with the MSP page to create a HTML String;
d) Compress the HTML string and write it to the web browser.

`compressPageAndWrite()`

Apply `GZIP` compression to the HTML string before writing it back to the browser;

`renderPageAsString(AbstractRequest request)`

Build an HTML string to be render by combining dynamic elements from the request object (that is, of subtype `AbstractRequest`) with a MSP.

5.4.5 Summary

To summarise, the following is an outline on how the new implementation behaves in the *view* layer (see Figure 5.9):

1) In the new implementation, when a client's browser submits a HTTP GET / POST request to the *controller* layer, the improved implementation will create a polymorphic object of base type `AbstractRequest`;

2) The *controller* layer will delegate responsibility of business logic '*model* ' processing of the newly created request object to the `Dispatcher` class by invoking the `Dispatcher`'s `dispatch()` method;

3) The *controller* layer will invoke its private `decideRenderingStrategyAndRenderPage()` method, which in turn asks the object of base type `AbstractRequest` for its *Type* parameter

(HTTP name-value string pair, for example, Type=JSP) by invoking the request object's `getTYPE()` method;

4) Once the *Type* parameter has been retrieved, the *controller* layer will delegate rendering responsibilities to the correct subclass of `RenderingStrategy` by invoking the `RenderingStrategy`'s `renderPage()` method; the subclass will then present the page in its own specific manner.

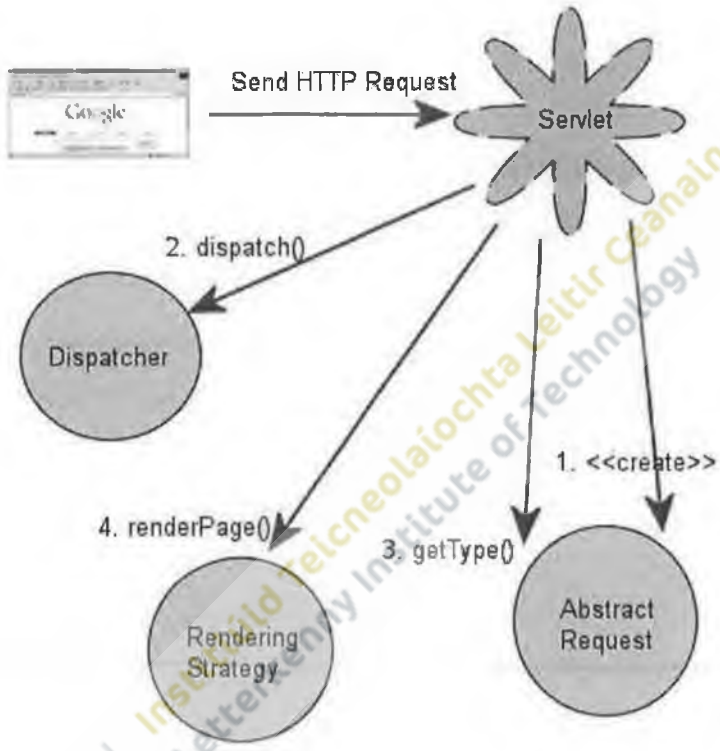


Figure 5.9: V*iew* layer outline behaviour diagram

We have discussed in detail the implementation of the *model*, *view* and *controller* layers of the new improved architecture. Therefore we can show an overall system diagram in Figure 5.10.

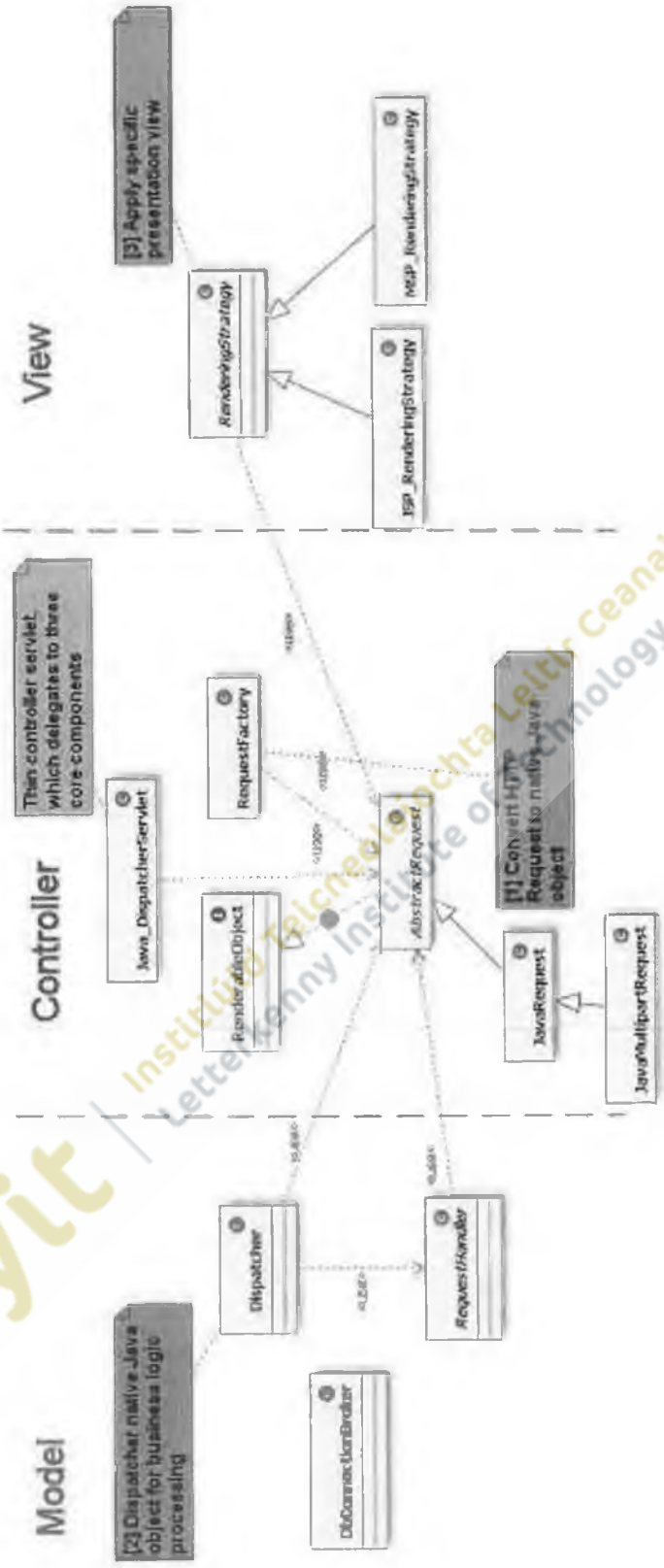


Figure 5.10: Overall system class diagram

5.5 MagnumServer Pages

Section 4.3.2 and 4.3.3 has proposed that there is a need for a new Java based dynamic web page technology called MSP and its responsibilities are as follows:

Provide an alternative to the JSP dependency on the HTTP protocol and the Java servlet API, which in turn provides the following solutions to some of JSP limitations:

a) Servlet thread spawning will be eliminated when using SSI, as MSP will only create additional objects in the JVM (a full explanation can be found in section 3.3.2);

b) In terms of testability, debugging an application's core business logic can now be performed without the JSP problems of trying to mimic the HTTP protocol, web browsers and web containers (a full explanation can be found in section 3.4);

c) In terms of testability, MSP parses its source files into native Java classes; therefore all compilation errors are in the form of native Java API exceptions. Thus eliminating JSP problematic native error handling (a full explanation can be found in section 3.4);

d) Again since MSP parses its source files into native Java classes and compiles them using the JVM. The JVM is more inclined to discover problematic runtime errors, identify warnings and depreciated methods oppose to the JSP compiler (a full explanation can be found in section 3.4);

e) Provide an alternative to the JSP optional dependency on JavaBean technology, which in turns solves one of JSP fundamental security problems (a full explanation can be found in section 3.5.1.3);

f) Provide an alternative to the JSP engine for page execution, MSP is solely reliant on the JVM for it execution therefore it will be less susceptible to JSP

application server vulnerabilities (a full explanation can be found in section 3.5.2).

5.5.1 MSP Scripting Language

MSP offers developers the ability to use its new Scripting Language (SL) because it is a new Java based dynamic web page technology. MSP SL is JSP syntax like language that allows developers to use HTML comments to add their dynamic syntax (Java syntax) instead of JSP scriptlet. Just like JSP there are three types of scripting elements: (i) code based scriptlet (used to execute a block of code); (ii) expression based scriptlet (an evaluated statement that is printed in the HTML) and (iii) declaration based scriptlet (used for declaring variables and methods). However unlike JSP, MSP SL does not provide implicit page objects (for example, *request*, *session* and *out*) or JSP Bean tags as it is not dependent on the servlet / JSP API. As mentioned previously MSP SL is in fact embedded HTML comments; therefore graphic designers can clearly view a MSP file in WYSIWYG editors without breaking the intuitive design of the surrounding static HTML.

The following are the MSP SL tags that can used to develop Java web pages that are not dependent on the JSP / servlet API.

Package tag

Declaration based scriptlet element that allows developers to specify what package the MSP file belongs to after page compilation (see Table 5.2).

	MSP	JSP
Syntax	<!--$ package <Java Package Name> -->	N/A
Example	<!--$ package com.thesis.pages -->	N/A

Table 5.2: Contrast between MSP and JSP package tag syntax

Import tag

Declaration based scriptlet element that allows developers to import necessary Java classes into their MSP page (see Table 5.3).

	MSP	JSP
Syntax	<!--$ import <Java Package Name> -->	<%@ page import="<Java Package Name>" %>
Example	<!--$ import java.util.* -->	<%@ page import="java.util.*" %>

Table 5.3: Contrast between MSP and JSP import tag syntax

Include tag

Declaration tag that provides the developer with the ability to make SSI statements in their MSP page. It is used to substitute additional text/html and/or code into the main body of their page. For example, developers can chop up their pages into significant sections (For example, header, footer, main body etc) so that if a change is needed throughout the website then only one file needs to be change as opposed to making changes to each page (see Table 5.4).

	MSP	JSP
Syntax	<!--$ incl <MSP Page Name> -->	<%@ include file="<FileName>" %>
Example	<!--$ incl CopyRight -->	<%@ include file="CopyRight.jsp" %>

Table 5.4: Contrast between MSP and JSP include tag syntax

Expression tag

Expression Tag evaluates the contents of the referred value and renders the value as a HTML string on the MSP Page. Only values of Java primitive types and/or of type `java.lang.String` can be evaluated otherwise an exception will be thrown (see Table 5.5).

	MSP	JSP
Syntax	<!--$ eval <expression to evaluate> -->	<%= <expression to evaluate> %>
Example	<!--$ eval firstName -->	<%= firstName %>

Table 5.5: Contrast between MSP and JSP expression tag syntax

Code tag

The code tag gives programmers the ability to insert java code snippets/fragments into their MSP Page. Typically the code tag is used to perform looping, boolean logic values and/or declare values in the page (see Table 5.6).

	MSP	JSP
Syntax	<!--$ code <insert code here> -->	<% <insert code here> %>
Example	<!--$ code for(int i=0; i < 10; i++) { } -->	<% for(int i=0; i < 10; i++) { } %>

Table 5.6: Contrast between MSP and JSP code tag syntax

For example, the following tags are listed in Figure 5.11.

1) Package tag;
2) Import tag;
3) Code tag (which initialises `java.util.ArrayList` with four strings);
4) Code tag (iteration of the `java.util.ArrayList`);
5) Expression tag (evaluate each string)
6) Code tag (for loop close brace)

```
<!--$ package com.thesis.pages --> (1)
<!--$ import java.util.* --> (2)
<!--$ code (3)
    String name1 = "Patrick";
    String name2 = "James";
    String name3 = "Michael";
    String name4 = "Matthew";

    ArrayList names = new ArrayList();
    names.add(name1);
    names.add(name2);
    names.add(name3);
    names.add(name4);
-->
<HTML>
<HEAD></HEAD>
<body>
<table border=1 cellspacing=0 cellpadding=0 width=150>
 <tr>
  <td width=150 valign=top><B>Name of Person</B>
  </td>
 </tr>
<!--$ code for(int i=0; i < names.size(); i++) (4)
      {
       String nameStr = (String)names.get(i);
-->
 <tr>
  <td width=150 valign=top><!--$ eval nameStr --></td> (5)
 </tr>
<!--$ code } --> (6)
</table>
</body>
</HTML>
```

Figure 5.11: Example MSP source file (`.msp`)

5.5.2 MSP significant classes

The MSP language is built from framework of interconnected classes which represent not only the tag symbols themselves but how the MSP source files (`.msp`) are parsed to Java classes and then how these classes output the dynamic content string.

5.5.2.1 CompiledPage

Once a MSP source file (`.msp`) is compiled into a Java source file (`.java`), it realises the contractual method `createDocument()` from the `CompiledPage` interface.

5.5.2.2 DocumentBuilder

Once `MSP_RenderingStrategy` invokes its implementation of `renderPage()` method, it calls this runtime document builder class `buildDocument()` method to return the dynamic content string. The `buildDocument()` method first instantiates (through reflection) an compiled MSP class and then invokes the MSP class `createDocument()` method to build the dynamic content string.

5.5.2.3 PageCompiler

MSP files (`.msp`) contains both static and dynamic tags (that is, HTML and MSP specific tags respectively), therefore once a page has been noted for page compilation. The `PageCompiler` class will first parse the `.msp` file into a collection of tags (that is, tokenising the file into static and dynamic blocks) and then construct a single `.java` source file by invoking the following routine (see Figure 5.12).

Create the new Java class definition by writing the following to a `.java` file:

1) The package location of the class (from parsing a MSP package tag);
2) The import statements of the class (from parsing a MSP import tag);
3) The class declaration (by retrieving the `.msp` filename and appending "*MSP_*" to the start file);
4) Realise the `CompiledPage` interface;
5) Create a class attribute of all static tags called *tags*. This is accomplished by sequentially looping through the collection of tags (both static and dynamic) and building an array of static tags only.

6) Create the new Java class method `createDocument()` (implemented from the `CompiledPage` interface) by writing the following to a .java file:
    a. Create the method declaration;
    b. Create a local variable of type `java.util.StringBuffer` (In production, this buffer will hold the contents of the dynamic HTML presentation before converting it to a string);
    c. Sequentially append static or MSP expression tags to the buffer, while intermixing Java code snippets (that is, MSP code tags).

Close the Java class by writing a brace "*}*" to the `.java` file.

```
* This is a generated class
*/
package com.thesis.pages;                                          (1)

import com.margey.msp.rendering.*;
import  java.util.*;                                               (2)

public class MSP_test implements com.margey.msp.rendering.CompiledPage {

        private static String[] tags = null;                       (3)
        static {                                                   (4)
            tags = new String[12];                                 (5)
            tags[1] = "\n";
            tags[3] = "\n";
            tags[5] = "\n<HTML>\n<HEAD></HEAD>\n<body>\n<table border=1 cellspacing=0 cellpadding=0 width=150>\n <tr>\r
            tags[7] = "\n <tr>\n  <td width=150 valign=top>";
            tags[9] = "</td>\n </tr>\n";
            tags[11] = "\n</table>\n</body>\n</HTML>\n";
        }

        /*
        * Handle a creation request                                (6)
        */
        public String createDocument(com.margey.msp.rendering.RenderableObject request) throws Exception {
            StringBuffer buff = new StringBuffer();                (7)

            buff.append( tags[1] );
            buff.append( tags[3] );                                (8)  <-- Static Tags

            String name1 = "Patrick";
            String name2 = "James";
            String name3 = "Michael";                              (8)  <-- Dynamic Tags
            String name4 = "Matthew";

            ArrayList names = new ArrayList();
            names.add(name1);
```

Figure 5.12: Example extract from MSP Java class file
(This is the result of parsing the MSP source file from Figure 5.11)

5.5.2.4 Tag

An abstract class that represents a general purpose tag (both static and dynamic). This class presents a series of boolean "*is<NameOfTag>()*"methods (for example, `isStatic()` method) that can be uniquely overridden by each MSP tag subclass, so that the subclass can be uniquely distinguishable. The Tag class also offers static methods for creating both static and dynamic tags.

5.5.2.5 PackageDirective

A subclass of `Tag` class that represents a MSP package tag (that is, `<!--$ package <Java Package Name> -->`). Overrides both `isPackageDirective()` (to return true) and `getDirectiveCode()` (which returns the proper package string) methods from the abstract `Tag` class.

5.5.2.6 ImportDirective

A subclass of `Tag` class that represents a MSP import tag (that is, `<!--$ import <Java Package Name> -->`). Overrides both `isImportDirective()` (to return true) and `getDirectiveCode()` (which returns the proper import string) methods from the abstract `Tag` class.

5.5.2.7 InclTag

A subclass of `Tag` class that represents a MSP include tag (that is, `<!--$ incl <MSP Page Name> -->`). Overrides both `isIncl()` (to return true) and `getRenderingCode()` (which returns a string signfying the creation of a new instance of `DocumentBuilder` and the invocation of the `buildDocument()` method) methods from the abstract `Tag` class.

5.5.2.8 EvalTag

A subclass of `Tag` class that represents a MSP expression tag (that is, `<!--$ eval <Expression to evaluate> -->`). Overrides `getRenderingCode()` (which returns a string value of the expression) method from the abstract `Tag` class.

5.5.2.9 CodeTag

A subclass of `Tag` class that represents a MSP expression tag (that is, `<!--$ code <insert code here> -->`). Overrides `getRenderingCode()` (which returns a string of the code) method from the abstract `Tag` class.

5.5.2.10 StaticTag

A subclass of `Tag` class that represents a collection of HTML static tags. Overrides both `isStatic()` (to return true) and `getRenderingCode()` (which returns the HTML block string) methods from the abstract `Tag` class. Also this class provides a conversion of any of the MSP files newline and tab characters to a Java string representation.

5.5.3 Summary

To summarise, MSP provides a range of classes and scripting tags to overcome the limitations of JSP. These tags and classes provide developers with the means to gain independence from the HTTP protocol / JSP engine, reduce application bugs through the intuitive testing and increased security.

6 Evaluation

6.1 Introduction

This section discusses performance benchmarking by comparing and contrasting the new MSP architecture against the following Java related web architectures:

a) Apache Struts

The most commonly used Java framework in today's software houses. Struts is a MVC architecture that uses a combination of servlets, JSP's and JSP custom tags technologies [Apache, 2004] (see Appendix B);

b) Apache Tapestry

Tapestry is Java component object model, which uses a high level API to develop web applications with the minimal amount of code [Apache, 2004b] (see Appendix B);

c) Page-centric JSP

Please refer to section 2.5 for further explanation.

Since the new architecture can render pages using both JSP and MSP (see section 5.5) technologies, it was decided that each technology within the new architecture should be individually benchmarked. Therefore, five performance benchmarks were performed.

6.2 System configuration

Before conducting each individual benchmark, the operating system was rebooted and all redundant applications and background processes were shutdown. The following system configuration given in Table 6.1 was used to conduct the benchmarks. The benchmarking client was Apache's JMeter 2.0.1, which is an application to load test functional behaviour and measure performance [Apache, 2004c]. JMeter sent multiple HTTP requests to a local application server of type Apache Tomcat 4.1 and subsequently retrieved the corresponding HTTP responses.

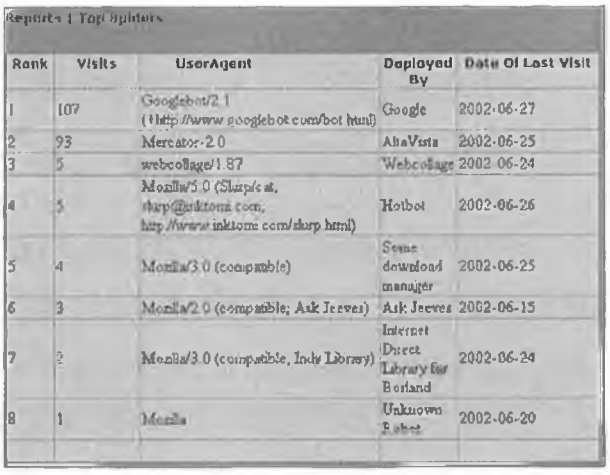
System type	Version
PC Type	Dell Dimension 4100
CPU	Intel Pentium 3 933Mhz
RAM	512MB
SDRAM	133Mhz
Hard Disk	20GB IDE Maxtor 32049H2
Operating system	Windows XP Professional SP 1
JVM	j2sdk 1.4.1_01
Java IDE	WebSphere Studio Application Developer 5.1
Application server	Apache Tomcat 4.1
Application client	Apache JMeter 2.0.1
Database	MySQL 3.23.55
JDBC Driver	MySQL Connector/J 2.0.14

Table 6.1: System configuration for benchmarking

6.3 Description of benchmarks

In total, two benchmark tests were performed on each of the five competing Java architectures. In the first benchmark, the JMeter client submitted a single HTTP request 300 times to measure performance under intense load. With the second benchmark, again the JMeter client submitted a single HTTP request 30 times at intervals of two seconds to measure performance under high volumes.

Each architectural design used a common dynamic web page throughout testing. The page is a simple table of data (see Figure 6.1). The data was contained in MySQL database, which was access via a JDBC driver. To be as unbiased as possible, it was decided to share as many common Java components between architectural benchmark. That is, only the specific architectural execution (not the business logic) and page rendering were different for each benchmark.



Reports | Top Spiders

Rank	Visits	UserAgent	Deployed By	Date Of Last Visit
1	107	Googlebot/2.1 (http://www.googlebot.com/bot.html)	Google	2002-06-27
2	93	Mercator-2.0	AltaVista	2002-06-25
3	5	webcollage/1.87	Webcollage	2002-06-24
4	5	Mozilla/5.0 (Slurp/si; slurp@inktomi.com; http://www.inktomi.com/slurp.html)	Hotbot	2002-06-26
5	4	Mozilla/3.0 (compatible)	Teleport download manager	2002-06-25
6	3	Mozilla/2.0 (compatible; Ask Jeeves)	Ask Jeeves	2002-06-15
7	2	Mozilla/3.0 (compatible, Indy Library)	Internet Direct Library for Borland	2002-06-24
8	1	Mozilla	Unknown Robot	2002-06-20

Figure 6.1: Example of common benchmark web page

Throughout the benchmarking process, each one of the architectures was analysed for average response time, thread rate per second, standard deviation of response times and finally a statistical sweep (that is, a comparison of average response time, median and standard deviation). After the results were analysed, the architectures were scored between one and five (that is, one being the lowest and five the highest) and then the overall results were collated to determine the architecture with the best performance.

6.4 Results of 1 thread executed 300 times

Upon analysing the average response time of requesting 300 top spider pages across the architectures (see Appendix C), it was declared that both MSP and *page-centric* JSP had an excellent average response time (see Figure 6.2).

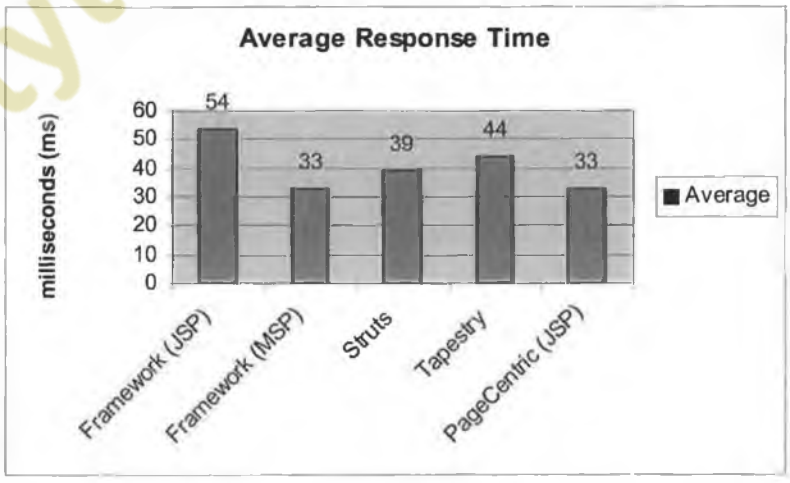


Figure 6.2: Column chart of average response times for first benchmark

When the thread rate per second was examined, It was discovered that MSP and *page-centric* JSP both performed the best. The Apache Struts framework pressed hard, however there was a noticeable drop off between the Tapestry framework and the improved architecture JSP (see Figure 6.3).

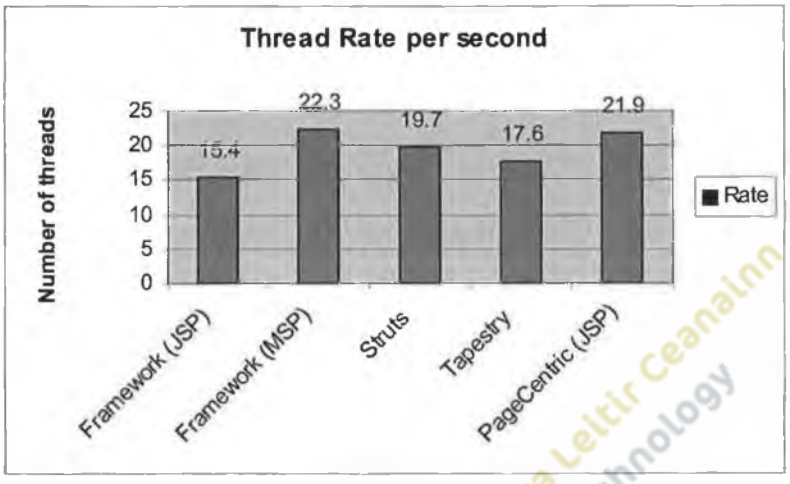


Figure 6.3: Column chart of thread rates for the first benchmark

Comparing the standard deviation of response times indicated that MSP won again, however it is interesting to note that Apache Struts performed better than expected. This can be attributed to the fact that Struts had the second fastest maximum response time (see Figure 6.4).

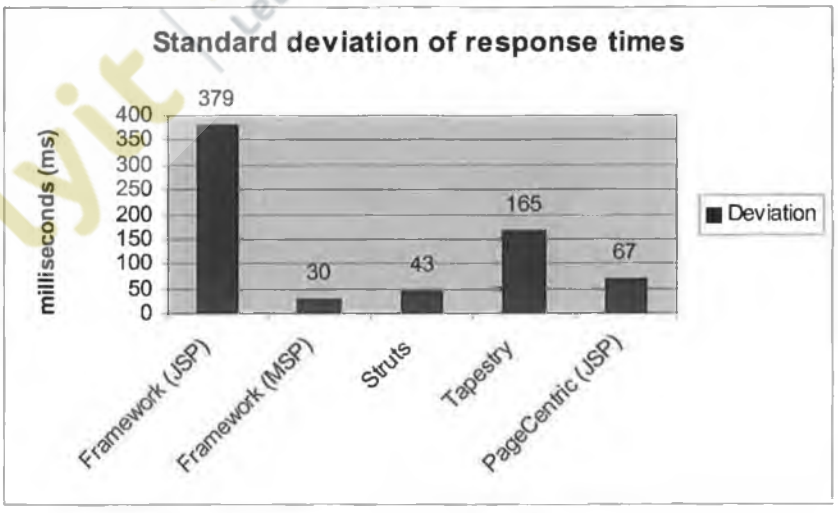


Figure 6.4: Column chart of standard deviations for the first benchmark

The combination of the average, standard deviation and median response times on the statistical chart displays very important performance information (see Figure 6.5). That is, the architecture that has the closest of the three results (average, standard deviation and median response times) means that the architecture in question is responding in a consistent and cohesive manner. Any dramatic changes between the three results means that the architecture in question is experiencing thread locking (for example, database pooling). Therefore the trio of MSP, Struts and *page-centric* JSP have performed in a consistent manner while Tapestry and the framework JSP could be experiencing performance problems (for example, XML processing and database pooling).

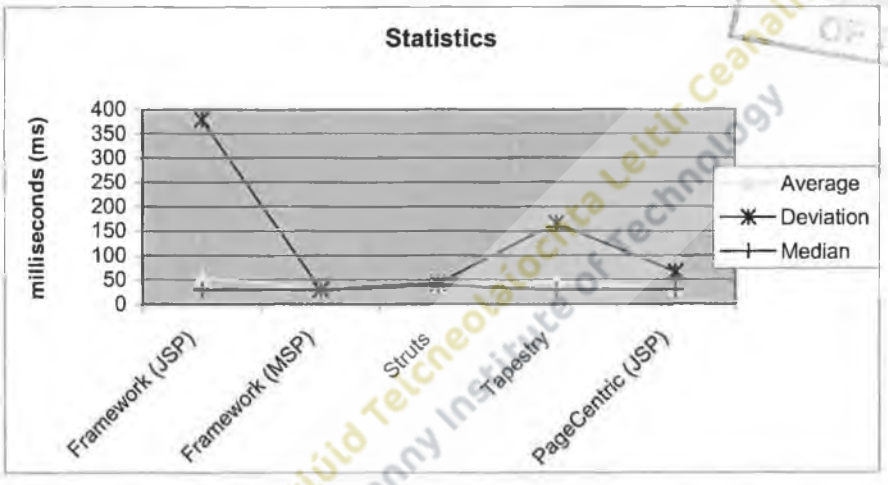


Figure 6.5: Line chart of statistical information for the first benchmark

6.5 Results for 1 thread executed 30 times between 2 second intervals

Upon analysing the average response time of requesting 30 top spider pages in two second intervals across the architectures (see Appendix D), it was declared that framework JSP was the winner with both MSP and *page-centric* JSP coming a close second and third (see Figure 6.6).

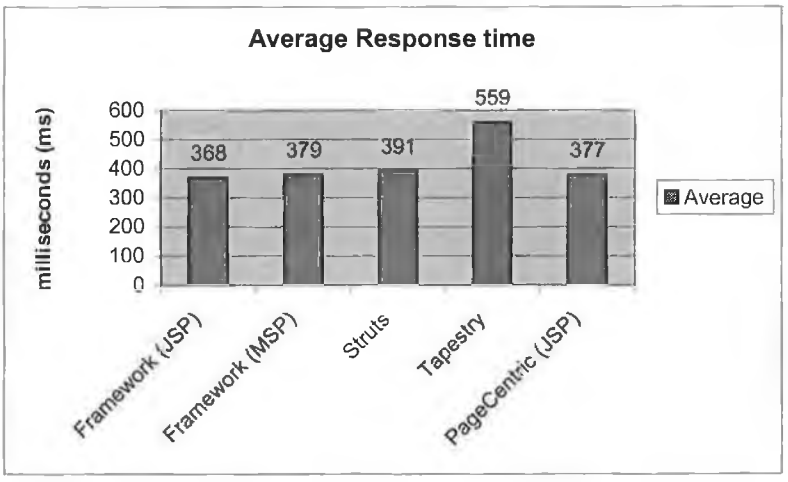


Figure 6.6: Column chart of average response times for the second benchmark

When the thread rate per second results were collated (see Figure 6.7), it was discovered that both the improved framework JSP and MSP had processed their HTTP requests significantly quicker than their competitors. It was assumed that the other architectures degraded due to architectural concerns, such as Apache Struts using a cache instance of `org.apache.struts.action.Action` class, which can cause thread locking issues on the class's `perform` method, Tapestry's heavy use of XML processing and *page-centric* JSP parsed and compiled servlet dealing with database processes such as retrieval and pooling processes while also performing massive amounts of object instantiation.

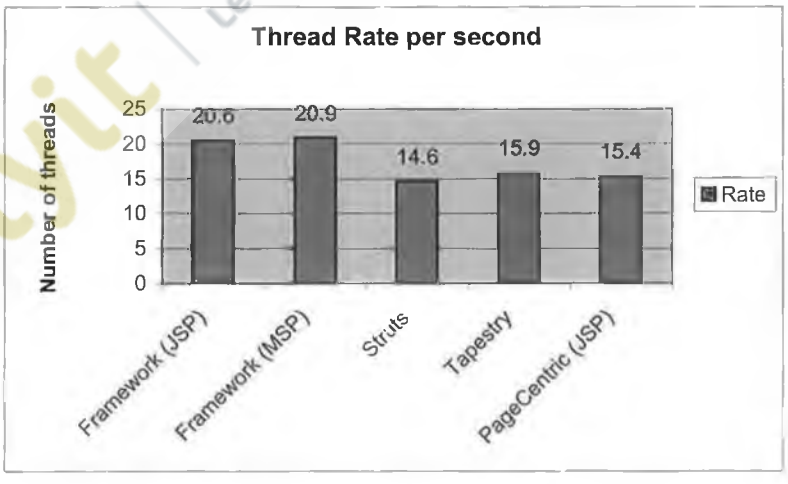


Figure 6.7: Column chart of thread rates for the second benchmark

It is interesting to note the both the framework's rendering strategies (that is, JSP and MSP) performed significantly better than its competitors. Both strategies had smaller

maximum response times, which again signifies threading concerns with the other architectures involved in the benchmark (see Figure 6.8).

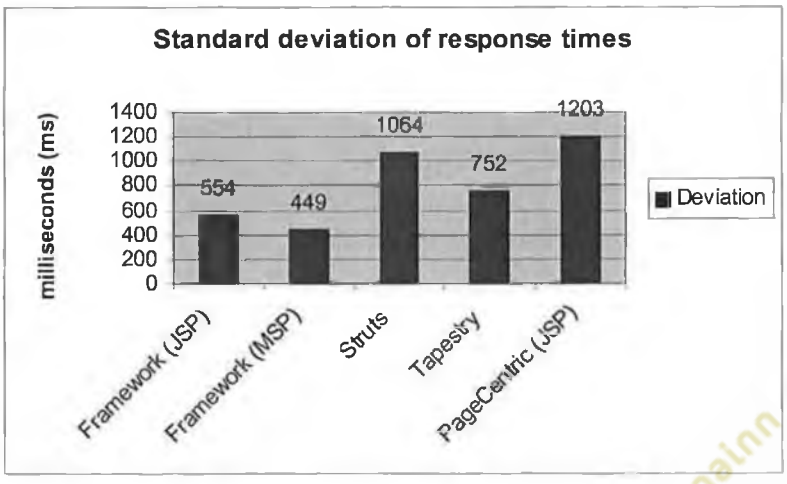


Figure 6.8: Column chart of standard deviations for the second benchmark

On closer inspection of the statistical results, It was discovered that framework JSP, MSP and Tapestry had performed well (the average, standard deviation and median response times are close together) however Tapestry was significantly slower than the framework JSP and MSP. Struts and *page-centric* JSP have an erratic spread of results, which again points to these architectures experiencing performance degradation due to thread problems (see Figure 6.9).

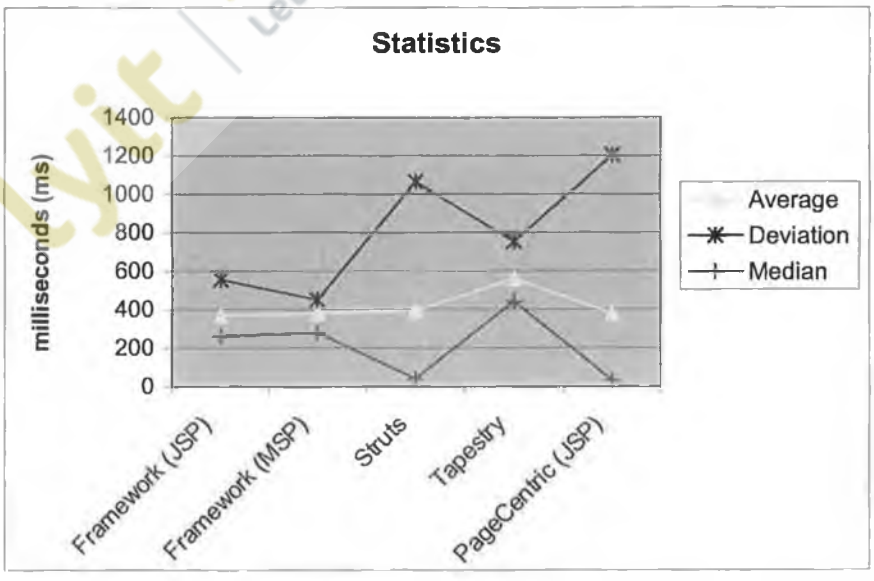


Figure 6.9: Line chart of statistical information for the second benchmark

6.6 Conclusions

On inspecting the collated score table from both benchmarks (see Table 6.2), it was surprisingly to find that the new MSP design using JSP to serve content had produced the lowest results for first benchmark (1 thread executed 300 times). However it had out performed the majority of other architectures in second benchmark (1 thread executed 30 times between 2 second intervals). It was judged that the first benchmark rogue results were due to performance degradation, which in argument was cause by the parsing and compilation of JSP servlet class upon the first HTTP request or system processes / resources not adequately being freed up.

Architecture	A (BM 1)	TR (BM 1)	SD (BM 1)	S (BM 1)	A (BM 2)	TR (BM 2)	SD (BM 2)	S (BM 2)	Total
Framework (JSP)	1	1	1	1	5	4	4	4	21
Framework (MSP)	5	5	5	5	3	5	5	5	38
Struts	3	3	4	4	2	1	2	2	21
Tapestry	2	2	2	2	1	3	3	3	18
PageCentric (JSP)	5	4	3	3	4	2	1	1	23

BM = Benchmark
A = Average Response Time
TR = Thread Rate
SD = Standard Deviation
S = Statistics

Table 6.2: Combined benchmark score card table

The new MSP design using its MSP language to serve content had the highest performance rating across both benchmarks. In argument this was due to a number of factors such as, no dependence on JavaBean introspection, XML processing, no parsing and compiling of JSP's and minimal coupling on the servlet API.

Even though the Apache Struts framework performed well in the first benchmark there was a noticeable drop off while performing the second. The problems could be attributed to the increase in JavaBean introspection calls and XML processing of its custom tag libraries. Also as argument, the class of type `org.apache.struts.action.Action` could be a factor as it is cached instead of reinitialised. Therefore while the class executes at an increased speed, it could adversely cause a thread locking with the database pooling component.

In relation to Apache Tapestry, which in one's opinion offers the best and most simplistic approach to solving the problem of separating of development roles (that is, graphic designer should only work on the web page while the programmer should work elsewhere on the background logic). It is clear that Tapestry under achieved in all categories; it can be argued that the reason for this is that there is a significant increase in Java reflection, introspection, XML processing and object creation in its architecture (see Appendix B)

It was surprising at how well *page-centric* JSP performed during the benchmarking. The results suggest that even with the problems of *page-centric* design (that is, maintainability and intuitiveness), it still performs better than some structured designs because *page-centric* design has less structured components to manage, which reduces JVM object creation and processing (for example, XML processing). It also has to be factored that the *page-centric* JSP approach has to parse and compile its servlet class upon the first HTTP request.

During the evaluation of the new architecture against competing frameworks, a number of software gaps became apparent regarding the new design. These were the following:

a) Lack of support of user friendly URLs

   Compared to the Apache Struts and Tapestry frameworks, the new MSP architecture does not support friendly URLs. The reason for this is that each HTTP request for a page must define two visible name–value parameters on a URL. That is, the parameters *Action* and *Type*, which define a subclass of `RequestHandler` to instantiate / execute and which rendering strategy to use respectively.

b) Clearer separation of development roles.

   In retrospect does the new framework with its MSP technology separate the role of developer from graphic / web designer? The answer for this is both yes and no, as it can be argued that there is no longer a reliance on JSP scriptlet or JavaBean technology as the MSP scripting language is just ordinary HTML comments. However the argument against is that the MSP scripting language

is too similar to native Java code therefore excluding most graphic / web designers.

c) Better error reporting.

Currently the new MSP design only catches exceptions in a generic error log. Shouldn't a developer have a feature to see exceptions directly on a client's web page? Apache Tapestry is excellent at providing a solution to this gap (that is, a detailed message about the error, application server and logic processor is displayed on the specific page where the error occurred).

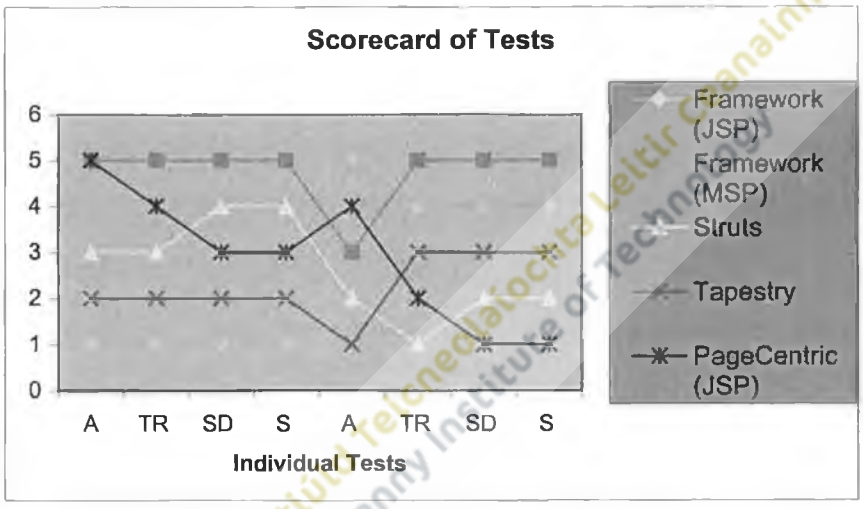


Figure 6.3: Line chart of scorecard results for combined benchmarks

7 Conclusion

7.1 Introduction

The implementation and benchmarking of the new MSP architecture provides proof that the new design does indeed fulfil its main objectives:

a) The new architecture provides a better design, which enables programmers to use a high level API that is devoid of the HTTP protocol. This API allows for faster, cleaner and better development of dynamic web pages.

b) The new architecture has outperformed all other competitors in terms of speed and scalability (see Figure 7.1).

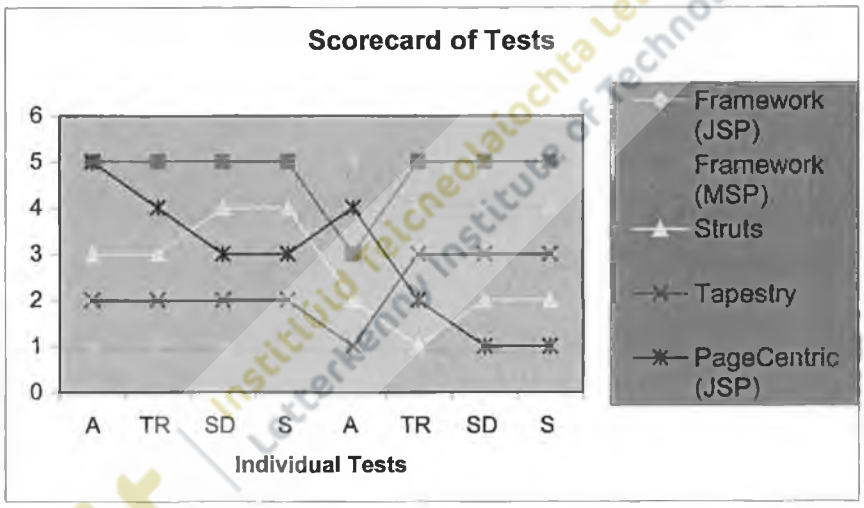


Figure 7.1: Line chart of scorecard results for combined benchmarks

c) Since the new MSP design reduces coupling with the HTTP protocol, its *model* components can be run as separate entities from the command line. Hence the new design can be easily used within a testing framework such as Apache JUnit, which will provide a solution to the problems with testing Java web application frameworks.

d) As outlined in section 3.5, JavaBeans or servlets have security holes associated to their technology. Since the MSP does not use any of these technologies, it can offer a better all round security solution. Also remember

the physical MSP page doesn't reside on the application server, only its representation as a standard Java class.

e) The new design intensively uses design patterns to decouple the model, view and controller layers to provide reusable components, which seamlessly fit into a '*true*' MVC design.

f) Specific page logic can now be implemented in a more intuitive UML use case fashion.

To summarise, compared to other frameworks, the new improved architecture can be considered an overall success and the architecture holds high potential to offer the development community with the next generation of robust, scalable and maintainable Java Internet dynamic rendering solution (see Table 7.1).

Capabilities	Normal JSP	Apache Struts	Apache Tapestry	Framework (MSP)
Automated testing	No	Yes	No	Yes
Presentation Speed	High	Normal / High	Slow	High
Coupling	High	Normal / Low	Very Low	Low
Development process	Easy	Normal	Complicated	Easy
Maintenance	High	Normal	Low	Normal
Security	Low	Normal	High	Normal / High
Extensible	Low	Normal	Normal	High

Table 7.1: Framework capability comparison

7.2 Future work

After performing benchmarking, it was concluded that additional functionality can be incorporated into the new design (see section 6.6), however it was considered that this functionality was outside the scope and timescale of the dissertation. The additional features should provide a starting point for future work, which would increase the functionality, usability and acceptance of the new framework as an all round viable web development system. The following are suggested enhancements:

a) Creation of a URL configuration file parser.
A centralised URL configuration file (which is XML based) could be

developed to hold URL information. This file could be read into memory at application start up. Thus when a user supplies a particular friendly URL (that is, a parameter less URL) then a new component parser could do a lookup on the configuration file to find both *Action* and *Type* XML elements and supply them to the rest of the system.

b) MSP tag libraries

The MSP scripting language could be extended to use a new specific tag library. One where common actions are simplified and intuitive to web designers.

c) Client side error reporting

The new framework catches exceptions and writes them to a specific error log. However, there is an issue with presenting errors on the page that it occurred. That is, in development or production environments there should be feature, which presents the developer with what type of specific error has occurred on screen instead of logging onto the application server and reading through log files.

d) Extensible MSP

The MSP technology itself could be extended to create a hybrid technology. Imagine the following in the new design; instead of sending a subclass of `AbstractRequest` to the *view* layer, the subclass could be decomposed into generic DNA (XML based) and then sent to a new rendering strategy. This new rendering strategy could read in a flat web page template file that contained extensible MSP markup tags. These tags would be intuitive and more user friendly (for example, a expression tag could be the following `<!--Element CustomerName-->` or a loop tag could be `<!--Loop Customers From 1 to 10-->`). Once the rendering strategy had both the generic XML DNA and the template then it could fuse the two together to create dynamic content. Also the generic XML DNA could be leverage so that it could be sent to some web service or even to an XSLT template.

e) Dynamic property files

An additional feature of the new framework could be a component which monitors the application property files for dynamic updates. That is, the application server or even the web application would not need a restart once a property file was changed.

f) Plug and play filtering components

The development of an abstract filtering component could have added value to the framework. That is, subclasses of the filtering component could be placed in between the *model*, *view* and *controller* layers. These filters could provide mechanisms such as XML parsing, object serialisation (for persistence) or localisation. These plug and play features could then be dynamically bound internally for certain situations that arose throughout the system.

g) Additional rendering strategies

Update the new improved architecture with new rendering strategies such as CGI, PHP and .Net which would enable the core architecture to deployed in a JVM and used with non Java web technologies.

References

[Altendorf et al, 2002] Altendorf, Eric, Hohman Mark and Zabicki, Roman, Using J2EE on a Large, Web-Based Project, p. 81-90, January/February, 2002, IEEE Software

[Althammer et al, 2003] Althammer, Egbert and Pree, Wolfgang, DESIGN AND IMPLEMENTATION OF A MVC-BASED ARCHITECTURE FOR ECOMMERCE APPLICATIONS, available @ www at http://citeseer.ist.psu.edu/443079.html accessed 31/05/2004

[Althammer et al, 1999] Althammer, Egbert and Pree, Wolfgang, AN ARCHITECTURE FOR A STRICT MODEL-VIEW SEPARATION IN JAVA, 1999, available @ www at http://citeseer.ist.psu.edu/althammer99architecture.html accessed 31/05/2004

[Apache, 2004] Apache software foundation, Kickstart FAQ, available @ www at http://jakarta.apache.org/struts/faqs/kickstart.html accessed 10/03/2004

[Apache, 2004b] Apache software foundation, Jarkarta Tapestry, available @ www at http://jakarta.apache.org/tapestry/index.html accessed 10/07/2004

[Apache, 2004c] Apache software foundation, Apache JMeter, available @ www at http://jakarta.apache.org/jmeter/ accessed 10/10/2004

[Alur et al, 2003] Alur, Deepak, Crupi, John and Malks, David, P. 34-54, Core J2EE Patterns: Best Practices and Design Strategies, Prentice Hall, 2003

[Ball, 2001] Ball, Michael, Dispatcher eases workflow implementation, P. 1-2, October, 2001 available @ www at http://www.javaworld.com/javaworld/jw-10-2001/jw-1019-dispatcher-p2.html, accessed 12/12/2002

[Bakken et al, 2003] Bakken, Stig, Aulbach, Alexander, Schmid, Egon, Winstead, Jim, Wilson, Lars, Lerdorf, Rasmus, Zmievski, Andrei and Ahto, Jouni, PHP Manual, April, 2003 available @ www at http://www.php.net/manual/en/ accessed 30/04/2003

[Bayern, 2002] Bayern, Shawn, JSTL in Action, Manning, 2002

[Bergsten, 2003] Bergsten, Hans, JavaServer Pages, 3rd Edition, O'Reilly, 2003

[Birznieks et al ,2000] Birznieks, Gunther, Guelich, Scott and Gundavaram, Shishir, CGI Programming with Perl Second Edition, 2000

[Booch et al, 1998] Booch, Grady. Jacobson, Ivar and Rumbaugh, James, Unified Modeling Language User Guide, Addison-Wesley, 1998

[Brown et al, 2001] Brown, Simon, Burdick, Robert, Falkner, Javson, Galbraith, Ben, Johnson, Rod, Kim, Larry, Kochmer, Casey, Kristmundsson, Thor, Li, Sing, Malks, Dan, Nelson, Mark, Palmer, Grant, Sullivan, Bob, Taylor, Geoff, Timney, John, Tyagi, Sameer, Van Damme, Geert and Wilkninson, Steve, Professional JSP 2nd Edition. P. 33-164, 165-196, 197-226, 227-263, 264-404, 708-738, Wrox Press Ltd, 2001

[Cavaness, 2002] Cavaness, Chuck, Programming Jakarta Struts, O'Reilly, 2002

[Challenger et al, 2000] Challenger, Jim, Iyengar, Arun , Witting, Karen, Ferstat, Cameron and Reed, Paul, A Publishing System for Efficiently Creating Dynamic Web Content, March, 2000, In Proceedings of IEEE INFOCOM 2000.

[Christiansen et al, 1998] Christiansen, Tom and Torkington, Nathan, O'Reilly, Perl Cookbook First Edition, 1998

[Cymerman, 1999] Cymerman, Michael, Building a Java servlet framework using reflection, Part 1, p. 1-2, November, 1999 available @ www at http://www.javaworld.com/javaworld/jw-11-1999/jw-11-servlet.html accessed 12/12/2002

[Cymerman, 2000] Cymerman, Michael, Building a Java servlet framework using reflection, Part 2. p. 1, February, 2000 available @ www at http://www.javaworld.com/javaworld/jw-02-2000/jw-02-servlets2.html accessed 12/12/2002

[Dai et al, 2000] Dai, Naci and Ellis, Michael, Best Practises for Developing Web Applications Using Java Servlets, P. 1-135, 2000, available @ www at http://www.smalltalkchronicles.net/papers/Practices.pdf accessed 27/05/2004

[Datta et al, 2002] Datta, Anindya, Dutta, Kaushik, Thomas, Helen, VanderMeer, Debra and Ramamritham, Krithi, Accelerating Dynamic Web Content Generation, p. 27-36, September/October, 2002, IEEE INTERNET COMPUTING

[Datta et al, 2002b] Datta, Anindya, Dutta, Kaushik, Thomas, Helen, VanderMeer, Debra, Ramamritham, Krithi and Suresha, Proxy-Based Approach for Dynamic Content Acceleration on the WWW, p. 159-165, June, 2002, Fourth IEEE International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems

[DeSoto, 1997] DeSoto, Alden, Using the Beans Development Kit 1.0, September, 1997 available @ www at http://java.sun.com/products/javabeans/docs/Tutorial-Sep97.pdf accessed 12/05/2004

[Dimov, 2002] Dimov, Jordan, JSP Security available @ www at http://www.developer.com/java/article.php/883381 accessed 12/12/2002

[Dorff et al, 2003] Dorff, Kevin C., Ship, Howard M. Lewis, Tapestry Tutorial, The Apache Software Foundation, 2003

[Dudney et al, 2003] Dudney, Ben and Lehr, Jonathan, Jakarta Pitfalls: Time-saving Solutions for Struts, Ant, Junit and Cactus. P. 1-65, 197-237, John Wiley & Sons Inc, 2003

[Fields et al, 2000] Fields, David and Kolb, Mark A., Web Development with JavaServer Pages, Manning, 2000

[Flanagan, 1999] Flanagan, David, Java in a Nutshell: A Desktop Quick Reference Third Edition. P. 330-333, Oreilly & Associates, 1999

[Fowler, 2003] Fowler, Martin, Patterns of Enterprise Application Architecture. P. 330-333, 344-349, 350-360, 379-386, Addison-Wesley, 2003

[Goodwill, 2000] Goodwill, James, Pure JSP -- Java Server Pages: A Code-Intensive Premium Reference. P. 10-21, SAMS, 2000

[Gamma et al, 1994] Gamma, Erich, Helm, Richard, Johnson, Ralph and Vlissides, John, Design Patterns: Elements of Reusable Object-Oriented Software. P. 87-95, 107-116, 223-232, 233-242, 293-303, 315-323, Addison-Wesley, 1994

[Gourley et al, 2002] Gourley, David, Totty, Brian, Sayer, Marjorie, Reddy, Sailu and Aggarwal, Anshu, HTTP: The Definitive Guide, O'Reilly, 2002

[Hall, 2001] Hall, Marty, Core Servlets and JavaServer Pages, p. 104-107, p. 287-309, Prentice Hall, 2001

[Hall, 2002] Hall, Marty, More Servlets and JavaServer Pages, p. 37-39, Prentice Hall, 2002

[Hall, 2003] Hall, Marty, Apache Struts: An MVC Framework p. 1-7 available @ www at http://courses.coreservlets.com/Course-Materials/ pdf/struts/Struts1.pdf accessed 03/04/2004

[Heaton, 2002] Heaton, Jeff, Comparing JSTL and JSP Scriptlet Programming, Dec, 2002 available @ www at http://www.samspublishing.com/articles/article.asp?p=30334&seqNum=1 accessed 23/09/2004

[Hieatt et al, 2002] Hieatt, Edward and Mee, Robert, Going Faster: Testing the Web Application, P. 60-65, March/April, 2002, IEEE Software.

[Hunter, 2000] Hunter, Jason, The Problems with JSP, February, 2000 available @ www at http://www.servlets.com/soapbox/problems-jsp.html accessed 12/12/2002

[Hunter et al, 1998] Hunter, Jason and Crawford, William, Java Servlet Programming, p 50-68, O'Reilly, 1998

[Huseby, 2001] Huseby, Sverre H, Tomcat may reveal script source code by URL trickery 2, April, 2001 available @ www at http://www.securityfocus.com/archive/1/173723 accessed 12/04/2004

[Iyengar et al, 2000] Iyengar, Arun, Challenger, Jim, Dias, Daniel and Dantzig, Paul, High-Performance Web Site Design Techniques, P.17-26, March/April, 2000, IEEE INTERNET COMPUTING

[Iyengar et al, 2002] Iyengar, Arun, Nahum, Erich, Shaikh, Anees and Tewari, Renu, Enhancing Web Performance, August, 2002, In Proceedings of the 2002 IFIP World Computer Congress (Communication Systems: The State of the Art, Kluwer)

[Johnson, 1997] Johnson, Mark, A walking tour of JavaBeans, August, 1997 available @ www at http://www.javaworld.com/javaworld/jw-08-1997/jw-08-beans.html accessed 14/05/2004

[Kalani, 2003] Kalani, Amit, MCAD/MCSD Training Guide 70-315: Developing and Implementing Web Applications with Visual C#.NET and Visual Studio.NET, Que, 2003

[Kassem et al, 2002] Kassem, Nick, Bodoff, Stephanie, Singh, Inderieet, and Johnson, Mark, p. 1-6, 75-128, Designing Enterprise Applications with the J2EE, Addison Wesley, 2002

[Klein, 2003] Klein, Amit, Cross Site Scripting Explained, August, 2003, available @ www at www.sanctuminc.com/pdf/WhitePaper_CSS_Explained.pdf accessed 14/04/2004

[Knight et al, 2002] Knight, Alan and Dai, Naci, Objects and the Web, P. 51-59, March/April, 2002, IEEE Software.

[Krasner et al, 1988] Krasner, Glenn E. and Pope, Stephen T. , A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System, Aug. 1988, P. 26–49, Journal of Object-Oriented Programming, vol. 1, no. 3.

[Kaewkasi et al, 2002] Kaewkasi, Chanwit and Rivepiboon, Wanchai, WWM: A Practical Methodology for Web Application Modeling, Aug, 2002, P. 603 – 609, 26th Annual International Computer Software and Applications Conference

[Knystautas, 2001] Knystautas , Serge, Cache in on faster, more reliable JSPs, May, 2001 available @ www at http://www.javaworld.com/javaworld/jw-05-2001/jw-0504-cache.html accessed 02/06/2004

[MageLang, 1999] MageLang Institute, Fundamentals of Java Servlets, 1999 available @ www at http://java.sun.com/developer/onlineTraining/Servlets/Fundamentals/contents.html accessed 05/05/2004

[Massol, 2003] Massol, Vincent, Unit Testing J2EE Applications, TheServerSide Symposium, Boston, p 6-43, June, 2003 available @ www at http://www.pivolis.com/pdf/Unit_Testing_J2EE_V1.1.pdf accessed 18/05/2004

[Mclaughlin, 2000] Mclaughlin, Brett, JSP Technology – friend or foe?, p. 6-11, October, 2000 available @ www at http://www-106.ibm.com/developerworks/library/w-friend.html accessed 12/12/2002

[McLaughlin, 2002] McLaughlin, Brett, Building Java Enterprise Applications Volume I: Architecture, O'Reilly, 2002

[NSCA, 98] The Common Gateway Interface available @ www at http://hoohoo.ncsa.uiuc.edu/cgi/overview.html accessed 12/07/2004

[Peeters, 2001] Peeters, Vera, Simple Design and Unit Testing with Enterprise JavaBeansTM: The Box Metaphor, p. 3-4, 2001 available @ www at http://www.xp2001.org/conference/papers/Chapter24-Peeters.pdf accessed 18/05/2004

[Ping, 2003] Ping, Yu, Kontogiannis, Kostas and Lau, Terence C., Transforming Legacy Web Applications to the MVC Architecture available @ www at http://www.swen.uwaterloo.ca/~kostas/STEP2003/ EAI-PAPERS/Ping-Lau-Kontog.doc accessed 01/06/2004

[Pipka, 2002] Pipka, Jens Uwe, Test-Driven Web Application Development in Java, p. 6-10, October, 2002 available @ www at http://www.netobjectdays.org/pdf/02/papers/node/0389.pdf accessed 18/05/2004

[Rayvok, 2002] Rayvok, Rossen, JSP Source code exposure in Tomcat 4.X, p.8, September, 2002 available @ www at http://online.securityfocus.com/archive/1/292936/2002-11-25/2002-12-01/2 accessed 12/12/2002

[Roschelle, 2000] Roschelle, Jeremy, Untangle your servlet code with reflection, p. 1-4, December, 2000 available @ www at http://www.javaworld.com/javaworld/jw-12-2000/jw-1221-reflection.html accessed 12/12/2002

[Rose, 2000] Rational Rose Enterprise Edition, Help facility, Version 7.5

[Scott et al, 2002] Scott, David and Sharp, Richard, Developing Secure Web Applications, p. 38-45, November/December, 2002, IEEE Internet Computing.

[Seshadri, 1999] Seshadri, Govindi, Understanding JavaServer Pages Model 2 architecture, December, 1999 available @ www at

http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssi-jspmvc.html accessed 12/12/2002

[Shah et al, 2000] Shah, Shreeraj and Shah, Saumil, IBM WebSphere default servlet handler showcode vulnerability, July, 2000 available @ www at http://www.securityfocus.com/archive/1/71508 accessed 17/06/2004

[Ship, 2004] Ship, Howard M. Lewis, p. 38–91, Tapestry in Action, Manning Publications, March, 2004

[Smith, 2004] Smith, Rob, Introduction to Jakarta Tapestry, May, 2004 available @ www at http://www.ociweb.com/jnb/jnbMay2004.html accessed 25/06/2004

[Sun, 1997] Sun Microsystems, JavaBeans™ API specification, August, 1997 available @ www at http://java.sun.com/products/javabeans/docs/spec.html accessed 14/05/2004

[Sun, 2001] Sun Microsystems, JavaServer Pages™ Specification Version 1.2, August, 2001

[Sun, 2002] Sun Microsystems, The J2EE Tutorial, April, 2002 available @ www at http://java.sun.com/j2ee/tutorial/1_3-fcs/index.html accessed 10/06/2003

[Sun, 2002b] Sun Microsystems, String Concatenation/Performance and Improving Java I/O Performance, March, 2002 available @ www at http://java.sun.com/developer/JDCTechTips/2002/tt0305.html accessed 26/05/2004

[Sun, 2003] Sun Microsystems, The Java Servlet API White Paper available @ www at http://java.sun.com/products/servlet/whitepaper.html accessed 06/05/2004

[Unger, 2000] Unger, Kevin, Solve your servlet-based presentation problems, November, 2000 available @ www at http://www.javaworld.com/javaworld/jw-11-2000/jw-1103-presentation-p3.html accessed 12/12/2002

[Welling et al, 2001] Welling, Luke and Thomson, Laura, p. 4 –5, PHP and MySQL Web Development, Sams Publishing, 2001

[Wu et al, 2000] Wu, Amanda w., Wang, Haibo and Wilkins, Dawn, Performance Comparison of Alternative Solutions For Web-To-Database Applications, p. 6-10, October, 2000, In *the Proceedings the Southern Conference on Computing*, the University of Southern Mississippi

[W3C, 1999] W3C, June, 1999 available @ www at http://www.ietf.org/rfc/rfc2616.txt accessed 15/05/2004

[Zeiger, 1999] Zeiger, Stefan, Servlet Essentials, November, 1999 available @ www at http://www.novocode.com/doc/servlet-essentials/ accessed 05/05/2004

[Zhao et al, 2002] Zhao, Weiquan, Kearney, David and Gioiosa, Gianpaolo, Architectures for Web Based Applications, 2002 available @ www at http://citeseer.nj.nec.com/cache/papers/cs/25755/http:zSzzSzwww.dstc.monash.edu.auzSzawsa2002zSzpaperszSzZhao.pdf/architectures-for-web-based.pdf accessed 01/06/2004

Bibliography

[Baskerville et al, 2003] Baskerville, Richard, Ramesh, Balasubramaniam, Levine, Linda, Pries-Heje, Jan and Slaughter, Sandra, Is Internet-Speed Software Development Different?, p. 70-77, November/December, 2003, IEEE Software

[Castagnetto et al, 1999] Castagnetto, Jesus, Rawat, Harish, Schumann, Sascha, Scollo, Chris and Veliath, Deepak, p. 14-16, Professional PHP Programming, Wrox Press Ltd, 1999

[Geary, 2000] Geary, David, JSP Templates, September, 2000 available @ www at http://www.javaworld.com/jw-09-2000/jw-0915-jspweb-p2.html accessed 12/12/2002

[Halloway, 2000] Halloway, Stuart, Improving Serialization Performance with Externalizable, April, 2000 available @ www at http://java.sun.com/developer/TechTips/2000/tt0425.html accessed 23/09/2004

[Maurer et al, 2002] Maurer, Frank and Martel, Sebastien, Extreme Programming - Rapid Development forWeb-Based Applications, p. 86-90, January/February, 2002, IEEE INTERNET COMPUTING

[Menascé, 2002] Menascé, Daniel A., Load Testing of Web Sites, p. 70-74, July/August, 2002, IEEE INTERNET COMPUTING

[Mercay et al, 2002] Mercay, Julien and Bouzeid, Gilbert, Boost Struts with XSLT and XML, p. 1-3, February, 2002 available @ www at http://www.javaworld.com/javaworld/jw-02-2002/jw-0201-strutsxslt.html accessed 21/02/2004

[Meyer, 1997] Meyer, Bertrand, Object-oriented software construction, Prentice-Hall, 1997

[Pooley et al, 2002] Pooley, Rob, Senior, Dave and Christie, Duncan, Collecting and Analyzing Web-Based Project Metrics, p. 52-58, January/February, 2002, IEEE Software

[Sun, 1999] Sun Microsystems, Comparing JavaServer Pages™ and Microsoft® Active Server Pages™, 1999 available @ www at http://java.sun.com/products/jsp/jsp-asp.html accessed 12/12/2002

Appendix A

UML Diagrams

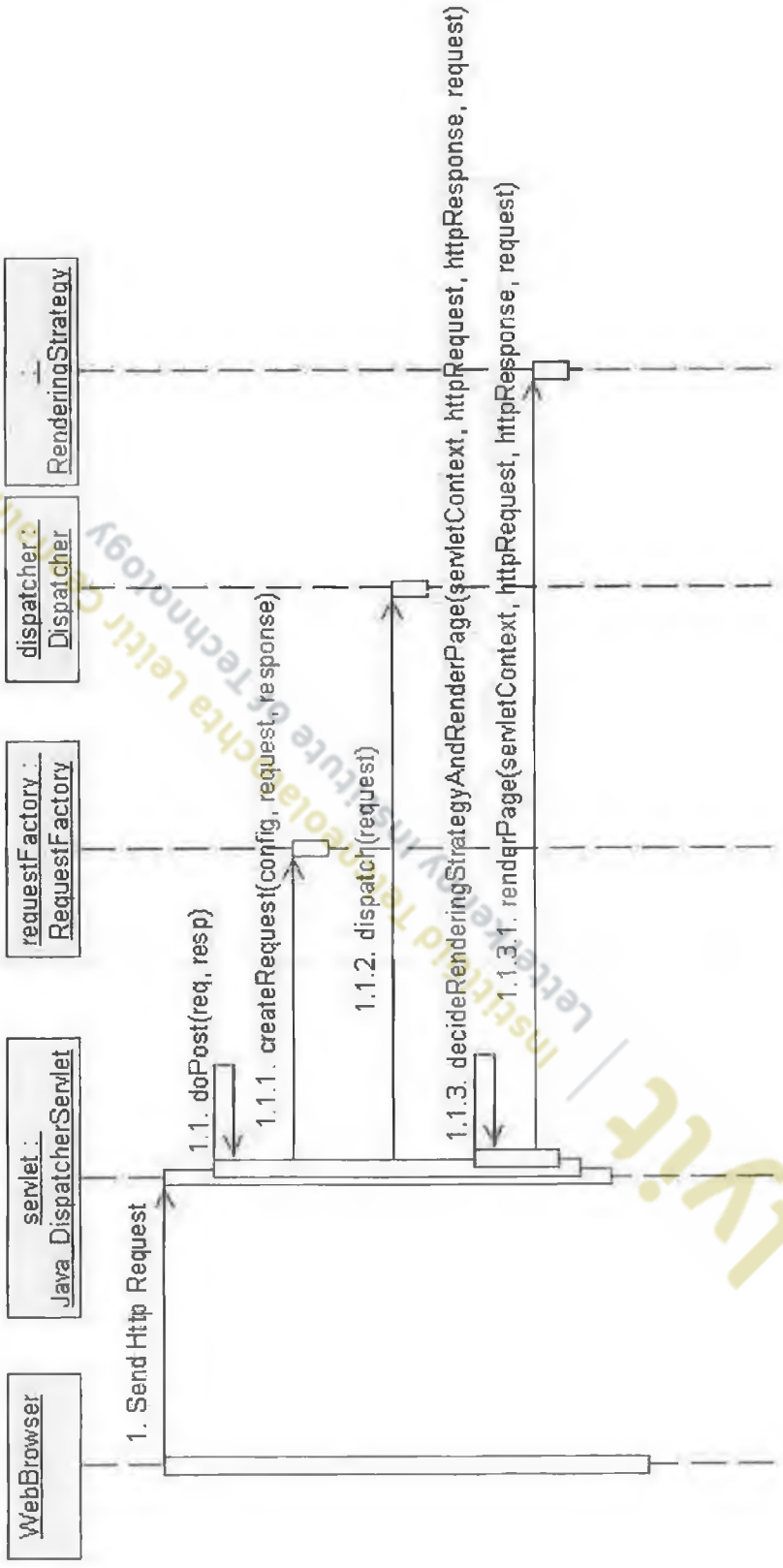


Figure A.1: UML Sequence diagram of HTTP processing by servlet

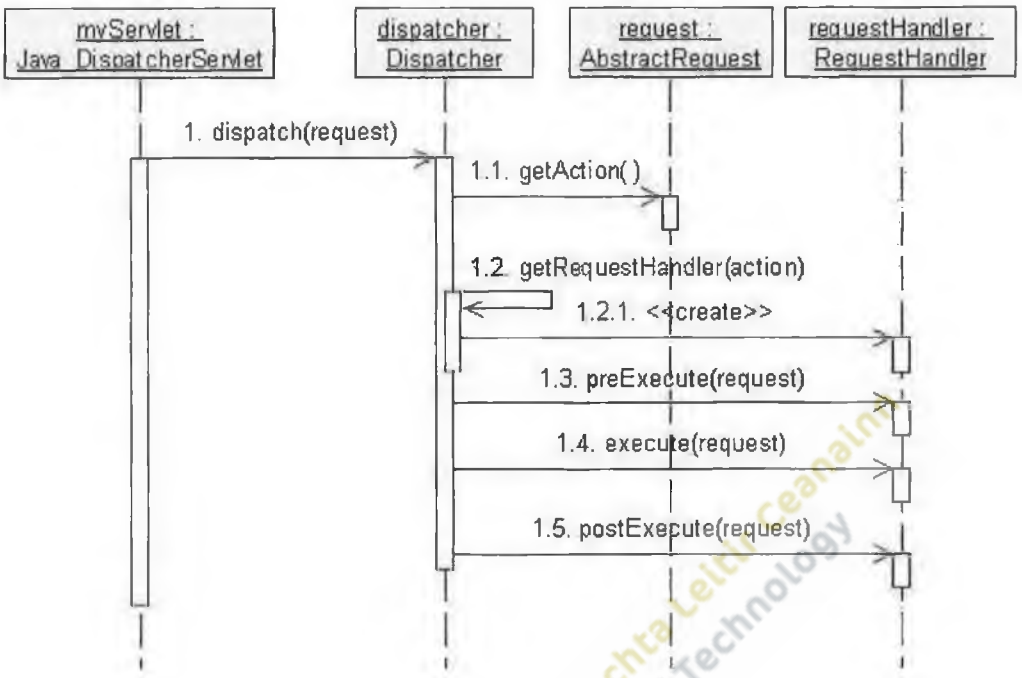


Figure A.2: UML sequence diagram of request dispatching

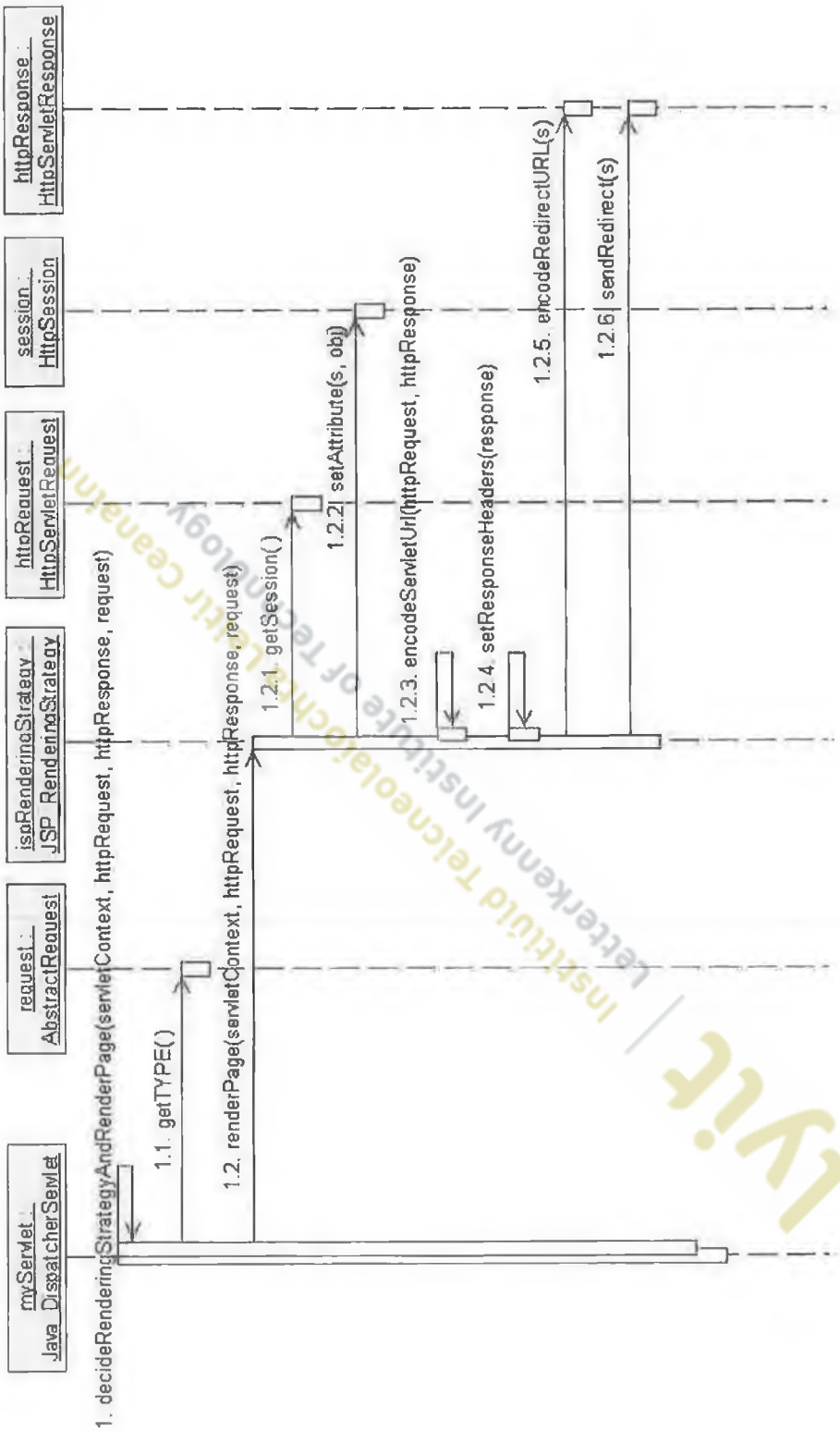


Figure A.3: UML sequence diagram of rendering page (JSP style)

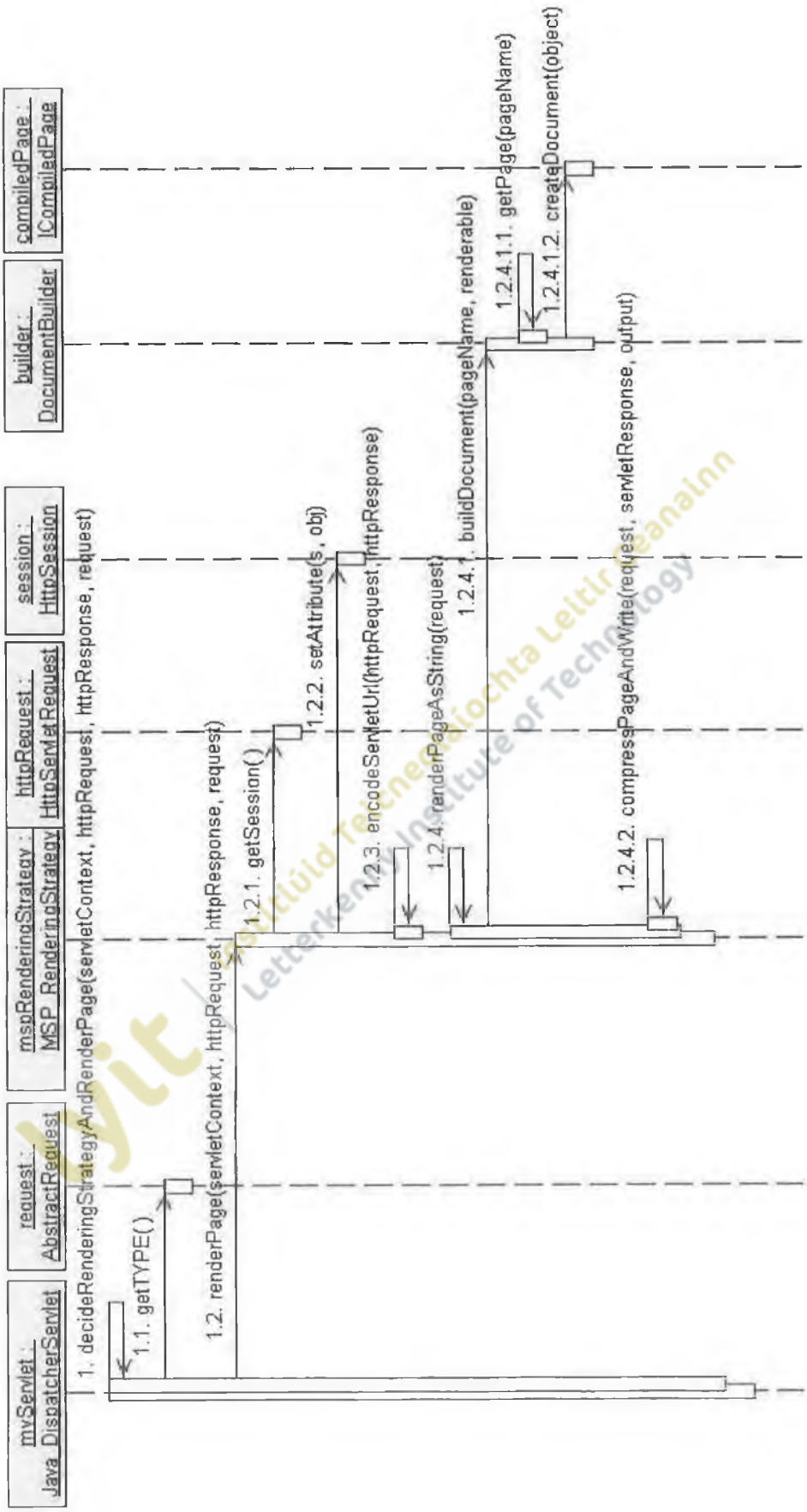


Figure A.4: UML sequence diagram of rendering page (MSP style)

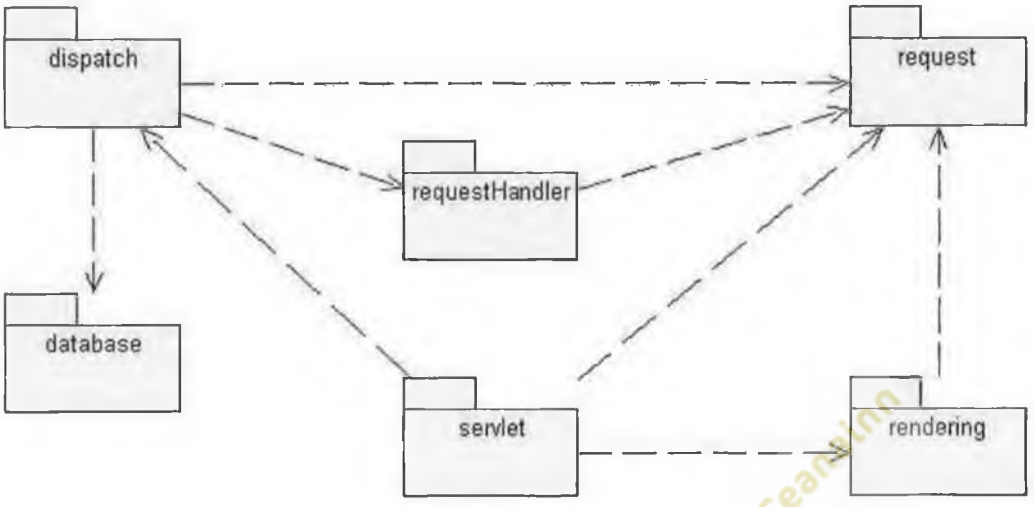


Figure A.5: UML component diagram of overall framework

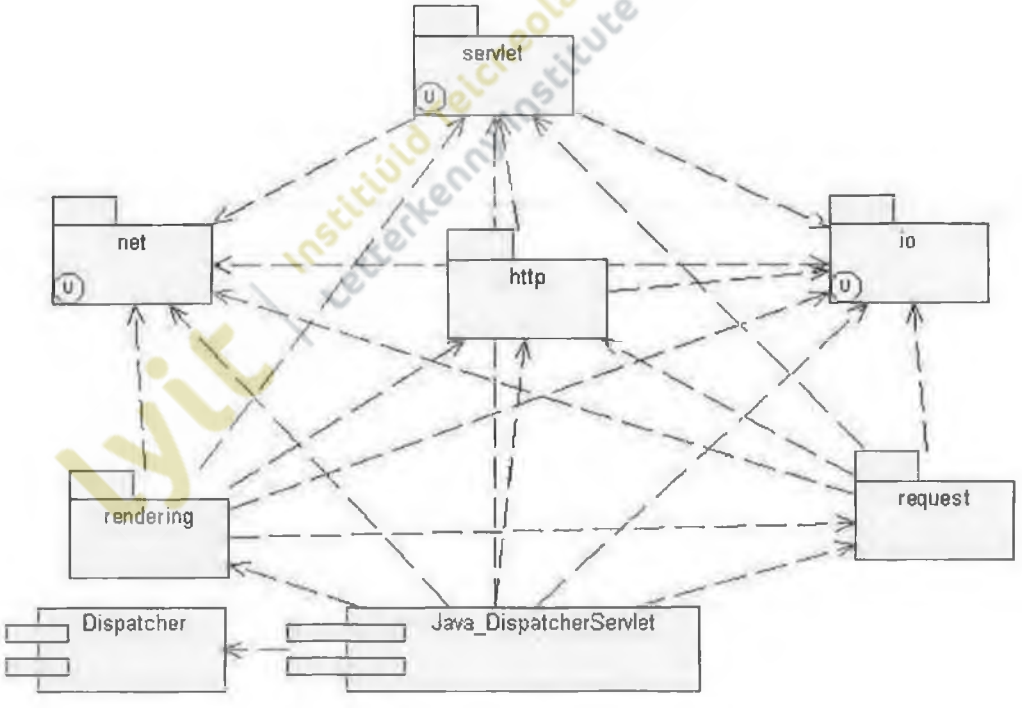


Figure A.6: UML component diagram of servlet

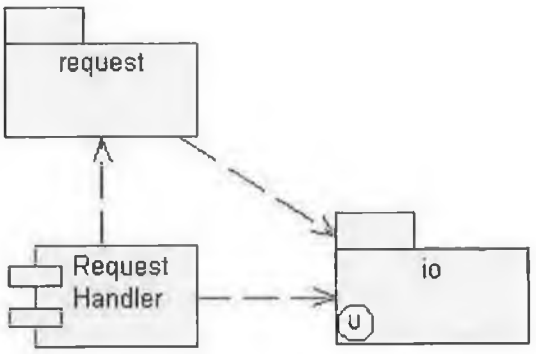


Figure A.7: UML component diagram of request handler

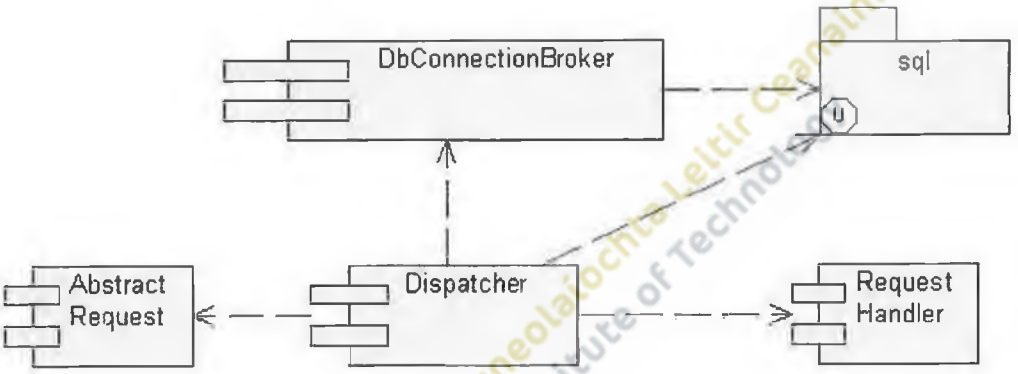


Figure A.8: UML component diagram of dispatcher

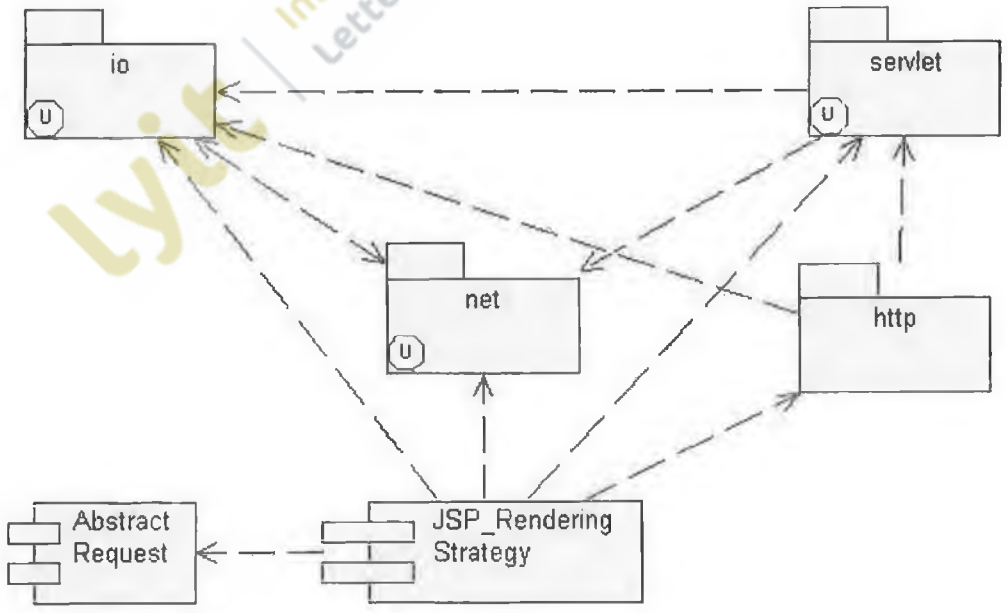


Figure A.9: UML component diagram of JSP rendering strategy

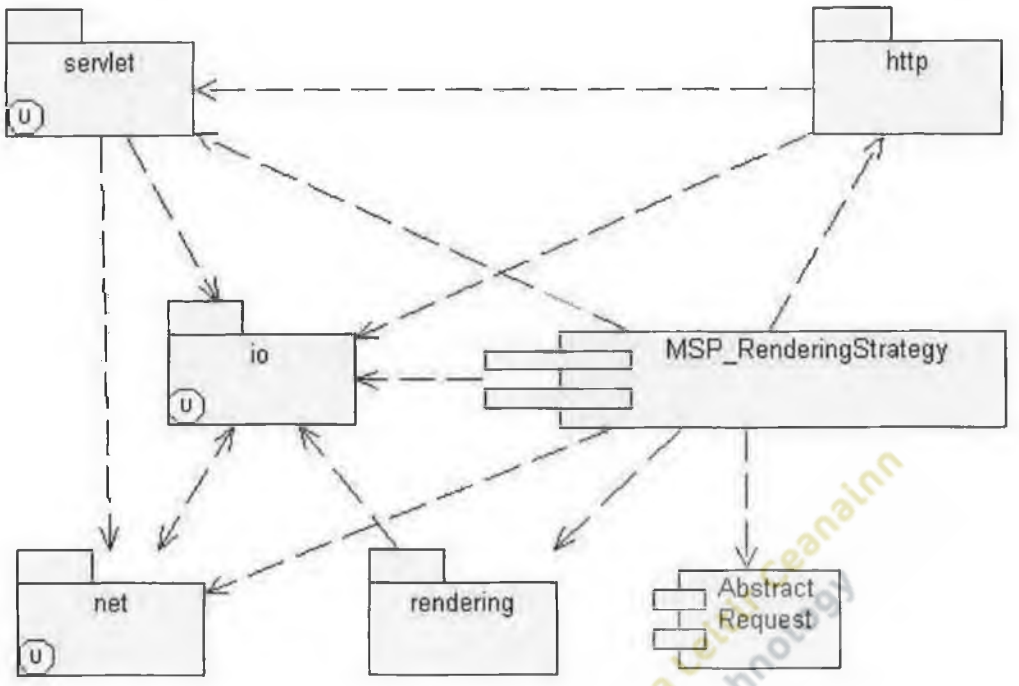


Figure A.10: UML component diagram of MSP rendering strategy

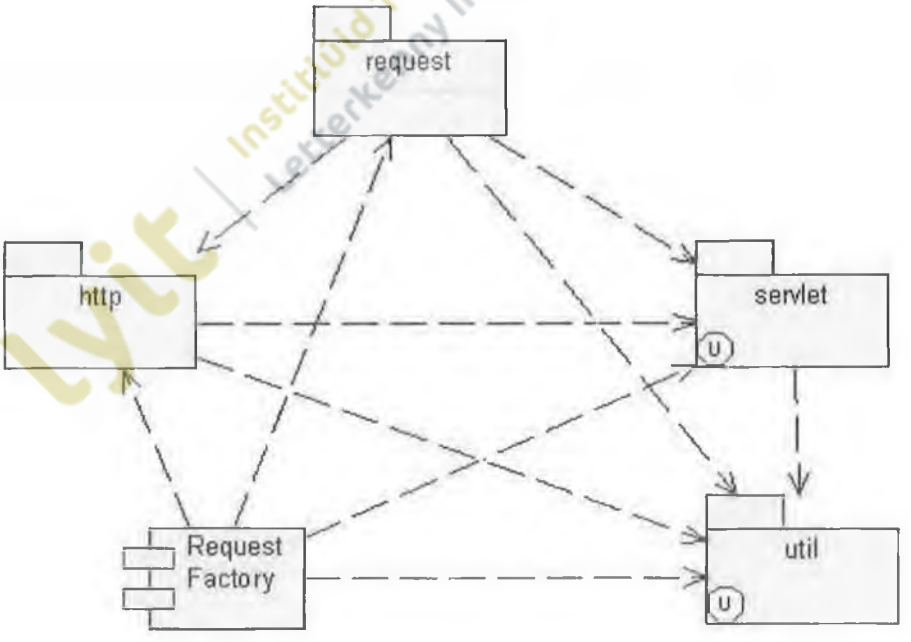


Figure A.51: UML component diagram of request factory

Appendix B

Alternative Java Architectures

B.1 Introduction

Although this dissertation's new framework architecture solves many of JSP architecture problems, it is not the first innovative idea to be suggested. The following section reviews and discusses the most popular alternative Java based architectures to determine which JSP problems they solve and what advantages / disadvantages do they have as part of their solution.

B.2 Apache Struts framework

Struts is a open source technology framework written Java. It was created by Craig R. McClanahan and donated to the Apache Software Foundation in May 2000. The framework was constructed to combine Java Servlets, JSP's and JSP Custom tags into a workable model view controller (MVC / Model 2) infrastructure [Apache, 2004] [Cavaness, 2002].

B.2.1 Components of Struts framework

The Struts framework provides five main components in which developers use to build web applications:

a) The controller servlet in the form of the `ActionServlet` class (`org.apache.struts.action.ActionServlet`). This class takes incoming HTTP requests and delegates them to the `RequestProcessor` component for processing [Apache, 2004] [Cavaness, 2002];

b) A developer must write the model component that encapsulates all the particular business logic for a given action / execution of an HTTP request. The model component must be a subclass of the `Action` class (`org.apache.struts.action.Action`) and define a perform method [Apache, 2004] [Cavaness, 2002];

c) A developer must write a form component (if needed) with maps directly to an HTTP form. The form component will encapsulate an HTTP post request to the `ActionServlet`. The form component must be a subclass of the `ActionForm` class (`org.apache.struts.action.ActionForm`) [Apache, 2004] [Cavaness, 2002];

d) The developer must write the view component (JSP page) to render the results of HTTP request [Apache, 2004];

e) The developer must configure the central struts XML file (`struts-config.xml`) that includes Action mappings to combine all above Struts components together [Apache, 2004] [Cavaness, 2002].

B.2.2 Struts Action mapping

An Action mapping file is defined in the form of `struts-config.xml`, which is located in the `WEB-INF` (see section 2.3.3) folder of a web application. This XML configuration file holds information on how to map individual HTTP requests to their corresponding Struts `Action` class (see Figure B.1).

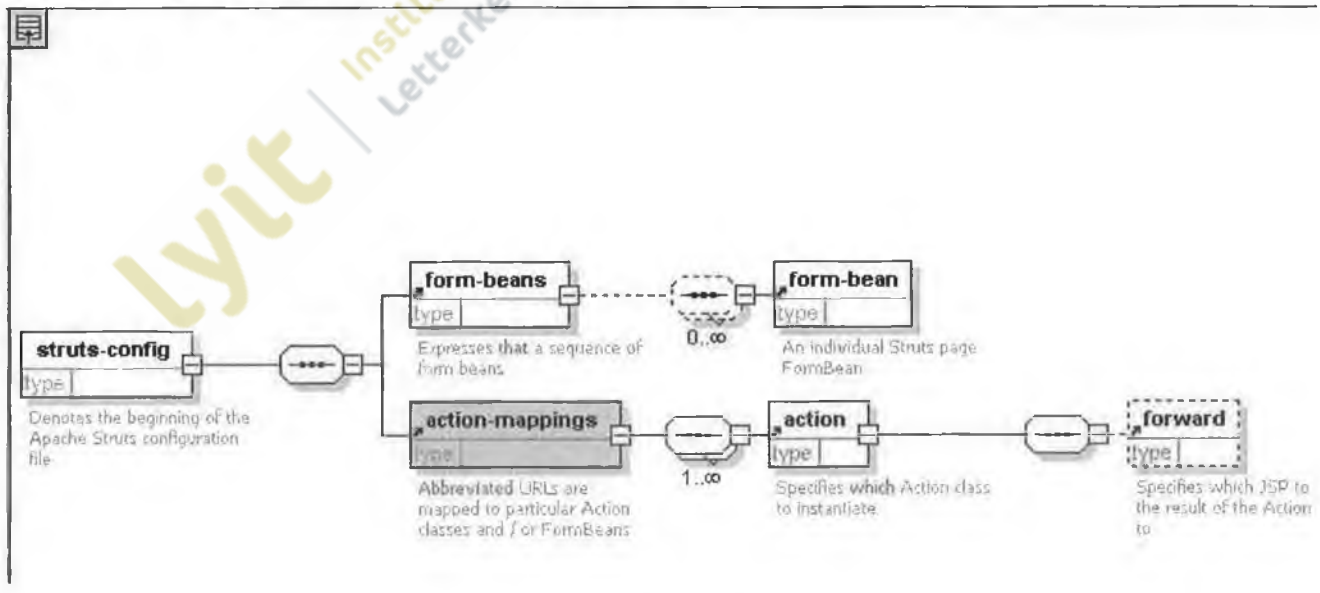


Figure B.1: Diagram of struts-config.xml file structure

An Action XML tag (see Figure B.1) can contain the following XML attributes:

- **Path attribute** - The URL to identify the Action;
- **Type attribute** - The fully qualified class name of the `Action` class;
- **Name attribute** - The name of the business logic worker `FormBean` class (if needed);
- **Scope attribute** – The page scope of `FormBean`;
- **Forward sub-element** - The simplified names (ActionForwards) of actual JSP files.

An Form-Bean XML tag (see Figure B.1) can be associated with the XML attributes:

- **Type attribute** - The fully qualified class name of the `ActionForm` class
- **Name attribute** - The name of the `FormBean` class (if needed) contumacious

For more clarity, the following is a *real-world* example of struts-config.xml file structure.


```
<struts-config>
    <form-beans>
        <form-bean name=" loginFormBean " type="myapps.formbean.LoginFormBean "/>
    </form-beans>
    <action-mappings>
        <action path="/loginAction" type="myapp.actions.LoginAction" name="loginFormBean" scope="session">
            <forward name="login" path="/login.jsp"/>
        </action>
    </action-mappings>
</struts-config>
```

B.2.3 How does Struts work?

The Struts framework processes individual Http request as follows (see Figure B.2):

1. The `ActionSerlvet` is first initialised with `struts-config.xml`, which indicates to the servlet how to deal with particular HTTP Requests;

2. The `ActionServlet` class will select the corresponding Action class and instantiate it through reflection;

3. Once object instantiation occurs the developer's `Action` object will make a call to its `perform()` method;

4. The perform method shall have all the necessary business logic to serve an HTTP request and instantiate the page's form bean / `ActionForm` object (if needed) to complete the process;

5. Once completed, the workflow the processing is forwarded on to the appropriate view based on the success, failure or alternative path to complete the action.

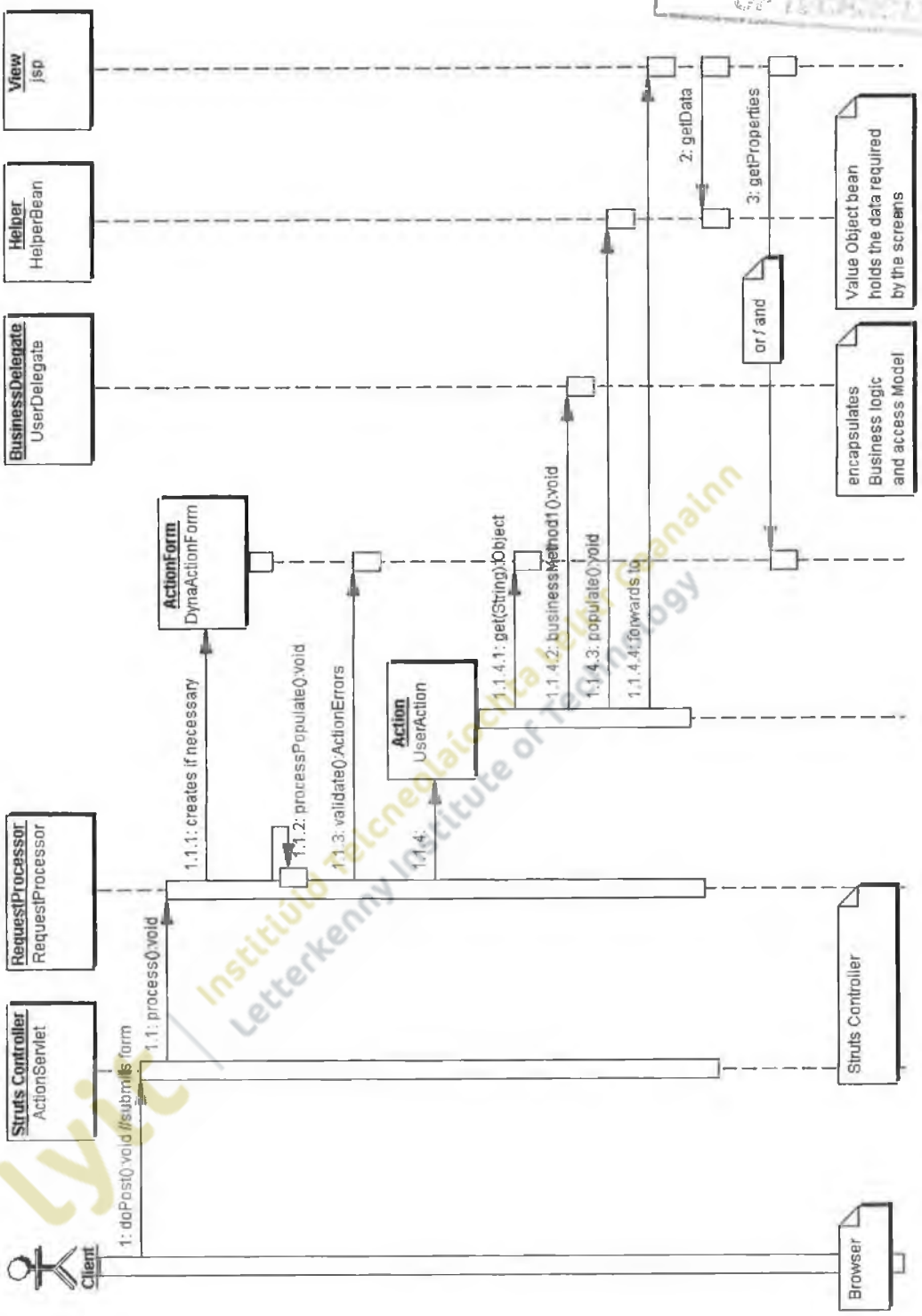


Figure B.2: Basic sequence diagram of Struts request (Extract taken from http://rollerjm.free.fr) Copyright (c) 1999-2002 The Apache Software Foundation. All rights reserved.

In the previous section the Struts Framework has clearly been defined and explained; however to gather a more rounded outlook an account of the framework's advantages and disadvantages must be given.

B.2.4 Advantages of Struts framework

The following are the advantages associated with using the Struts framework:

a) Stable and mature framework

Since 2000 Struts has been adopted and widely used by major software houses (IBM, Allstate etc) in building industry standard web applications.

Many new integrated development environments such as WASD (Websphere application studio developer), Netbeans and IBM's Eclipse provide easy to use and logical support for developing Struts applications.

b) Internationalisation and Localization support

The Struts framework installation package provides a rich set of language support mechanisms in the form of built in ResourceBundles.

c) Uses proven Java technologies

Struts provides support for many Java industry standard technologies (JSP, Tag Libraries etc)

d) Free to the public

There is no licensing or cost associated with Struts and it is freely available on the web.

e) Platform Independent

The Struts framework can run on any UNIX systems (e.g) Linux, Solaris etc and any Windows based platform.

f) Unit Testing

Struts provides an extension to the JUnit framework called StrutsTestCase. This extension allows developers to extensively test against from an application main entry point (the Struts ActionServlet) [Apache, 2004].

B.2.5 Disadvantages of Struts framework

The following are the disadvantages associated with using the Struts framework:

a) Learning Curve

A developer using the Struts framework must be proficient in JSP, Servlet and Custom Tags API and must have a firm grasp on the internals of the struts framework. Thus the framework adds another layer of complexity for less experienced developers. [Hall, 2003]

b) Poor Documentation

Compared to other open source frameworks (JUnit, PHP etc) Struts has quite poor documentation. Many users who experimented with Struts find the online documentation (the Apache resource site) very hard to understand. The documentation seems to be pitched at a developer with senior to expert level in the Java language. There are also very little recommended books on the subject matter compared to other languages and frameworks (.NET, PHP, ASP, JSP and Servlets) [Hall, 2003]

c) Problematic Custom Tags

It has been noted that several custom JSP tags within the Struts framework can be problematic and often lead to confusion and development down time. [Maturo, 2002]

d) Unseen static methods

Since Struts extensively uses reflection to build its dynamic content; any business logic classes static methods cannot be call through reflection. [Maturo, 2002]

B.3 Tapestry framework

Tapestry is an open source technology framework written Java. It was created by Howard M. Lewis Ship and donated to the Apache Software Foundation in 2000. Not unlike Java Swing's component object model for building desktop GUIs, Tapestry was built for the purpose of representing a dynamic web page as a Java component object model. Therefore the framework provides developers with a high level API, where the HTTP and servlet protocols are hidden so that a developer need only implement minimal code to develop a web application [Apache, 2004b] [Dorff et al, 2003].

B.3.1 Components of Tapestry framework

Since Tapestry provides a high level API, only three main components are needed to build a dynamic Tapestry web page:

a) Page class

The page class is a Java class (with a `.java` extension for source code) that represents a unique instance of a web page. By virtue of introspection and reflection, the page class methods and properties support the rendering of the HTML by dynamically populating a Tapestry HTML template. A page class must inherit from a Tapestry parent class called `org.apache.tapestry.html.BasePage` [Dorff et al, 2003].

b) Page specification

A Tapestry page specification is an validated XML file (with a .page extension) that is contained within the WEB-INF folder of a Java web application. The main responsibility of the page specification is to make a declaration of page components. These page components represent information on how to identify the page class that needs to be instantiated and which page class attributes are needed to dynamically populate the respective HTML template.

A page specification is made up of the following XML elements (see Figure B.3).

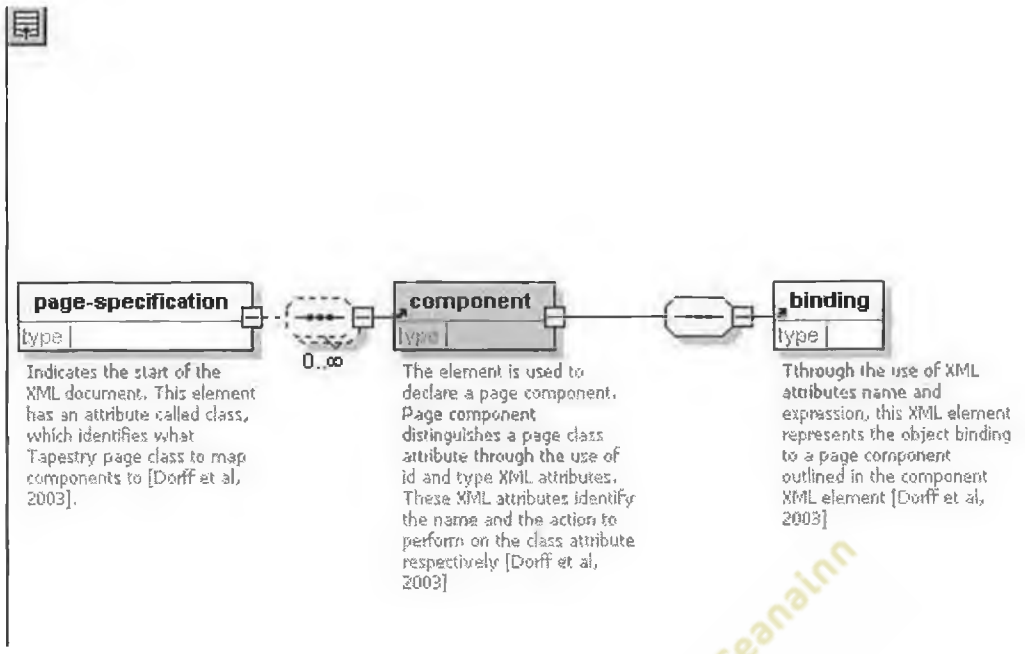


Figure B.3: Diagram of Tapestry page specification file structure

The following is an *real world* example of a Tapestry page specification.

```
<page-specification class="com.example.PersonDetailsPage">
   <component id="name" type="Insert">
      <binding name="value" expression="components.person.name"/>
   </component>
   <component id="address" type="Insert">
      <binding name="value" expression="components.person.address"/>
   </component>
</page-specification>
```

The example above indicates a page class called "`com.example.PersonDetailsPage`", which has two page components called "`name`" and "`address`". These page components perform an "`insert`" action, which subsequently binds to an object of type `Person` which contains two class instance attributes called "`name`" and "`address`"

c) HTML template

On first viewing a Tapestry HTML template looks like a normal HTML file. However the use of HTML <SPAN> tags indicate to Tapestry which parts of the template are dynamic components. HTML templates can be viewer in any WYSIWYG HTML editor as the file is composed totally of HTML markup tags.

The following is an example code snippet from a Tapestry HTML template

```
<span jwcid="spiders">
        <tr bgcolor="#CCCCCC">
          <td><span jwcid="ranking"/></td>
          <td><span jwcid="numberOfVisits"/></td>
          <td><span jwcid="userAgent"/></td>
          <td><span jwcid="deployedBy"/></td>
          <td><span jwcid="date"/></td>
        </tr>
</span>
```


B.3.2 How does Tapestry work?

The Tapestry framework processes individual Http request as follows:

a) Since application initialisation has parsed an XML file of type `.application` file extension (which maps URLs to their appropriate page specification). The Tapestry framework begins to parse the appropriate page specification (`.page` file extension) for page components;

b) During the parsing of the page specification, the HTML template is parsed to check what dynamic elements are needed;

c) After parsing the page specification (`.page` file extension), the framework by means of reflection then instantiates the appropriate page class and using introspection binds the page components to the dynamic elements outlined in the HTML template.

B.3.3 Advantages of Tapestry framework

The following are the advantages associated with using the Tapestry framework:

a) Simplicity

Compared to servlet and JSP applications, Tapestry's true power is through its ease of use. Tapestry developers need only create a page class and write an XML page specification to run a dynamic Tapestry web page, as oppose to implementing more code through using JavaBeans, servlets and `.jsp` files for servlet/JSP page rendering. Tapestry removes the low level servlet and JSP API's (`javax.servlet.http.*`) from its pages, developers are

developing at a high level, where the HTTP protocol has been hidden in favour of a pure Java object which acts as a page object. Low level programming and business logic is clearly separated, that is, Tapestry handles all low level aspects of web development (for example, session management) and the business logic can follow a Unified Modelling Language (UML) Use Case format [Ship, 2004] [Smith, 2004].

b) Consistency

Tapestry provides implementation consistency through the outlining of strict rules for building dynamic web pages. These pages follow a set of guidelines, such as coding standards and using reusable components that rule out inconsistencies when developing web applications [Ship, 2004] [Smith, 2004].

c) Efficiency

Tapestry web pages offer high application scalablility because during application initialisation, all Tapestry's dynamic web page XML specifications and HTML templates are read and parsed only once, and then cached to minimize processing time for each request. Also all page instances are stored in objects pools for later reuse [Ship, 2004] [Smith, 2004].

d) Error handling

Tapestry provides excellent error handling in the form of a complete diagnostic report on why the error occurred, that is a detailed exception page showing all nested exceptions, a stack trace at the deepest exception and a detailed description of the servlet and HTTP request environment. Also file and precise line numbering are presented to display what caused the error [Ship, 2004] [Smith, 2004].

e) Free to the public

There is no licensing or cost associated with Tapestry and it is freely available on the web.

f) Platform Independent

The Tapestry framework can run on any UNIX systems (e.g) Linux, Solaris etc and any Windows based platform.

B.3.4 Disadvantages of Tapestry framework

The following are the disadvantages associated with using the Tapestry framework:

a) Poor Documentation

Tapestry is not a widely accepted framework like Struts, therefore documentation on Tapestry is somewhat limited. Many users find the online documentation (the Apache resource site) very hard to understand. Also there are also very little recommended books on the subject matter compared to other languages and frameworks (.NET, PHP, ASP, JSP and Servlets)

b) Learning curve

There is a high learning curve to fully understand the whole component based Tapestry framework. Thus the framework adds another layer of complexity for less experienced developers.

c) Application initialisation

Although Tapestry uses caching and object pooling to increase page request performance, developers must recognise that during application initialisation the Tapestry framework will use a tremendous amount of introspection and XML parsing of meta data, therefore a performance lag will occur.

B.4 JSP Standard Tag Library

JSP Standard Tag Library (JSTL) is set of standardized JSP custom tags that provide a means for developers to create JSPs at an accelerated rate. These standardized JSP tags provide developers with a high level JSP tag API, where common mundane JSP tasks, for example, database access, internationalisation support and XML processing are hidden so that a developer need only implement minimal code to develop a JSP web application. These custom tags in turn reduce coding errors and promote overall JSP readability.

B.4.1 Components of JSTL

Since JSTL provides a high level API, there are four main components / libraries that can be use to build a simplified JSP.

a) JSTL core

This tag library provides a set of core utilities for simplifying common JSP scriptlet actions. For example, conditional statements, iterating collections, URL redirection and manipulation are all handled by this library [Bayern, 2002] [Bergsten, 2003]. To use this JSTL library, one must declare the following `taglib` directive tag

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"
%>
```

b) JSTL fmt (Internationalisation and formatting)

Previously developers using plain JSP scriptlet notation had always to provide their own set of functionality to support localization and general Java primitive type formatting. Again this tag library is set of common utilities that reduces the amount of development overhead, by providing tags that help developers input and output dates and numbers as well as localized formatting [Bayern, 2002] [Bergsten, 2003]. To use this JSTL library, one must declare the following `taglib` directive tag

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"
%>
```

c) JSTL sql (Database)

The Database tag library provides a set of utilities that simplify the connecting, querying and updating to a JDBC resource. Previously JSP developers usually had to develop their own Database JavaBean for simplified JDBC resource querying [Bayern, 2002] [Bergsten, 2003]. To use this JSTL library, one must declare the following `taglib` directive tag

```
<%@ taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql"
%>
```

d) JSTL XML

This tag library offers a set of tags to simplify XML document parsing, looping and transformation to XSLT [Bayern, 2002] [Bergsten, 2003]. To use this JSTL library, one must declare the following `taglib` directive tag

```
<%@ taglib prefix="x" uri="http://java.sun.com/jsp/jstl/xml" %>
```

B.4.2 JSTL Expression Language

Not only does JSTL reduce maintenance of JSP applications by avoiding JSP scriptlet elements by providing programmers with a set of custom tag libraries, it also offers developers the ability to use JSTL Expression Language (EL). The JSTL EL is a JavaScript like language which allows developers to use abbreviated object name syntax instead of JSP scriptlet (Java syntax) for data access upon the dynamic page's implicit and session based objects, for example, JavaBeans contained in session or JSP HTTP request header information.

The EL data access can denoted by using the following syntax [Bayern, 2002] (see Figure B.4)

EL Syntax	${<Name of JavaBean instance>.<Variable Name>} or ${<Name of JavaBean instance>["<Variable Name>"]}
EL Example Usage	${customer.firstName} or ${customer["firstName"]}

Figure B.4: JSTL EL Example usage diagram

The EL uses automatic JavaBean inspection to access data variables. For example, currently in JSP scriptlet programmers must downcast their base JavaBean class types after `java.lang.Object` retrieval from `HttpSession`. Where in JSTL, programmers need only call the base JavaBean class type directly [Heaton, 2002]. For example (see Figure B.5).

JSP Example

```
<%
    Customer aCustomer = (Customer)session.getAttribute("customer");
    if( aCustomer.getAge( ) > 18 )
    {
%>
    *** Do Something Here ***
<%
    }
%>
```

JSTL Example

```
<c:if test='${customer.age > 18}'>
    *** Do Something Here ***
</c:if>
```

Figure B.5: Example code difference between JSTL and JSP

B.4.3 JSTL Custom Tags

While JSTL provide a set of tags that solve many of the standard problems encountered by web developers; it does not cover all specific problem areas such as sending emails and file manipulation. The true power of JSTL is that it allows developers to build their own custom tag libraries to solve their own project specific problems. A developer must build a special class called a *tag handler* to handle a new custom JSP tag. Instead of developing a completely new *tag handler* class, JSTL has several support / base classes which can be extended / inherited from, for example, `javax.servlet.jsp.tagext.TagSupport`. The process of extending JSTL base classes focuses development time on writing custom code and not traditional *tag handler* methods. A class must realise the `javax.servlet.jsp.tagext.Tag` interface if it is to become a tag handler. However before a new tag can be considered a tag handler it must be associated to a JSP tag library [Bayern, 2002] [Bergsten, 2003] [Brown et al, 2001].

Before the JSP tag library association can be made, a file called a *tag-library descriptor* (TLD) must be created. A TLD is an XML document that describes the main tags contained in a new JSP tag library (see Figure B.6).

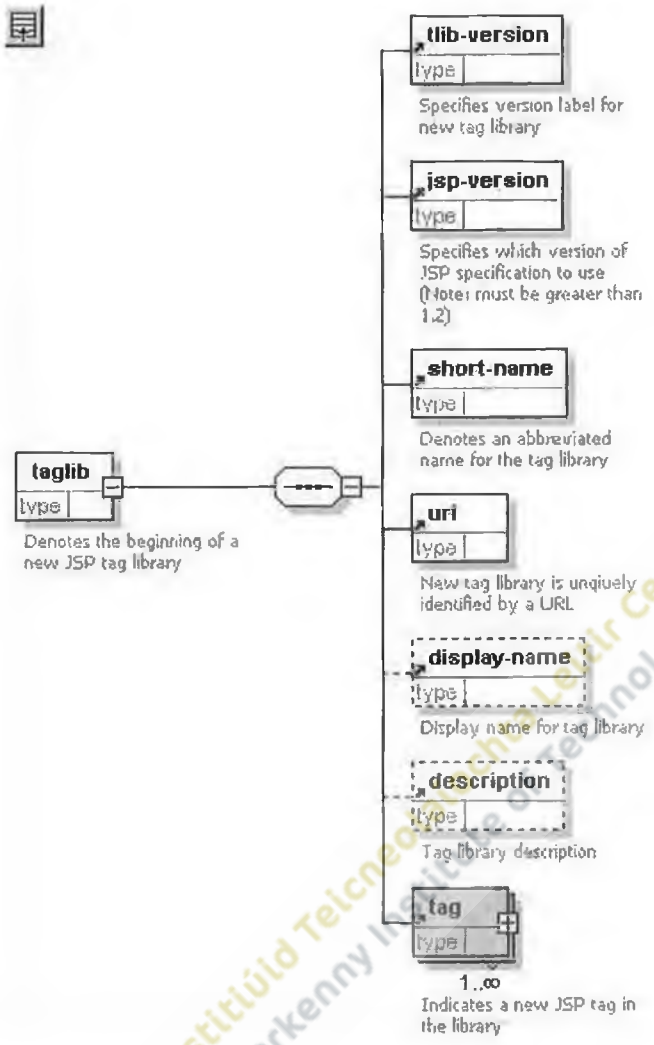


Figure B.6: Diagram of TLD file structure

In additional to the main XML elements of the TLD, The `<taglib>` element has a child `<tag>` element for each tag (see Figure B.7).

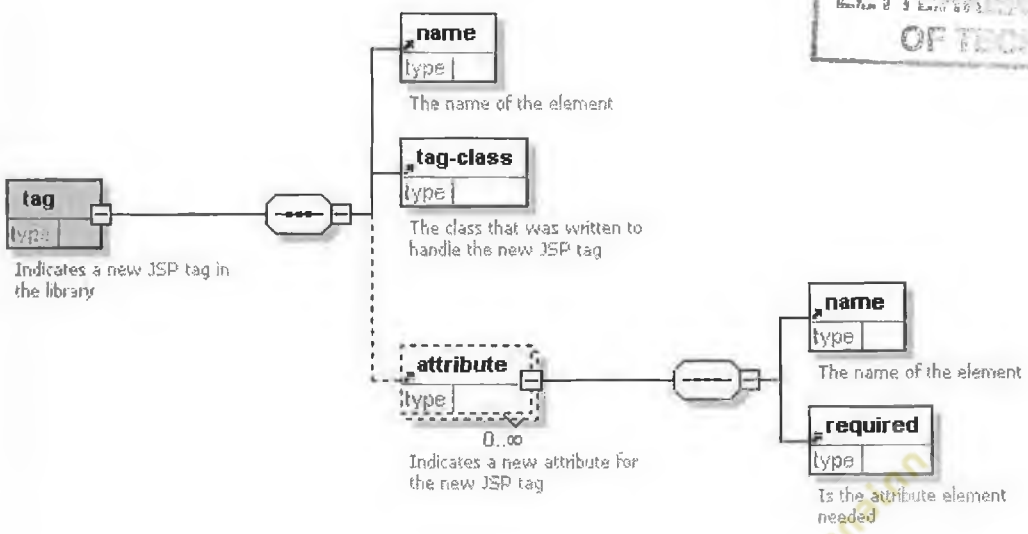


Figure B.7: Diagram of TLD file structure Tag XML element

The following process must be followed before a developer can use a new tag from a new tag library [Bayern, 2002] [Bergsten, 2003]:

a) A developer must copy their new TLD XML file to the *WEB-INF* directory (see section 2.3.3);
b) The developer must copy their new tag handler classes to the *lib* or *classes* directory (see section 2.3.3);
c) The new tag library must be imported into the JSP using the `<%@ taglib %>` directive.

B.4.4 Advantages of JSTL

The following are the advantages associated with using the JSTL:

a) Internationalisation and Localization support

   The JSTL `fmt` tag library provides a rich set of language support mechanisms in the form of built in tags [Brown et al, 2001] [Heaton, 2002]

b) Compatibility with web WYSIWYG development tools

   As the JSTL expression language is XML compliant, it is easier for web WYSIWYG tools (such as Macromedia Dreamweaver) to parse the intermixed HTML and JSTL, therefore these combined mark-up languages can be display

in more readable format that graphic designers and developers can understand [Bayern, 2002] [Heaton, 2002].

c) Readablity

Compared to servlet and JSP applications, JSTL true power is through its ease of readablity. Since graphic / web designers do not come from a computer science background, they find it difficult to understand programming language scriptlet (JSP) which is intermixed with their HTML. As JSTL is based on XML (which is similar in syntax to HTML) these designers have some conceptual awareness of how the JSP page is formed and could even place JSTLs into the page themselves.

On the flip side, since JSTL uses automatic JavaBean inspection, programmers can simplify the JSP scriptlet syntax (which is really normal Java code after the JSP page has been parsed) by using JSTL [Bayern, 2002] [Brown et al, 2001] [Heaton, 2002].

B.4.5 Disadvantages of JSTL

The following are the disadvantages associated with using the JSTL:

a) Performance

A performance lag will occur during JSP page execution, as JSTL uses significant amount of extra server processing than JSP scriptlet. The reason behind this is that JSTL uses significant amounts of introspection for JavaBeans and XML parsing for the JSTL expression language [Heaton, 2002].

b) Learning curve

There is a significant learning curve to fully understand the JSTL specification. Thus the specification adds another layer of complexity for less experienced developers.

c) Extra overhead

Compared to JSP scriptlet, JSTL is wonderful for creating simplistic JSP pages however experienced developers may judge that there is an extra work in creating a new JSTL XML tag compared to writing JSP scriptlet (which they already know) [Heaton, 2002].

d) Extensibility

JSTL is not as extensive language as JSP scriptlet, as JSTL is still an evolving specification that doesn't allow the full use of all other Java classes as the way JSP scriptlet does [Heaton, 2002].

e) Database security

JSTL Database library promotes the use of Database functionality from within an JSP, this maybe problematic as security breeches may enable a hacker direct access to your Database resource. Therefore for larger applications, it is better to separate / hide this functionality by moving the Database access to a JavaBean or another Java class [Bayern, 2002].

B.5 Conclusions

This chapter has provided an insight on other competing Java based solutions for the fundamental problems to JSP. It has described what these technologies are and how do they work. But the chapter has also provided an objective view towards their strong and weak points. Therefore to further this discussion, we must now provide an independent and objective performance benchmark using the dissertation's new framework architecture and several competing alternatives.

Appendix C

Benchmark One Results

(Threads 1 - RampUp 0 - Loop 300)

Graph Results

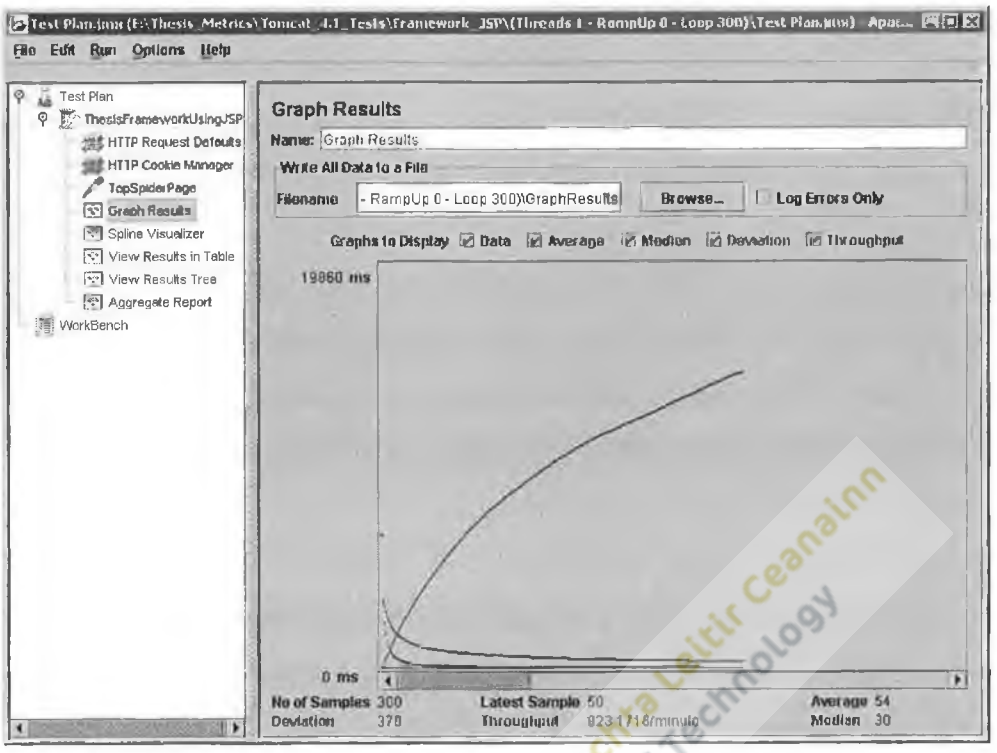


Figure C.1: Graph results of new framework using JSP

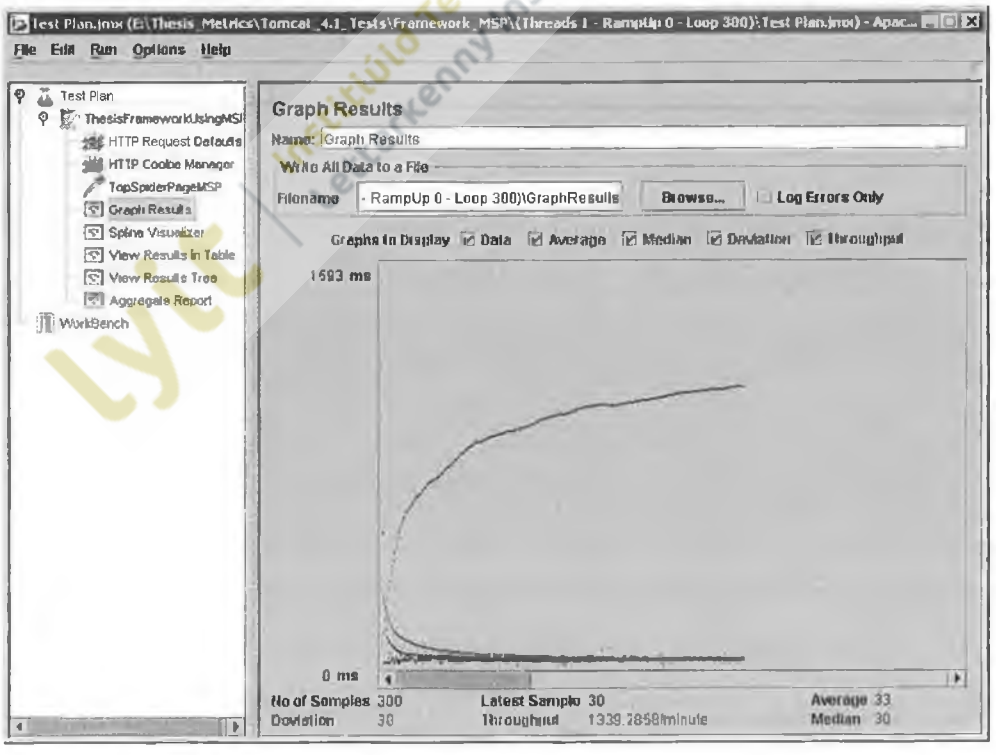


Figure C.2: Graph results of new framework using MSP

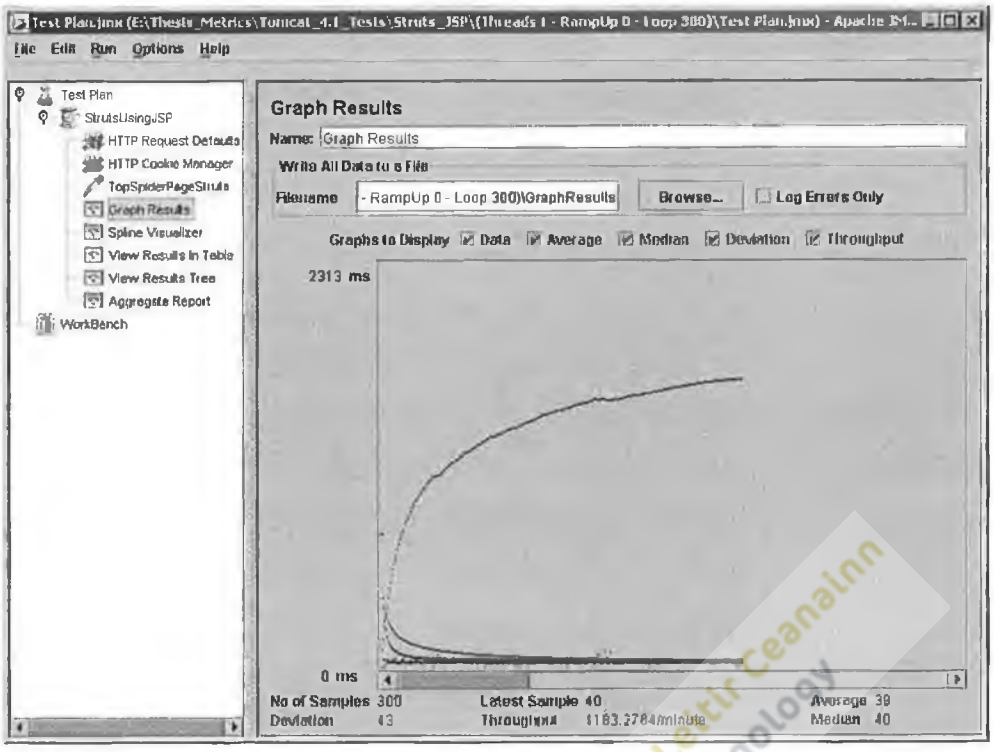


Figure C.3: Graph results of Apache Struts using JSP

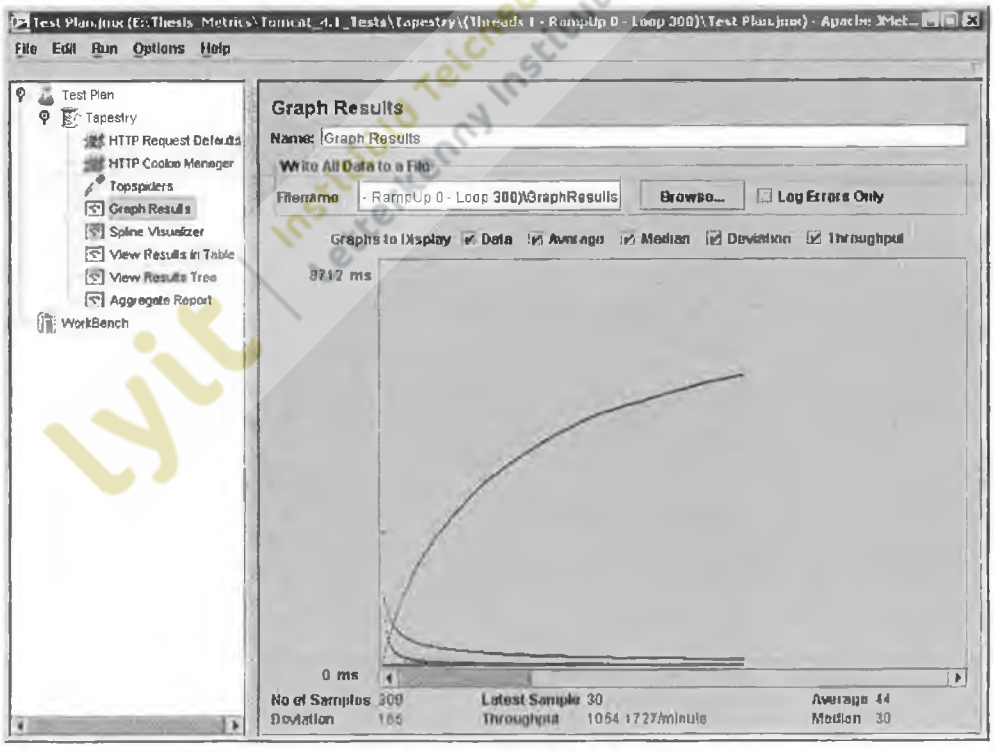


Figure C.4: Graph results of Apache Tapestry

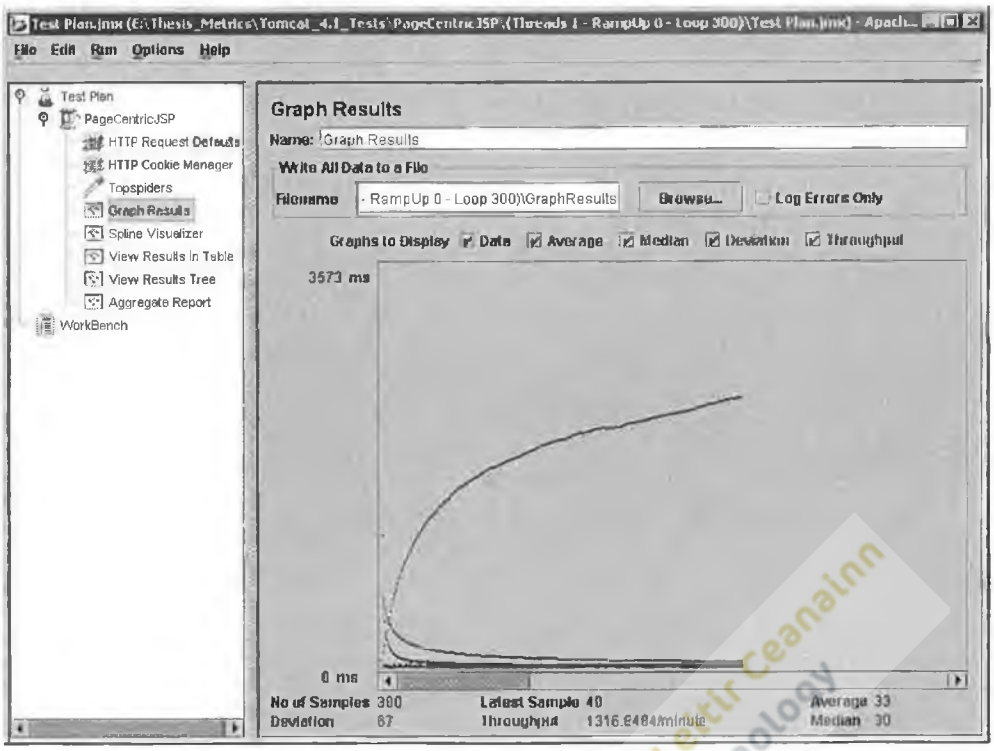


Figure C.5: Graph results of page-centric JSP

Spline Visualiser

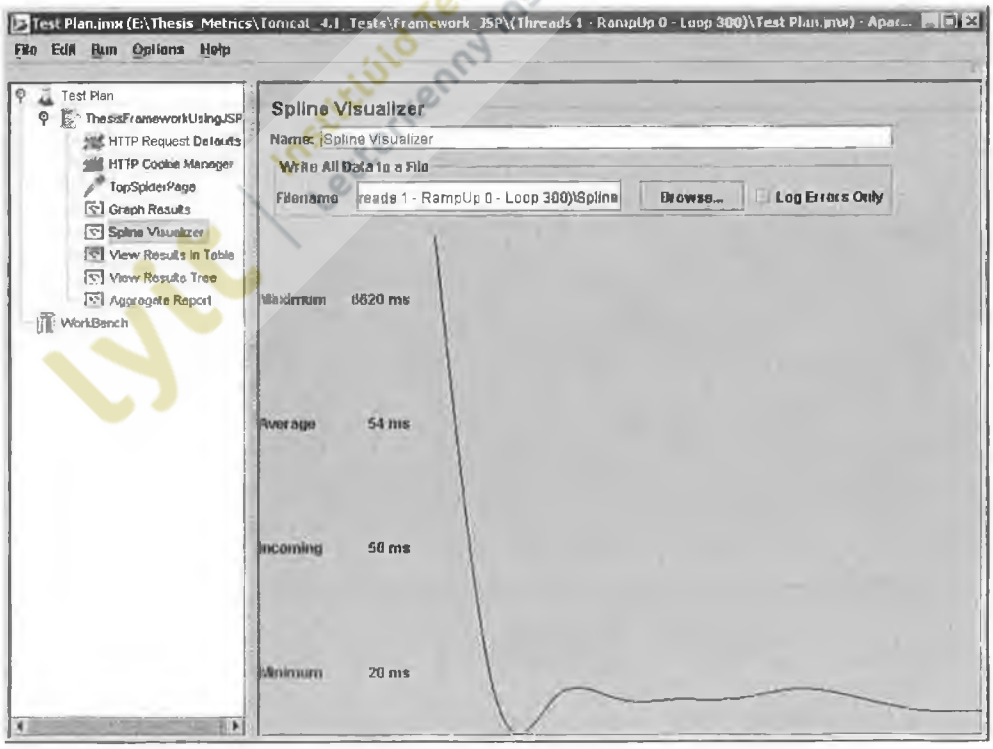


Figure C.1: Spline visualiser of new framework using JSP

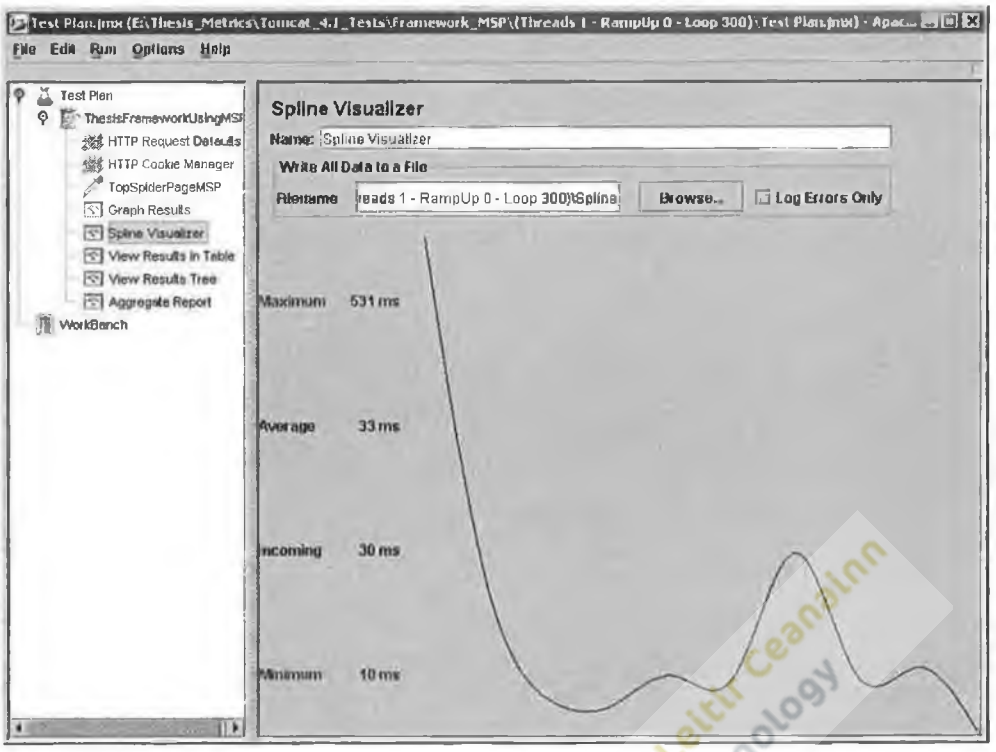


Figure C.2: Spline visualiser of new framework using MSP

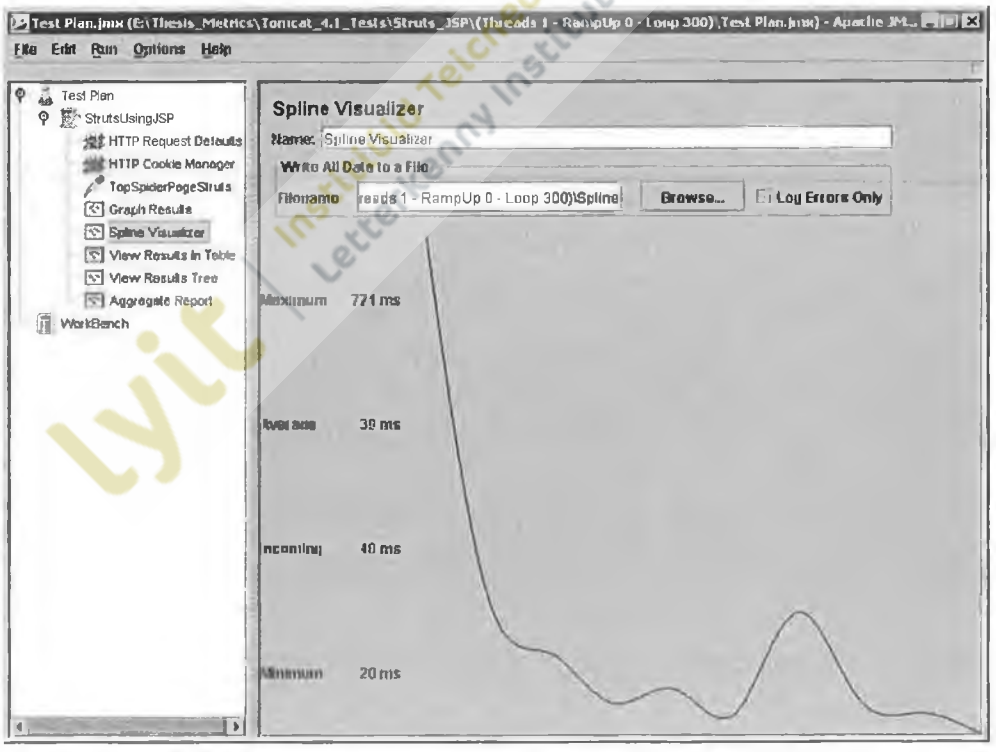


Figure C.3: Spline visualiser of Apache Struts using JSP

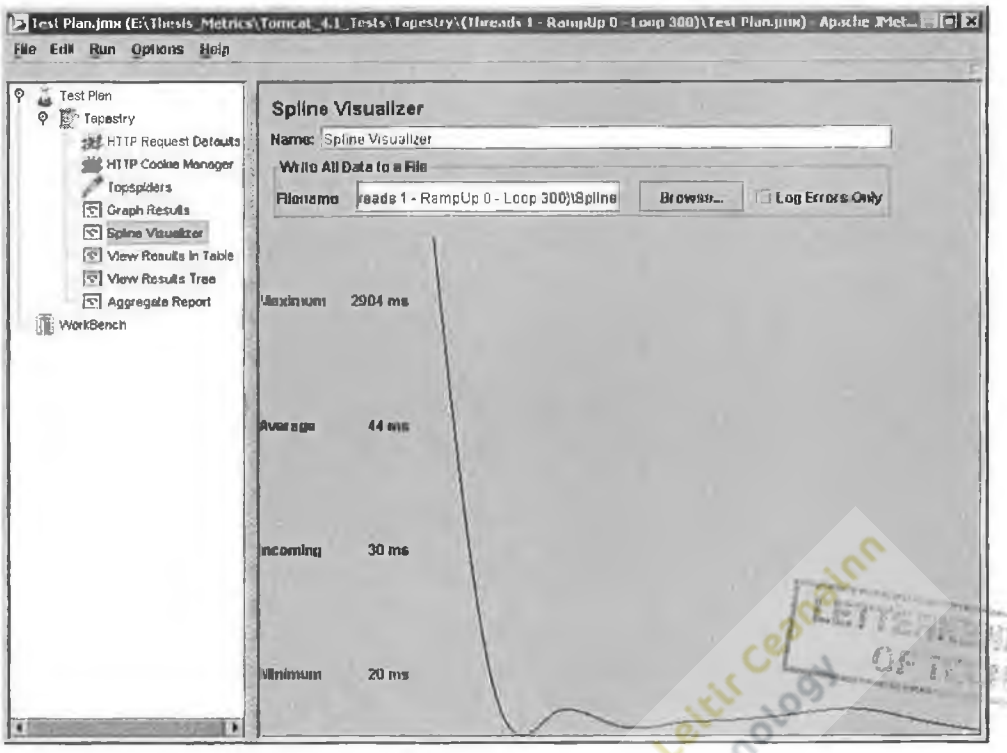


Figure C.4: Spline visualiser of Apache Tapestry

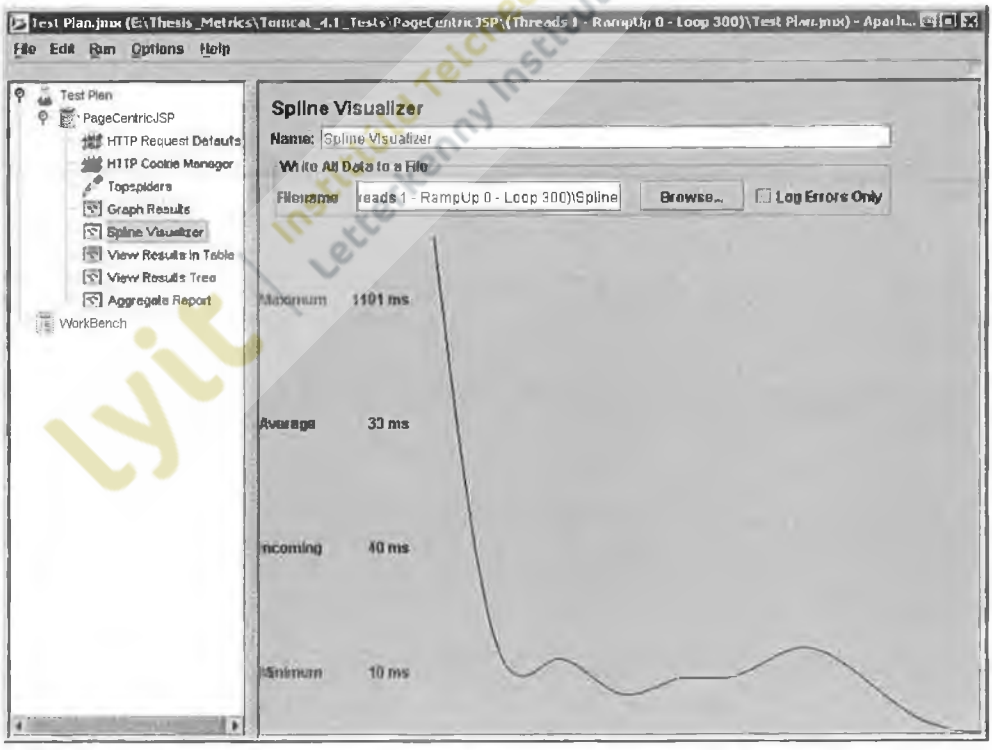


Figure C.5: Spline visualiser of page-centric JSP

Aggregate Report

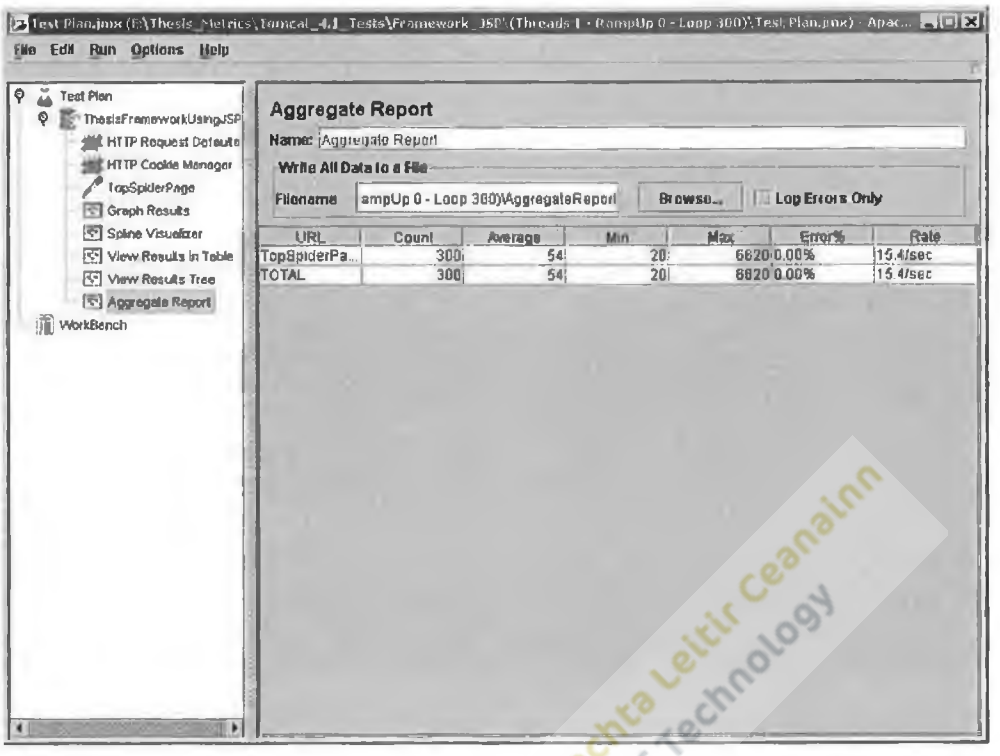


Figure C.11: Aggregate report of new framework using JSP

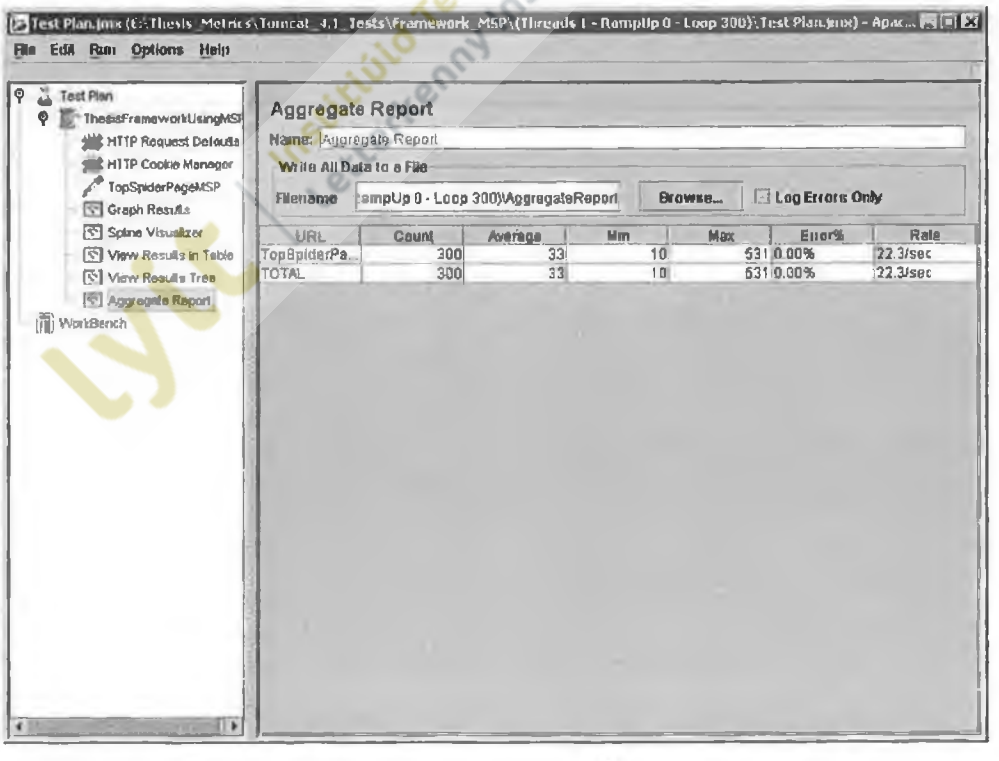


Figure C.12: Aggregate report of new framework using MSP

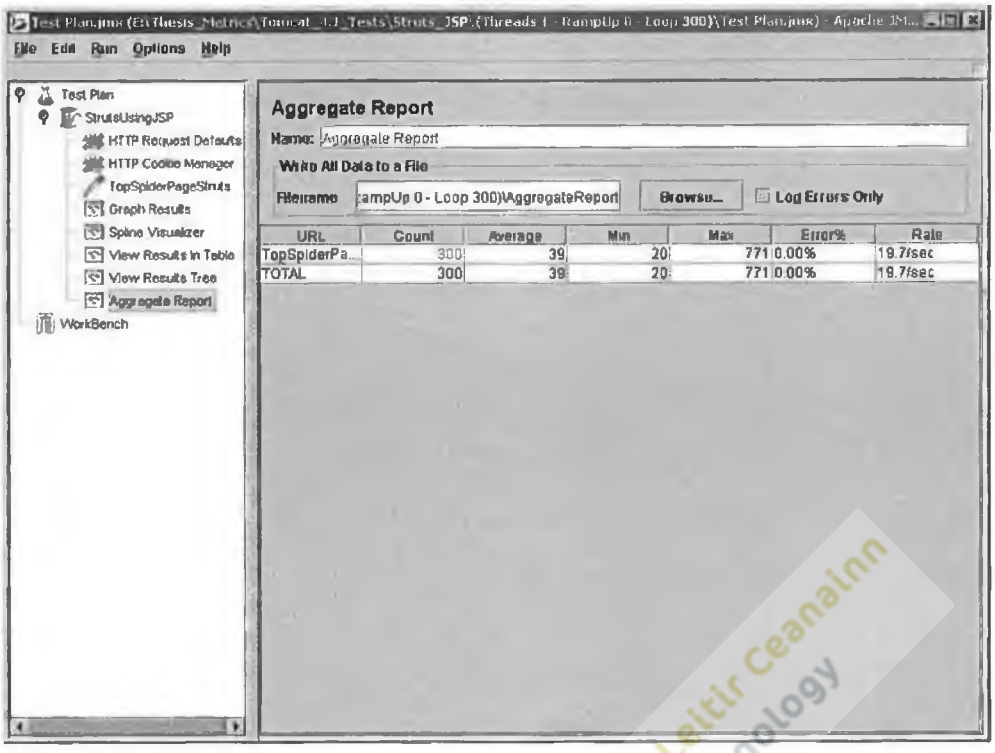


Figure C.13: Aggregate report of Apache Struts using JSP

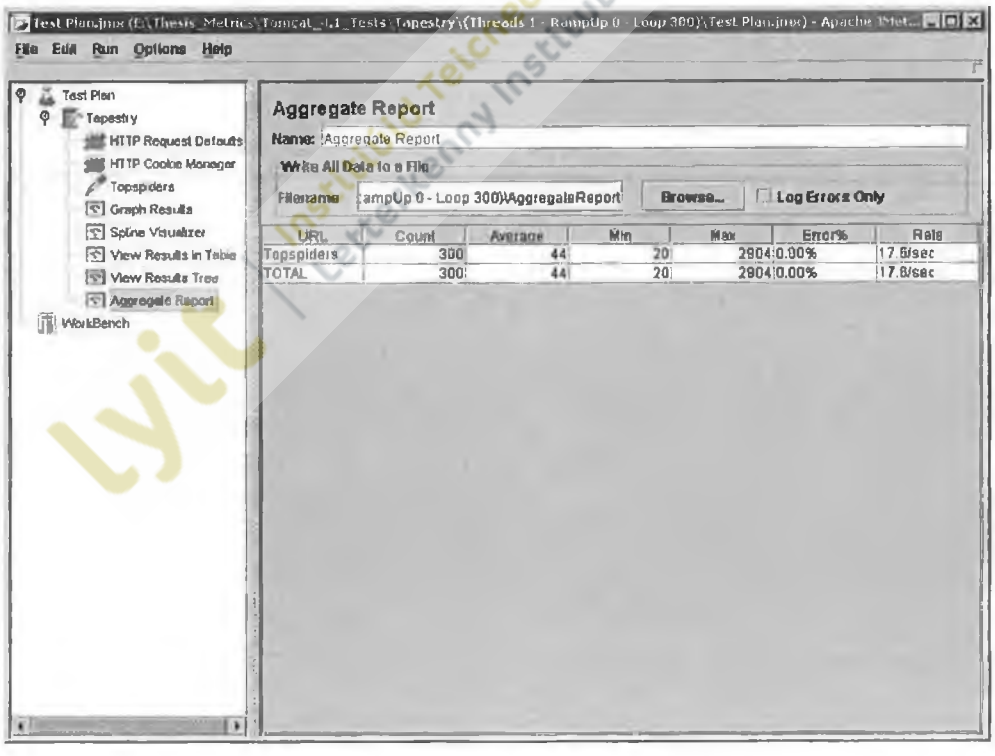


Figure C.14: Aggregate report of Apache Tapestry

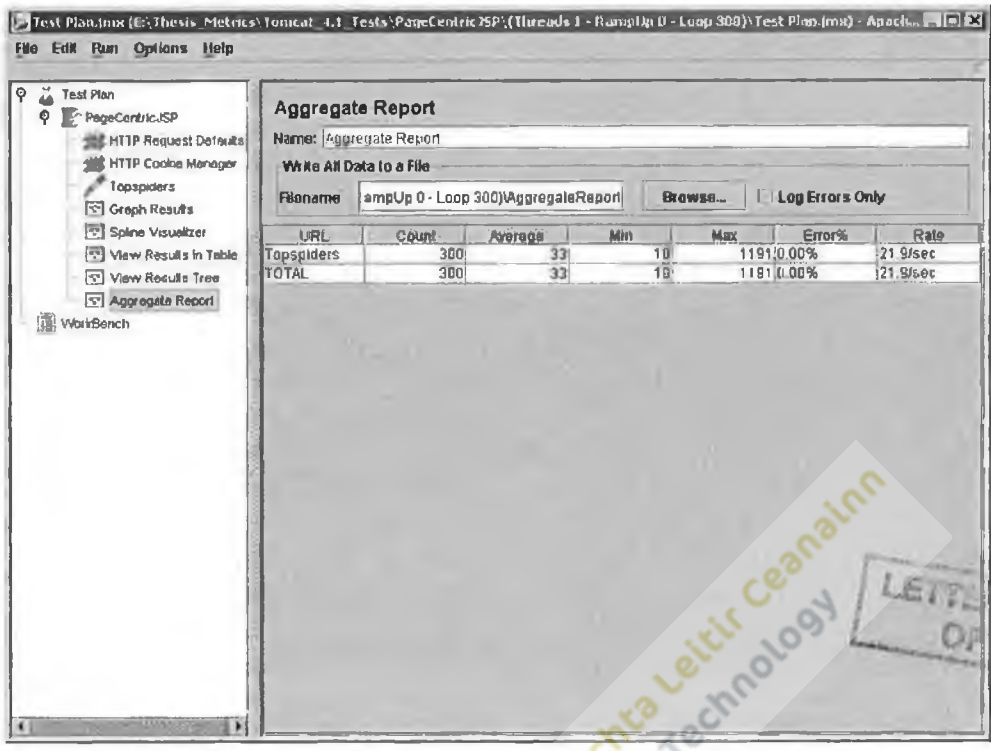


Figure C.15: Aggregate report of page-centric JSP

View Results In Table

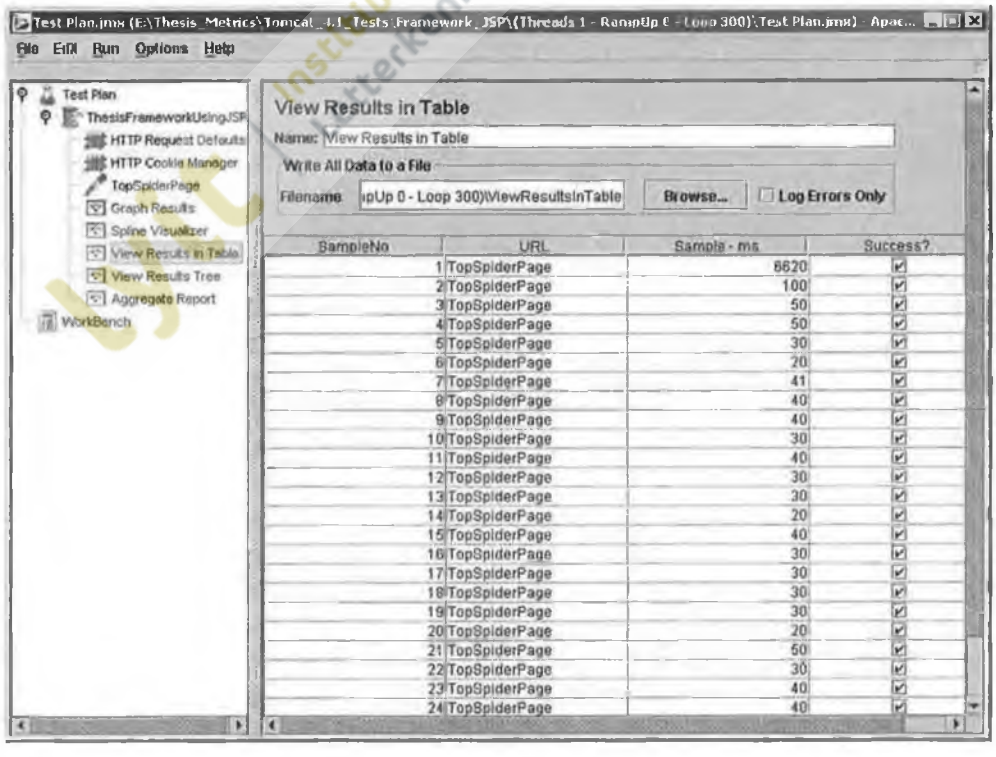


Figure C.16: View results in table of new framework using JSP

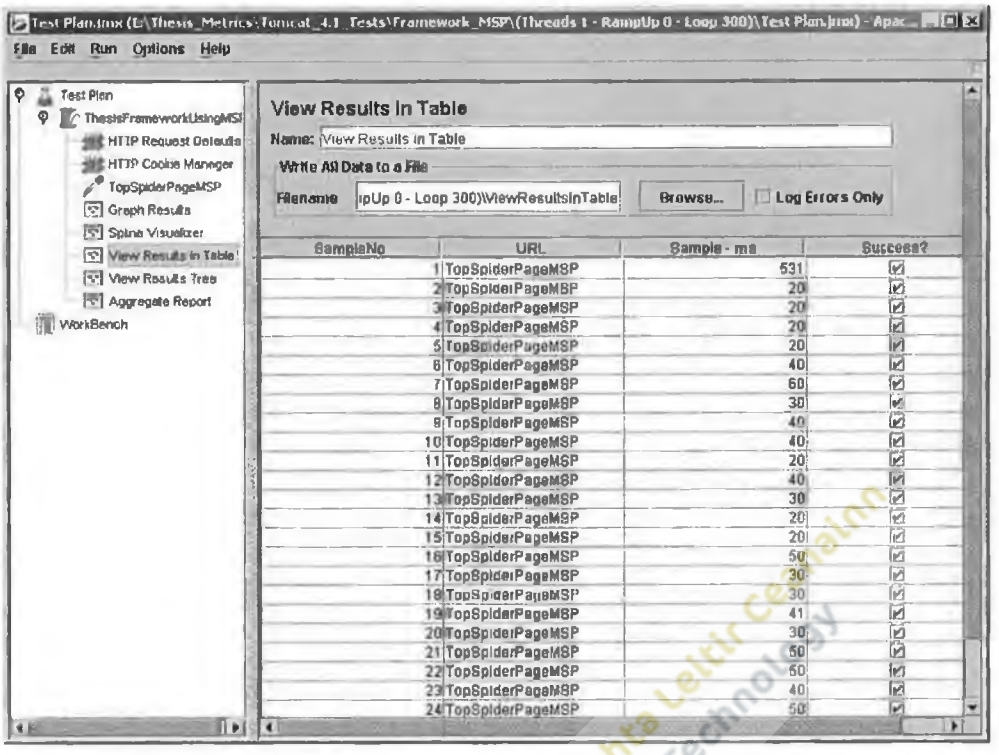


Figure C.17: View results in table of new framework using MSP

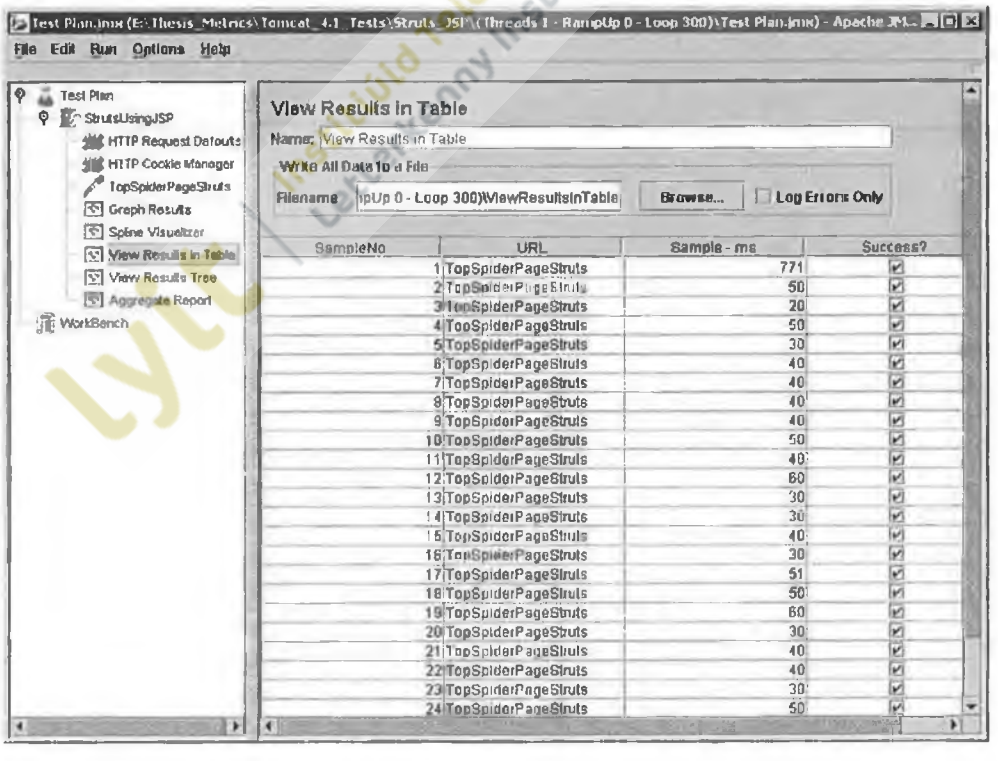


Figure C.18: View results in table of Apache Struts using JSP

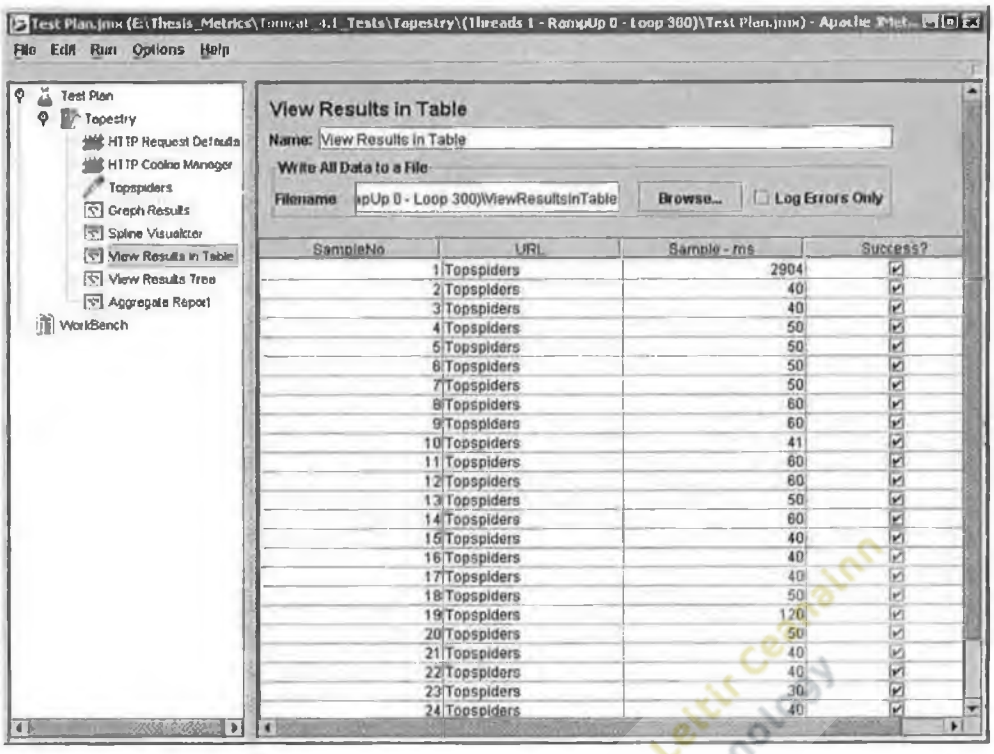


Figure C.19: View results in table of Apache Tapestry

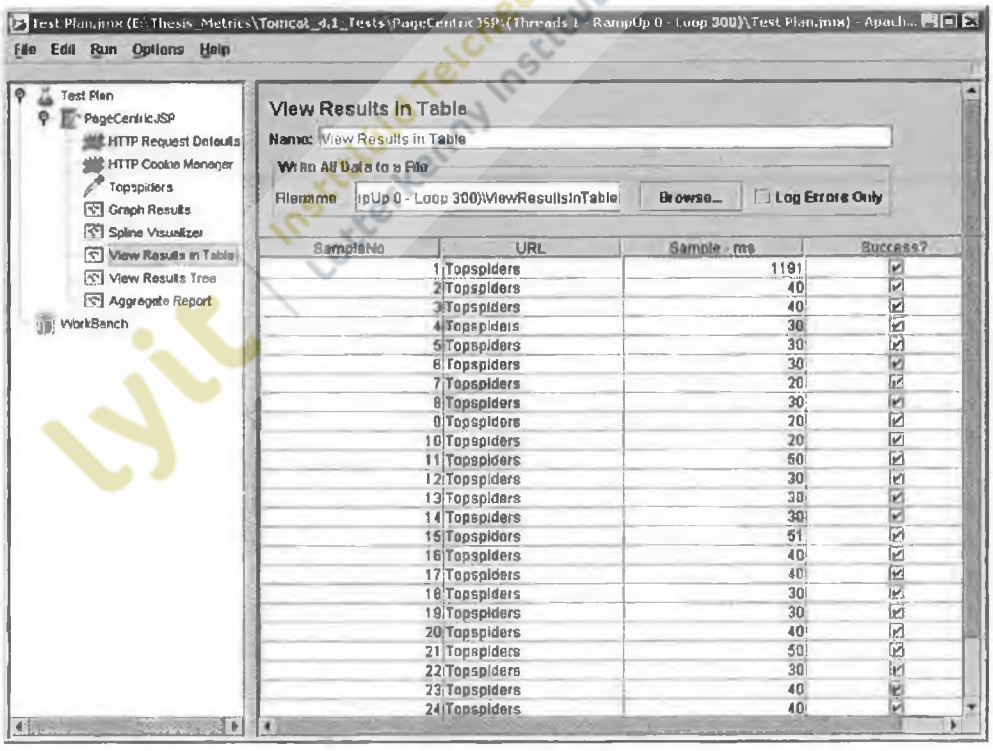


Figure C.20: View results in table of page-centric JSP

View Results in Tree

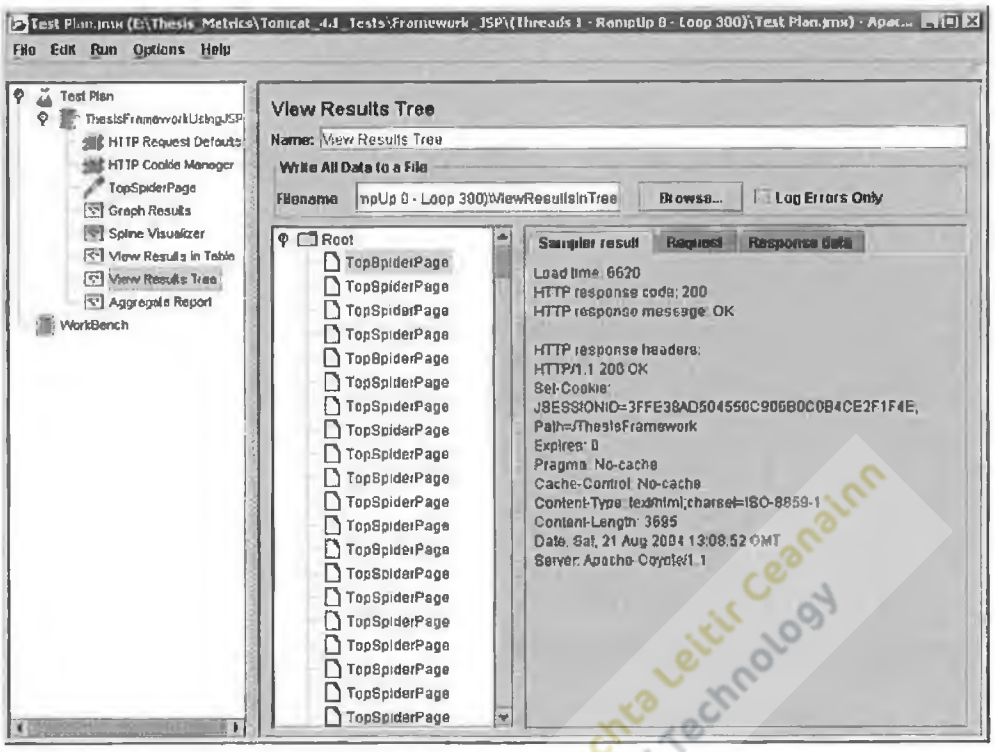


Figure C.21: View results in tree of new framework using JSP

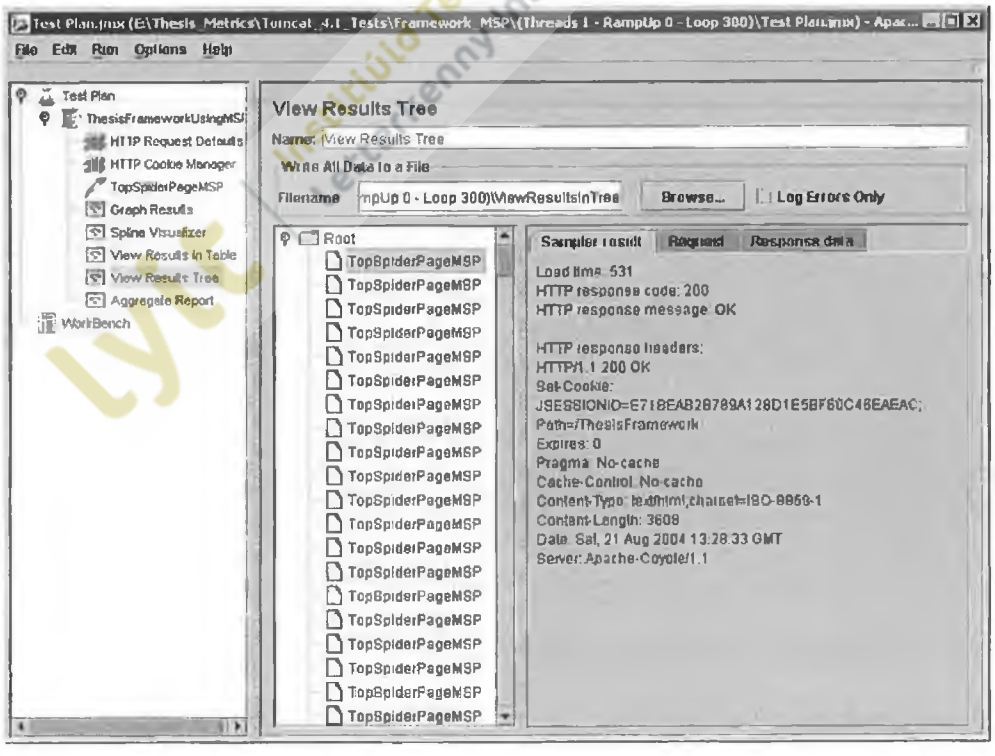


Figure C.22: View results in tree of new framework using MSP

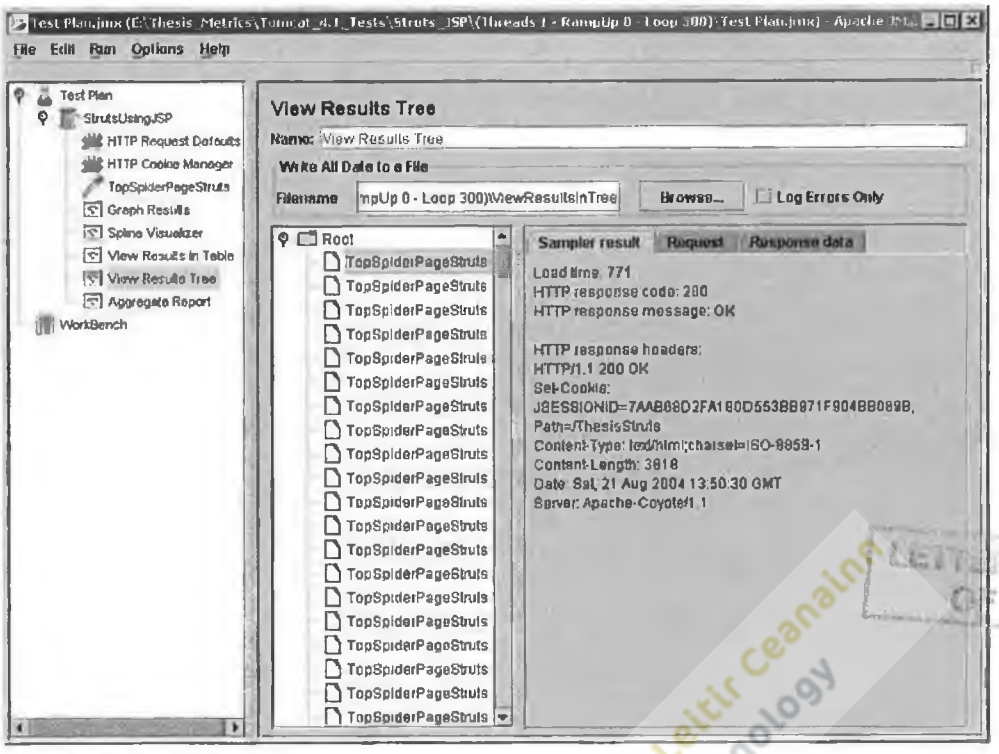


Figure C.23: View results in tree of Apache Struts using JSP

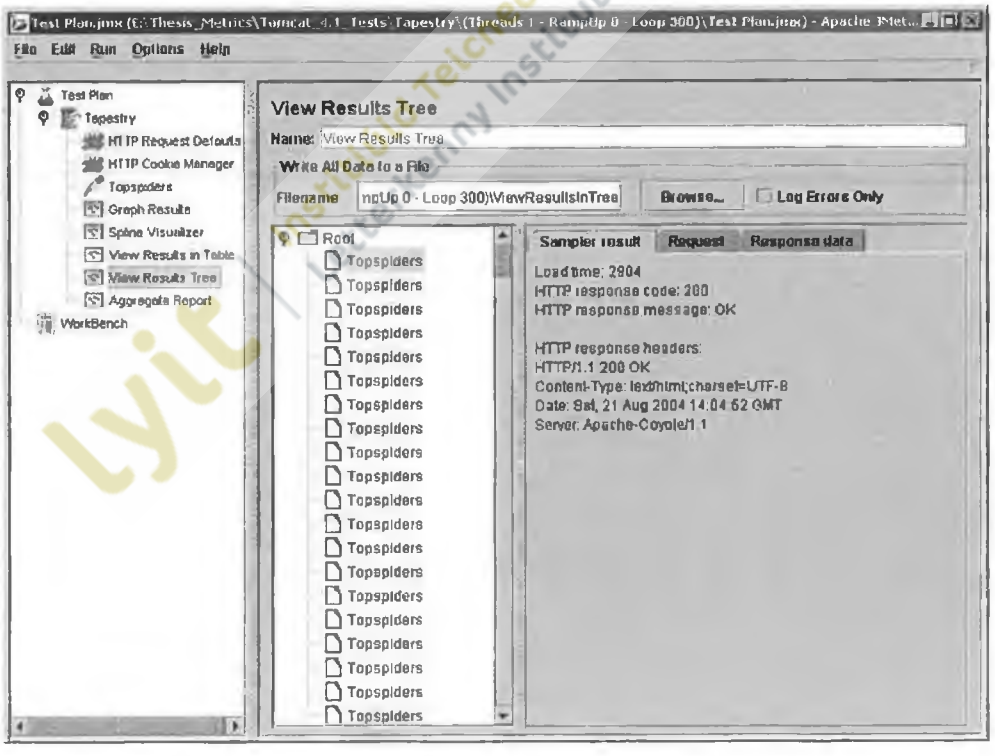


Figure C.24: View results in tree of Apache Tapestry

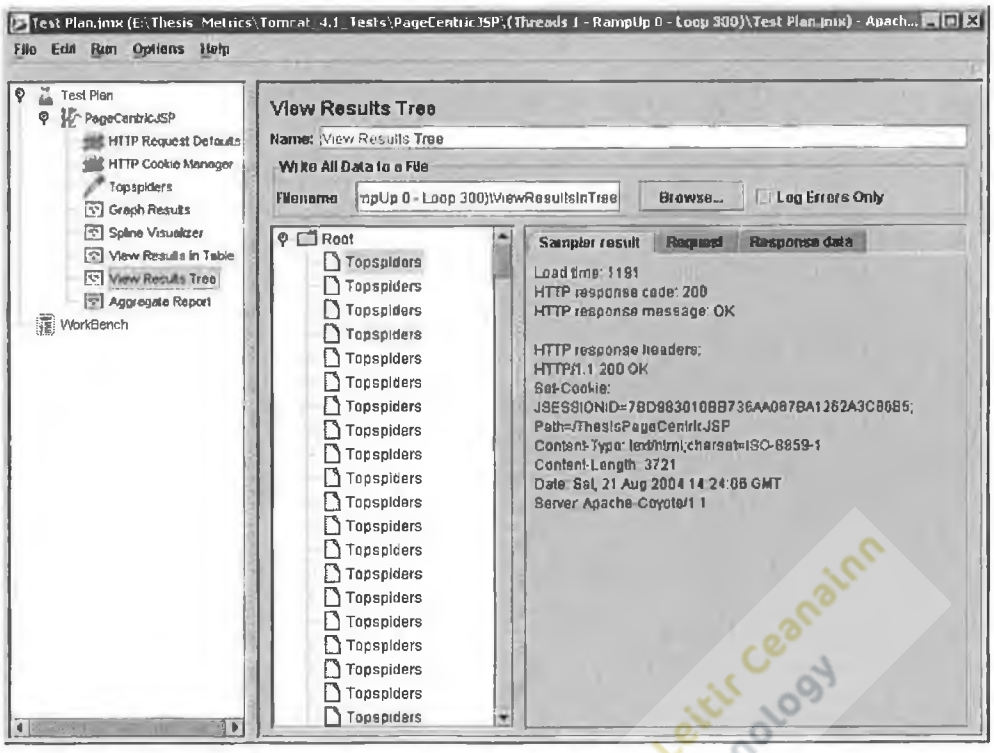


Figure C.25: View results in tree of page-centric JSP

Overall Results

Architecture	minTime	maxTime	Average	Rate	Deviation	Throughput	Median
Framework (JSP)	20	6620	54	15.4	379	923	30
Framework (MSP)	10	531	33	22.3	30	1339	30
Struts	20	771	39	19.7	43	1183	40
Tapestry	20	2904	44	17.6	165	1054	30
PageCentric (JSP)	10	1191	33	21.9	67	1316	30

Table C.1: Benchmark one's overall result

Appendix D

Benchmark Two Results

(Threads 10 - RampUp 2 - Loop 30)

Graph Results

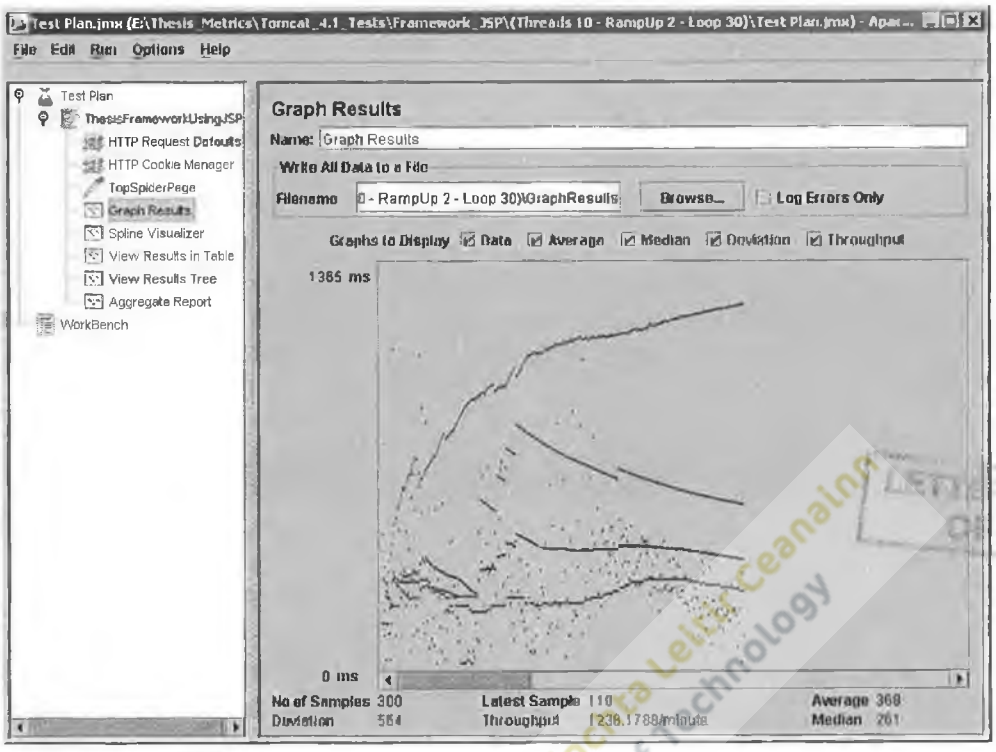


Figure D.1: Graph results of new framework using JSP

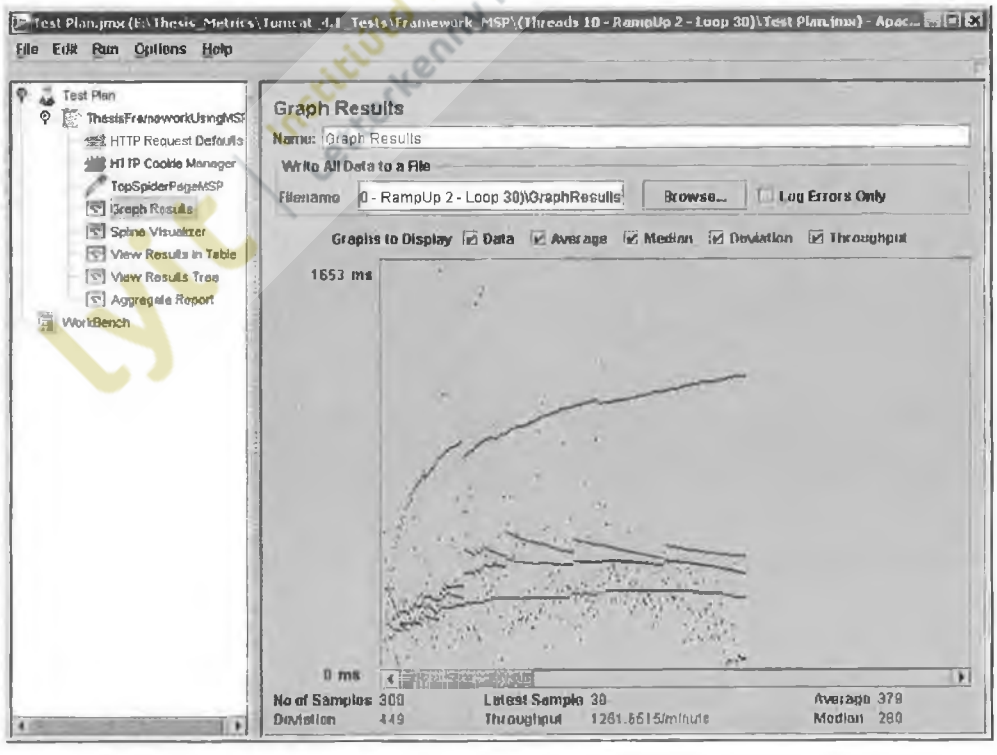


Figure D.2: Graph results of new framework using MSP

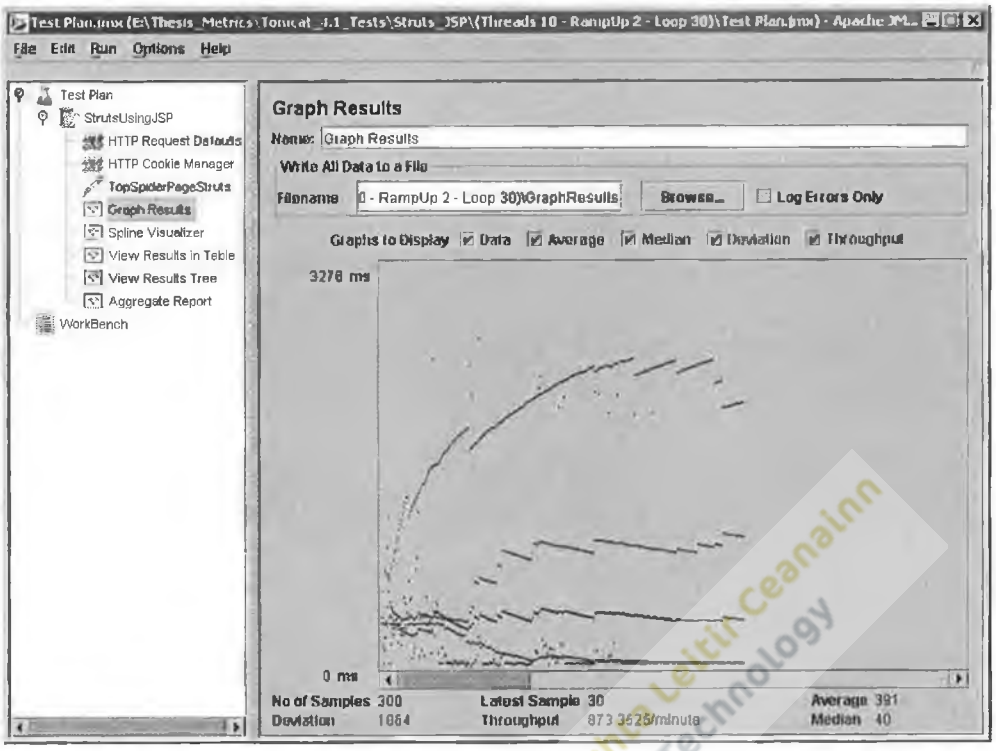


Figure D.3: Graph results of Apache Struts using JSP

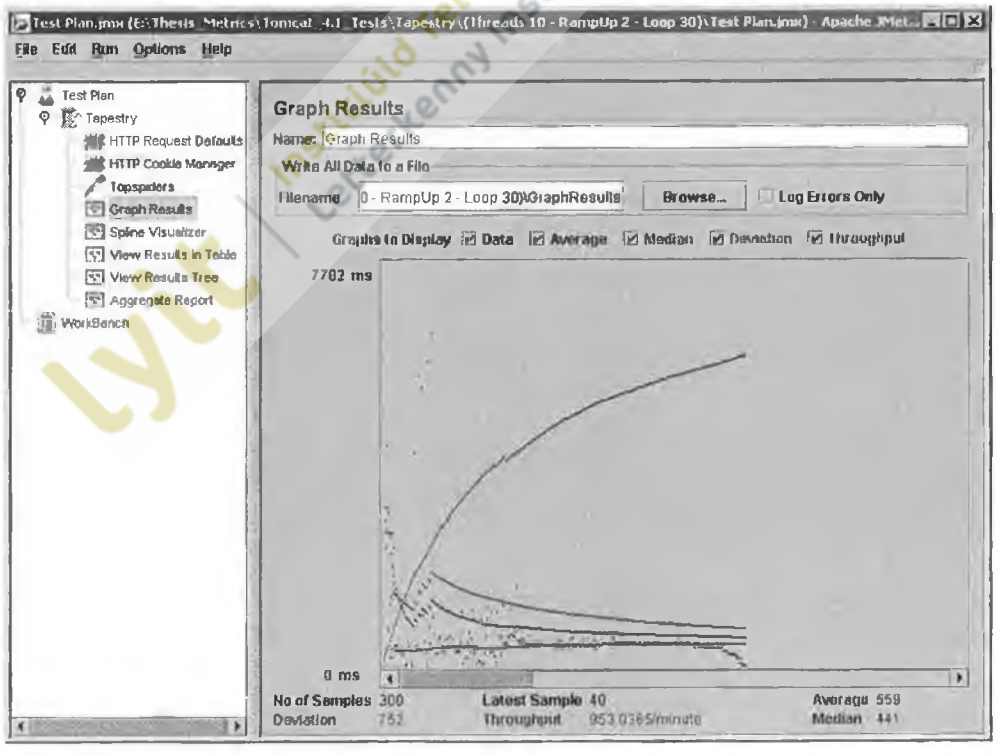


Figure D.4: Graph results of Apache Tapestry

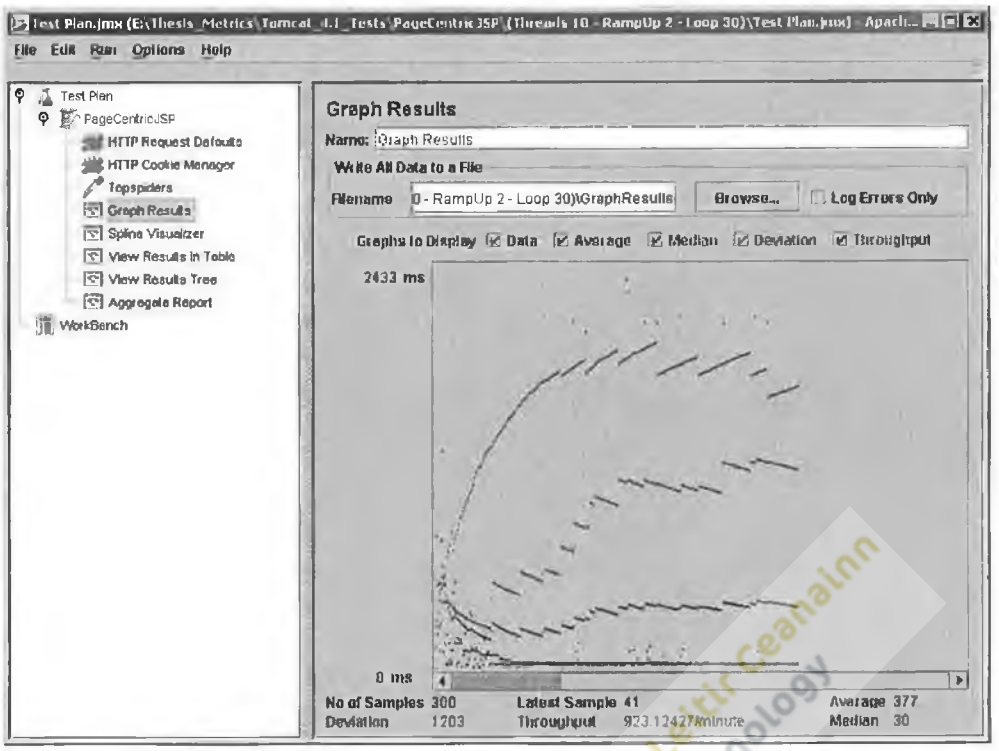


Figure D.5: Graph results of page-centric JSP

Spline Visualiser

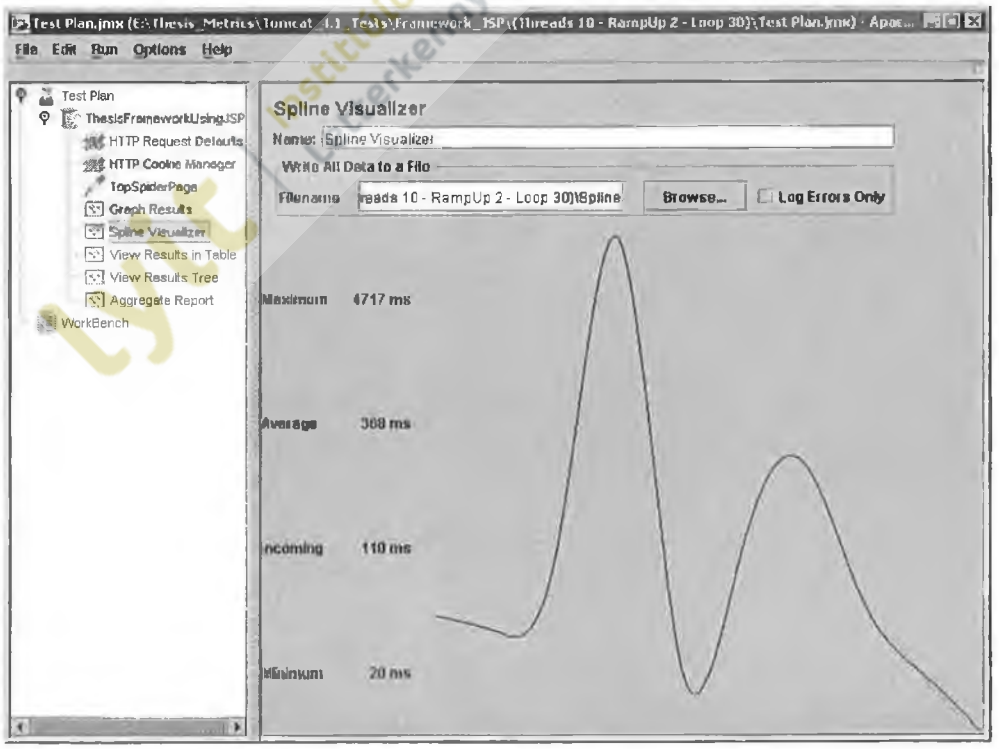


Figure D.6: Spline visualiser of new framework using JSP

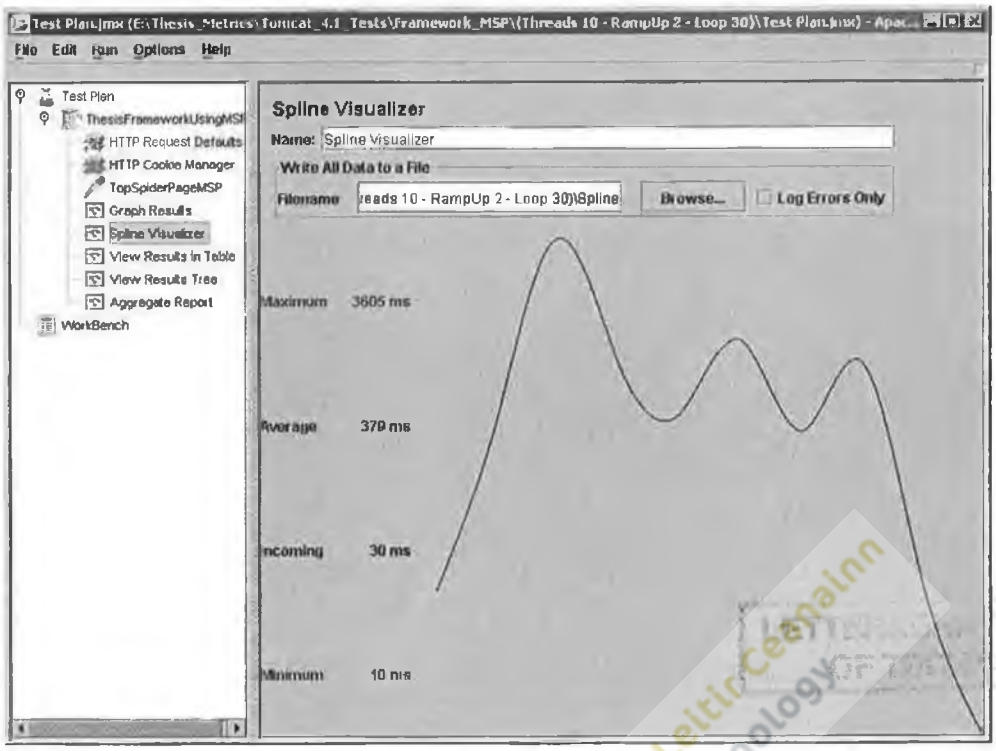


Figure D.7: Spline visualiser of new framework using MSP

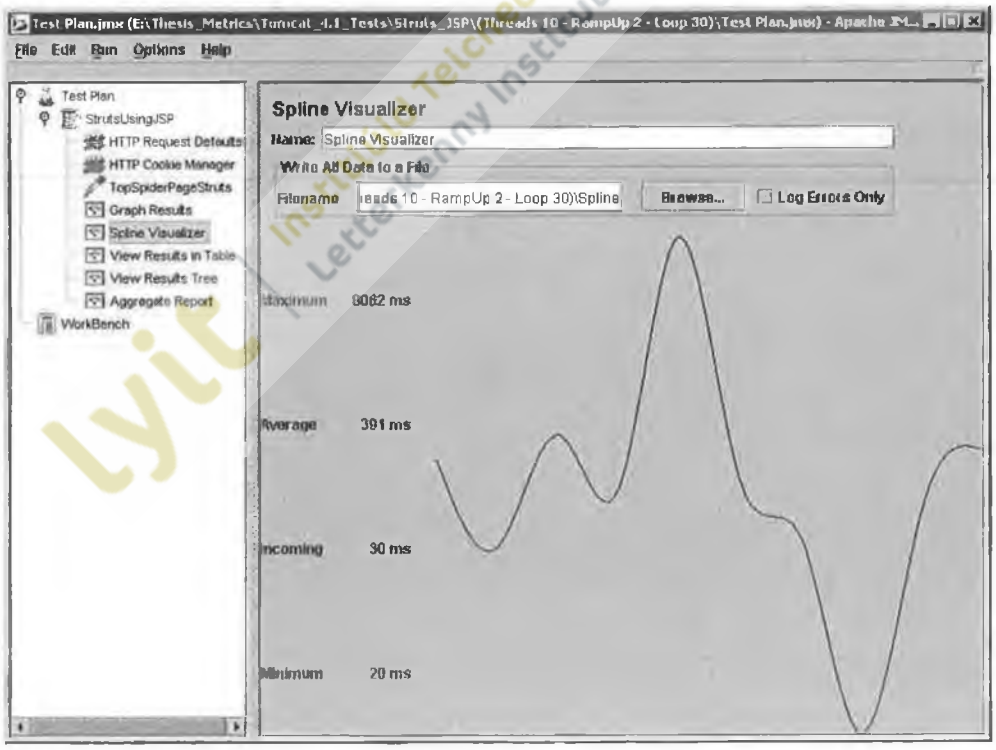


Figure D.8: Spline visualiser of Apache Struts using JSP

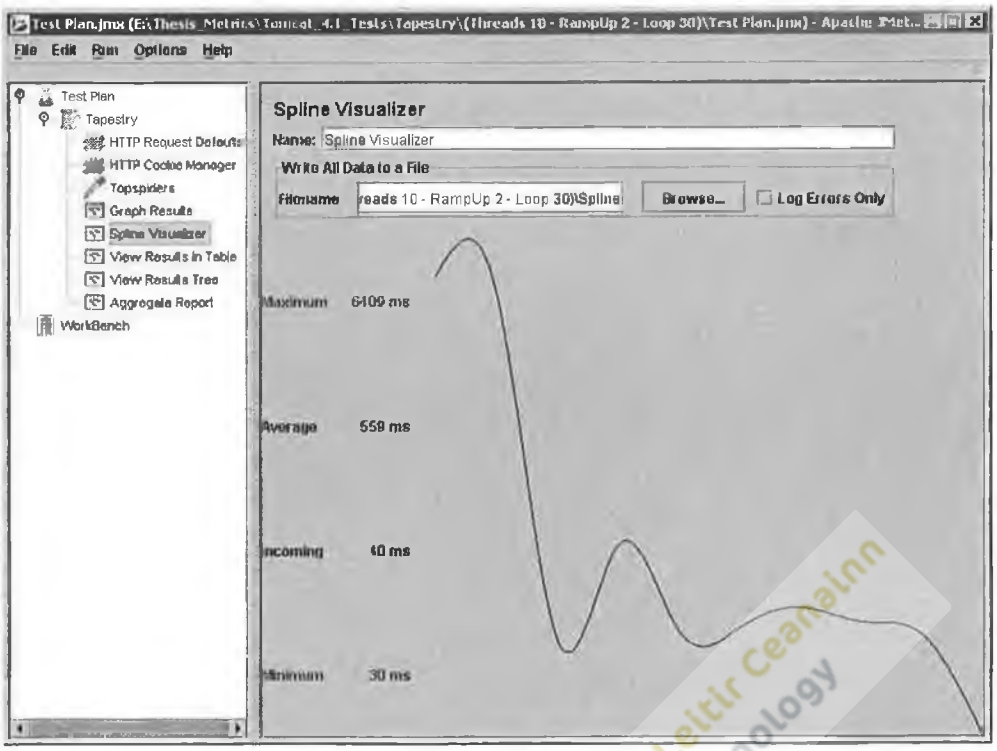


Figure D.9: Spline visualiser of Apache Tapestry

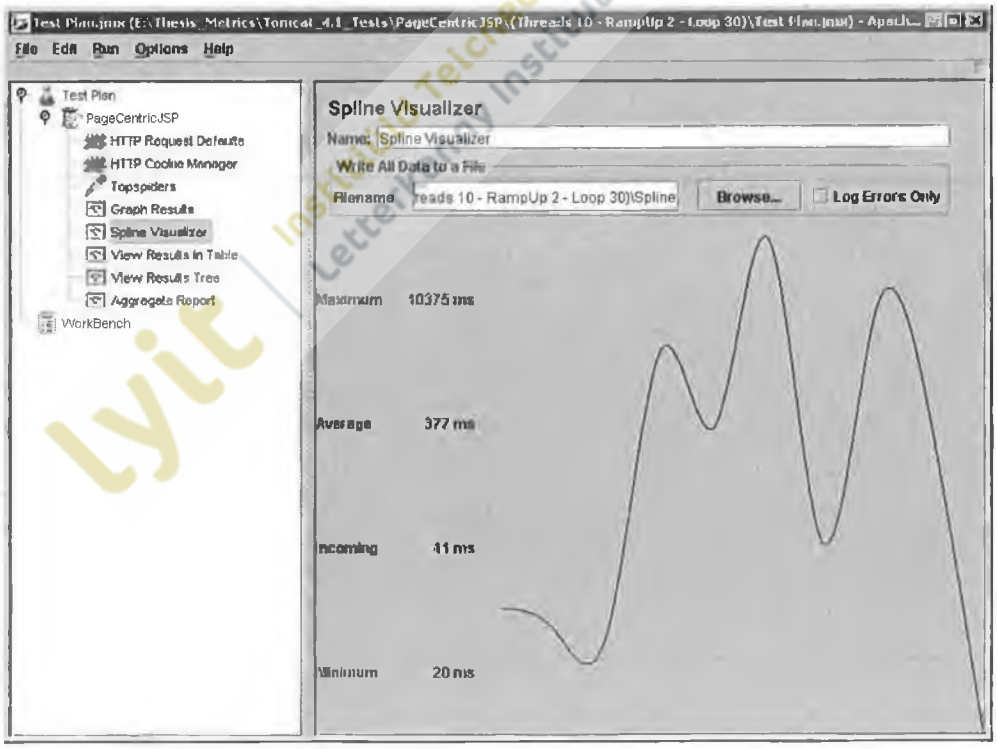


Figure D.10: Spline visualiser of page-centric JSP

Aggregate Report

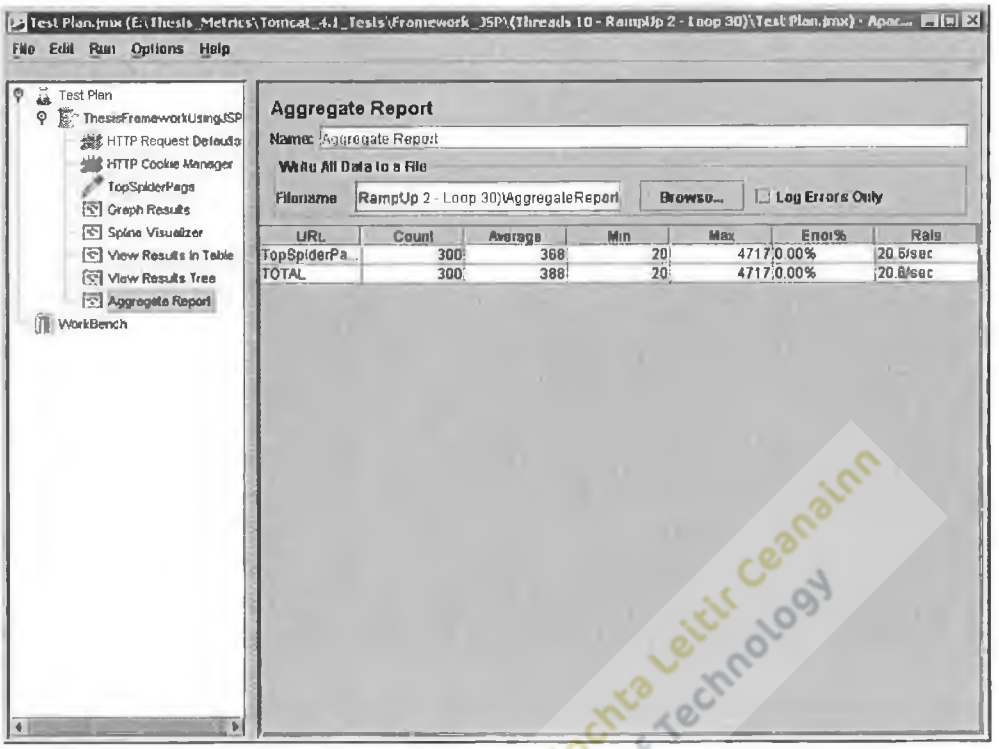


Figure D.11: Aggregate report of new framework using JSP

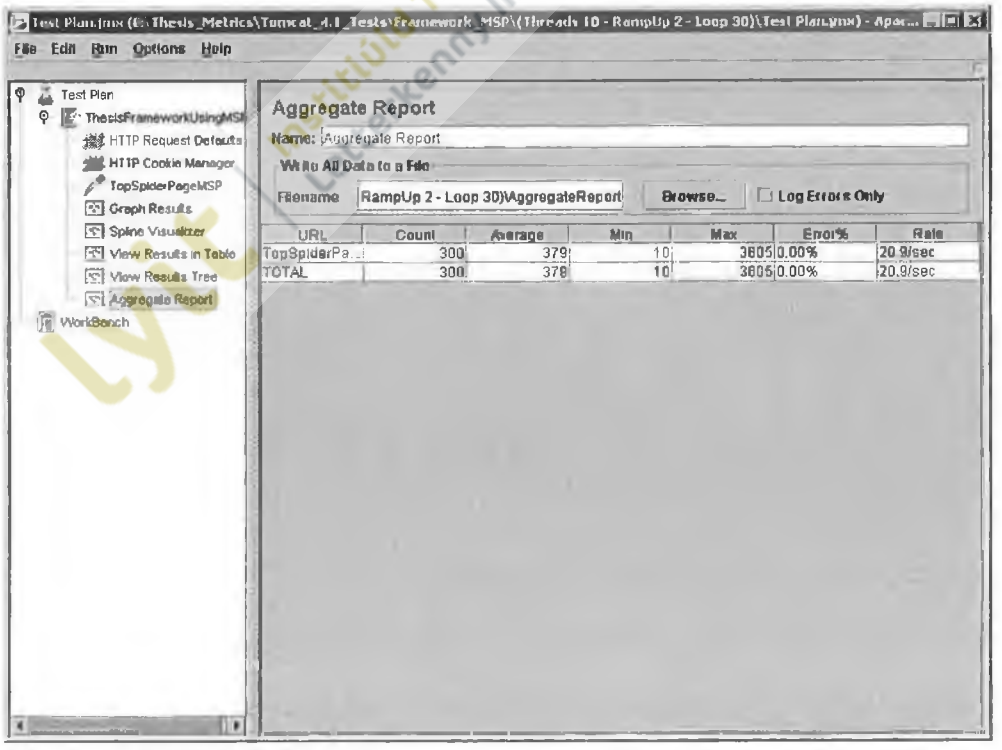


Figure D.12: Aggregate report of new framework using MSP

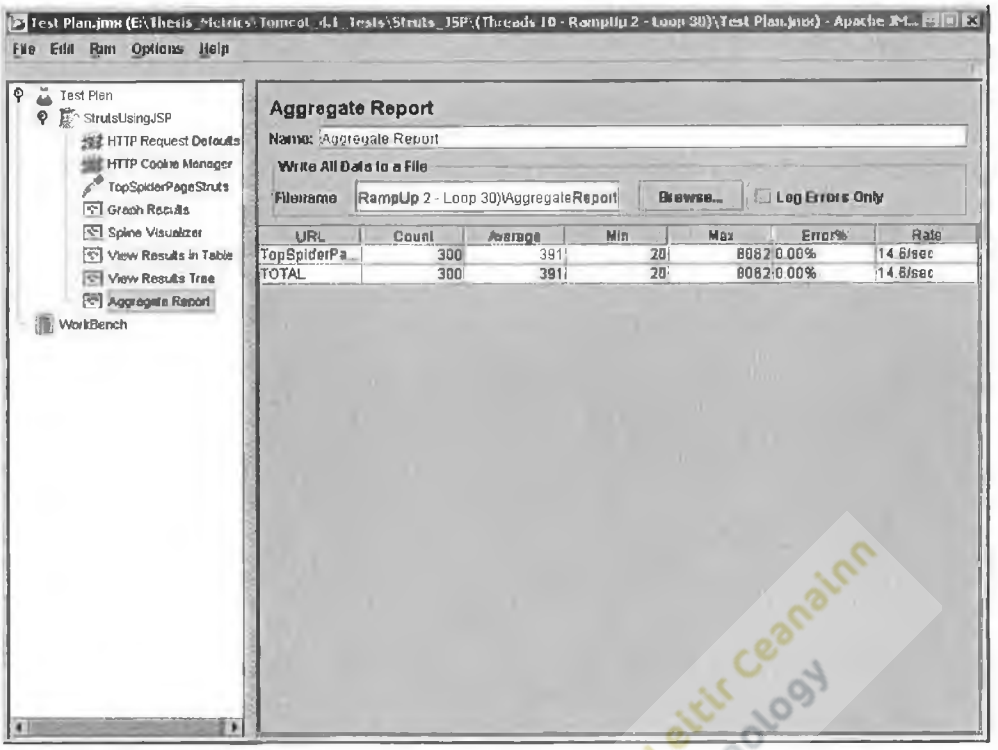


Figure D.13: Aggregate report of Apache Struts using JSP

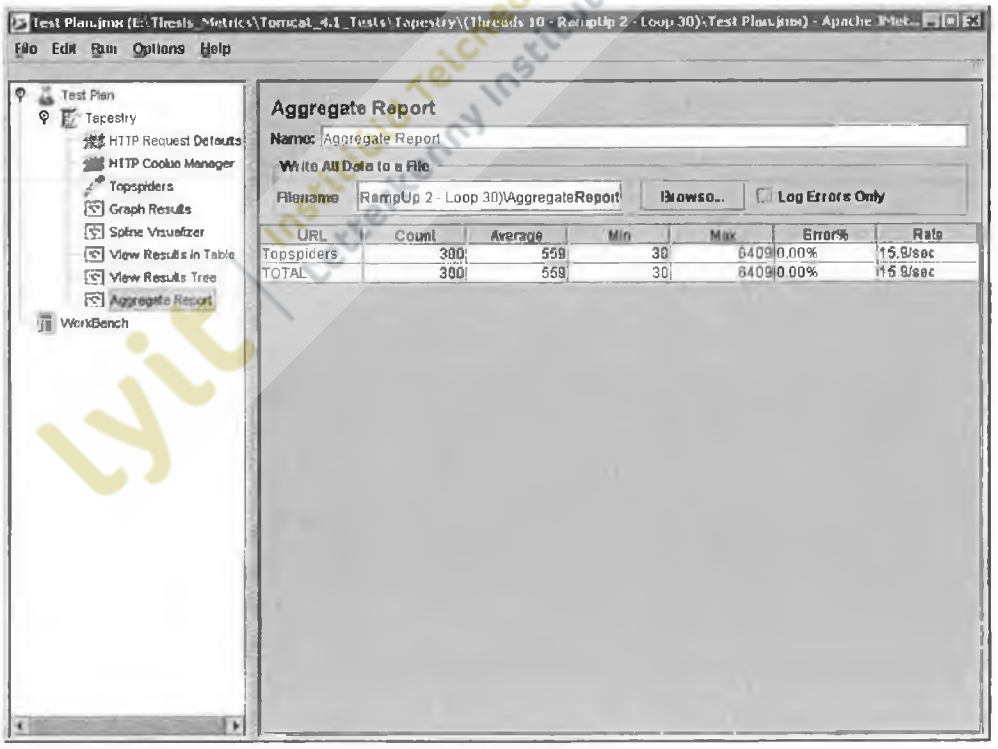


Figure D.14: Aggregate report of Apache Tapestry

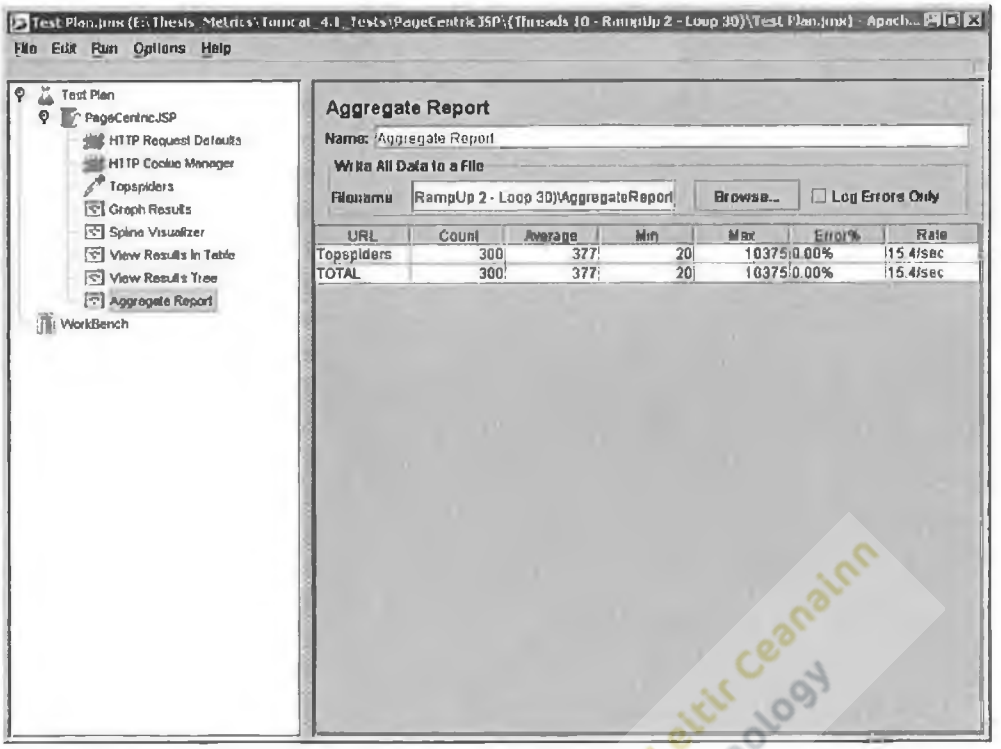


Figure D.15: Aggregate report of page-centric JSP

View Results In Table

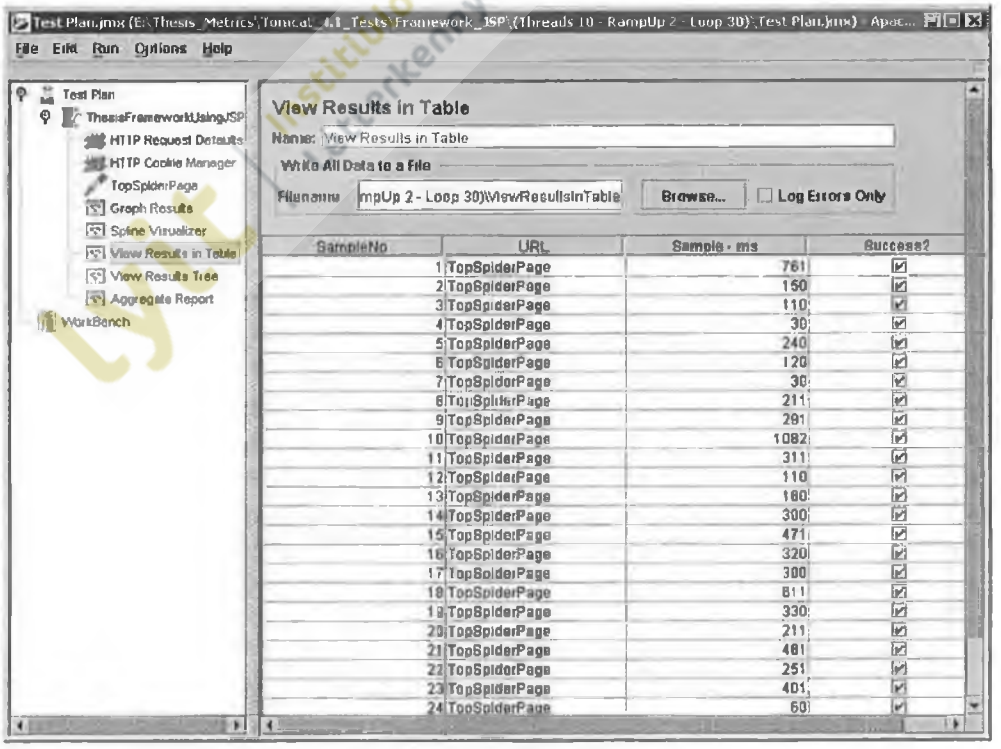


Figure D.16: View results in table of new framework using JSP

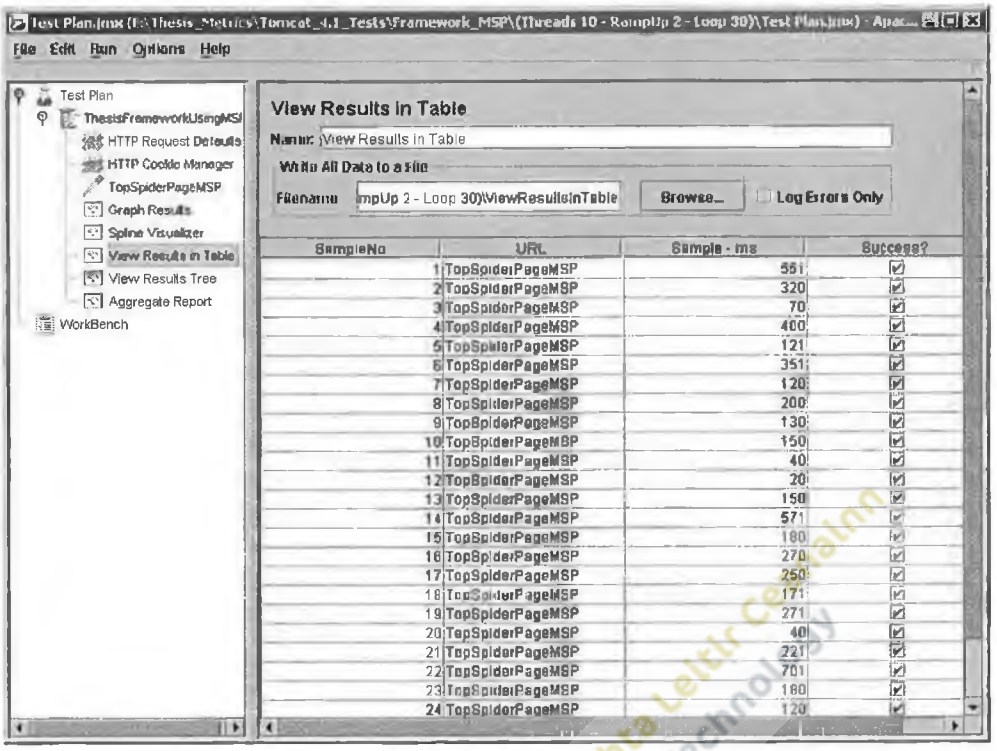


Figure D.17: View results in table of new framework using MSP

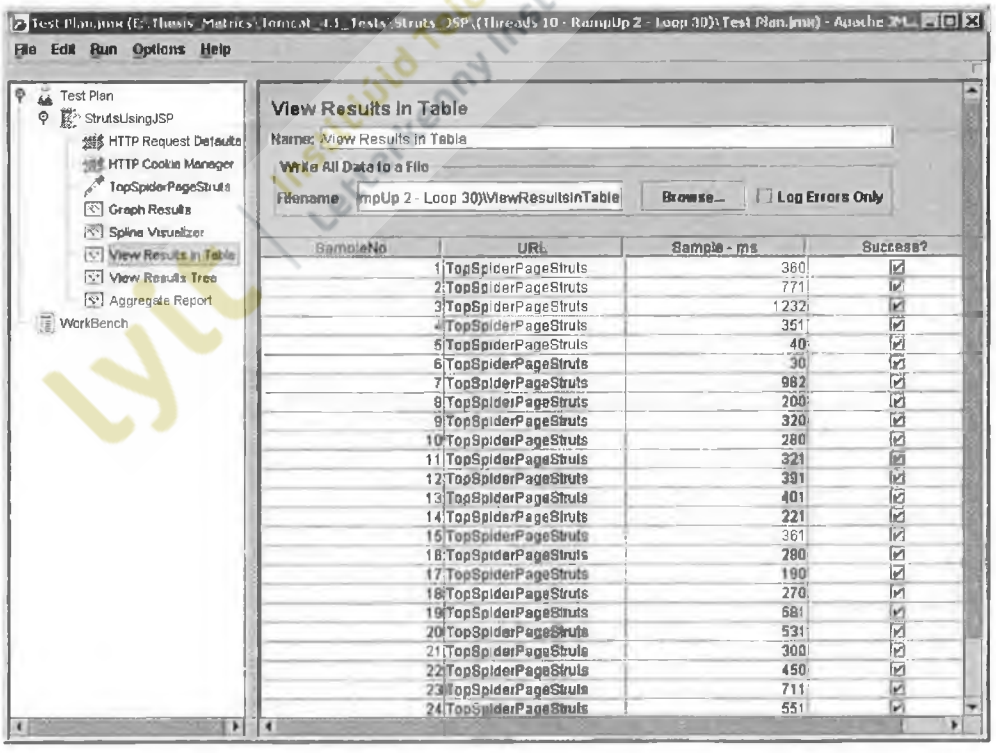


Figure D.18: View results in table of Apache Struts using JSP

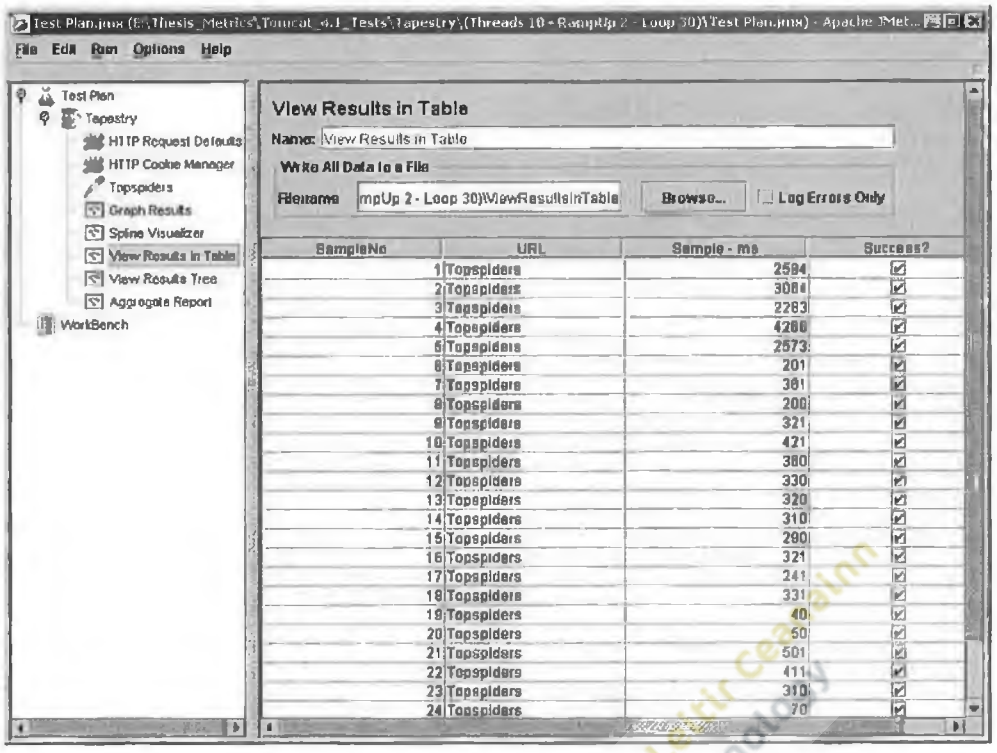


Figure D.19: View results in table of Apache Tapestry

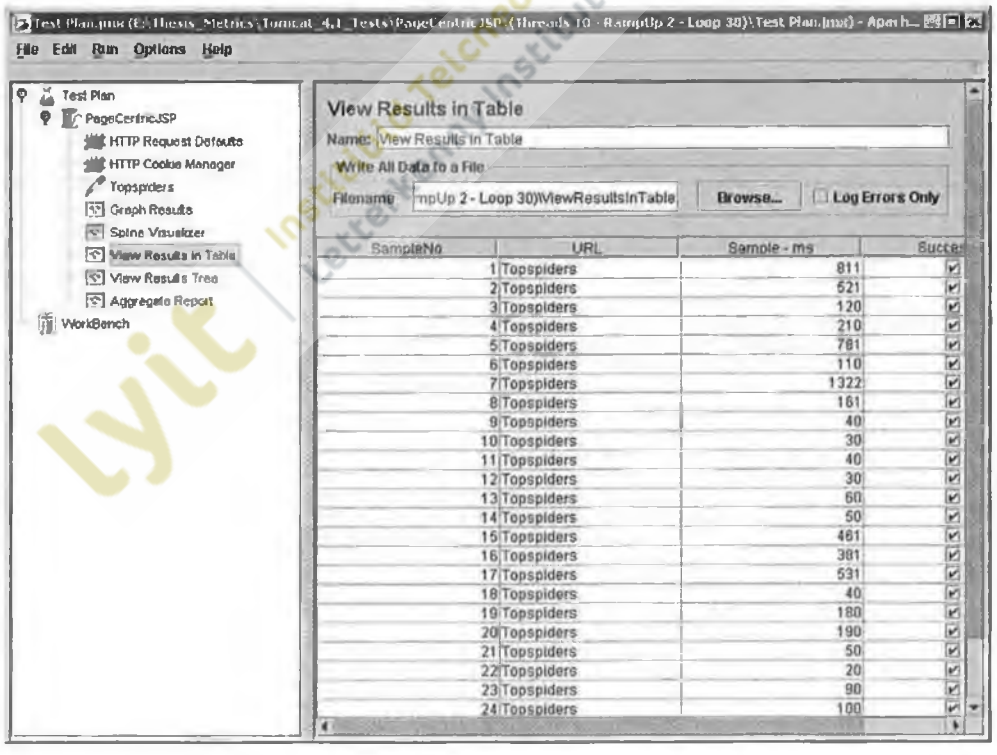


Figure D.20: View results in table of page-centric JSP

View Results In Tree

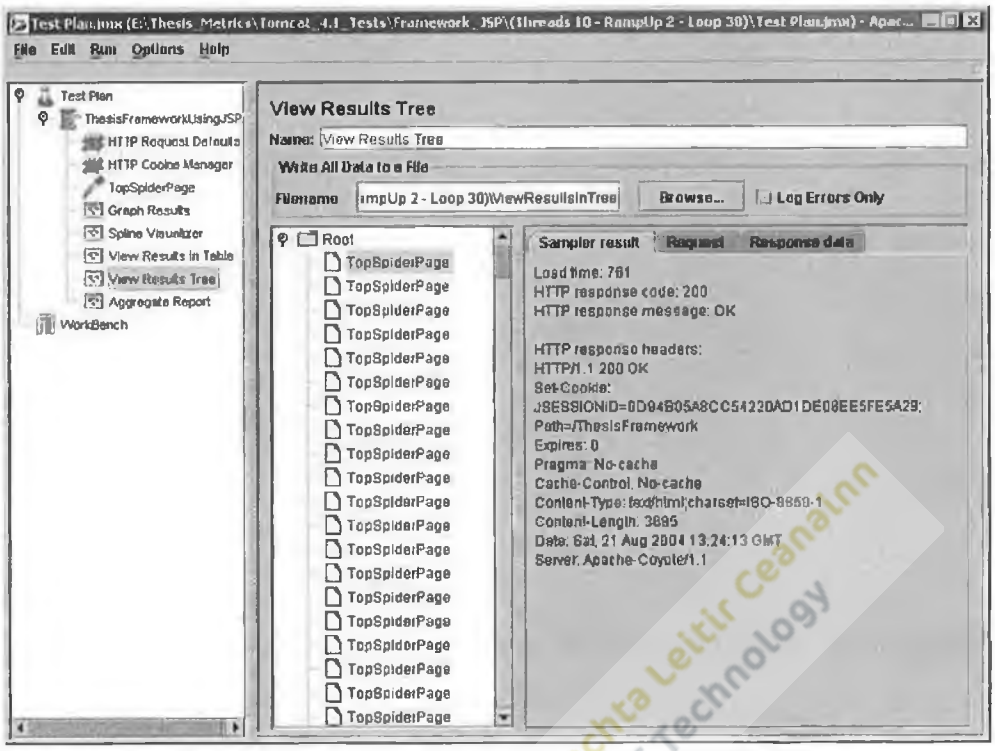


Figure D.21: View results in tree of new framework using JSP

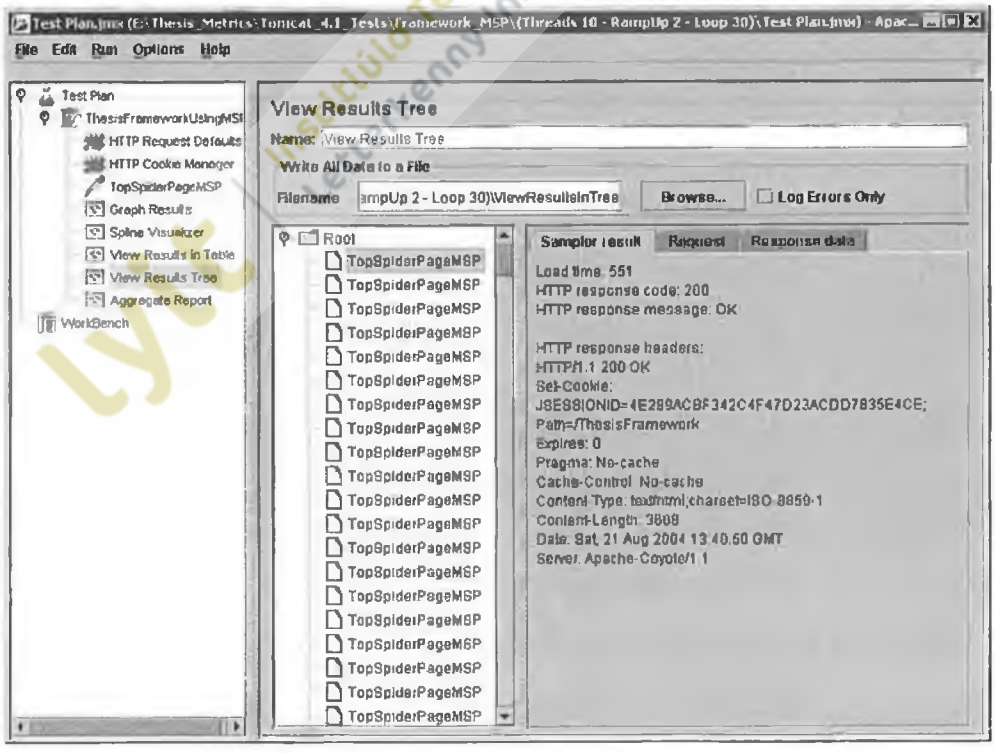


Figure D.22: View results in tree of new framework using MSP

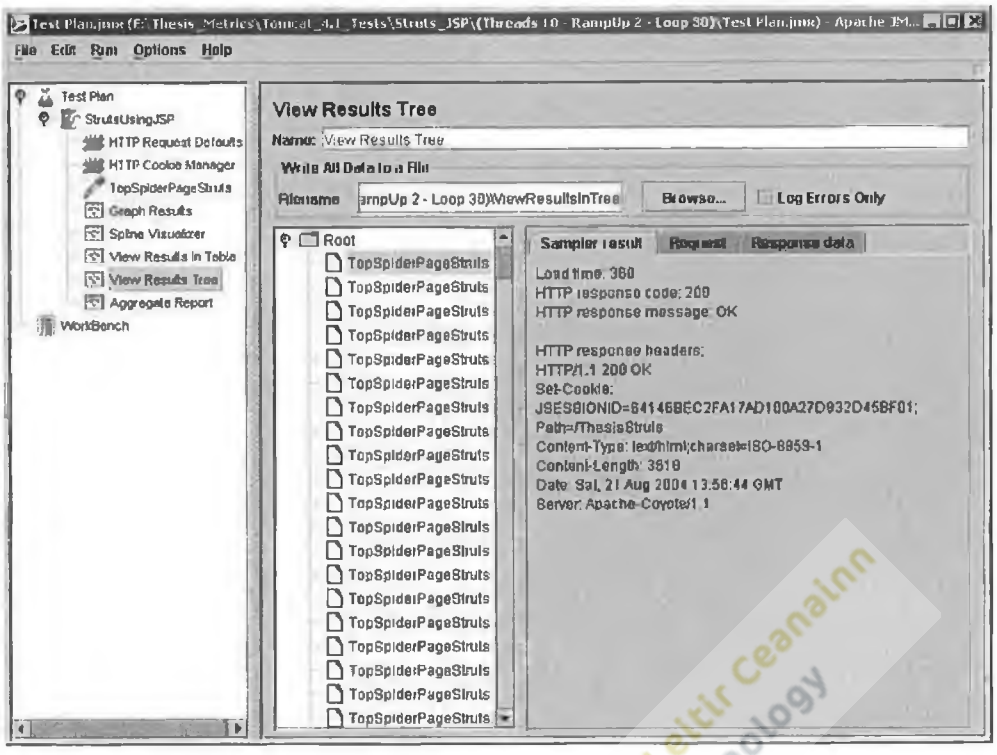


Figure D.23: View results in tree of Apache Struts using JSP

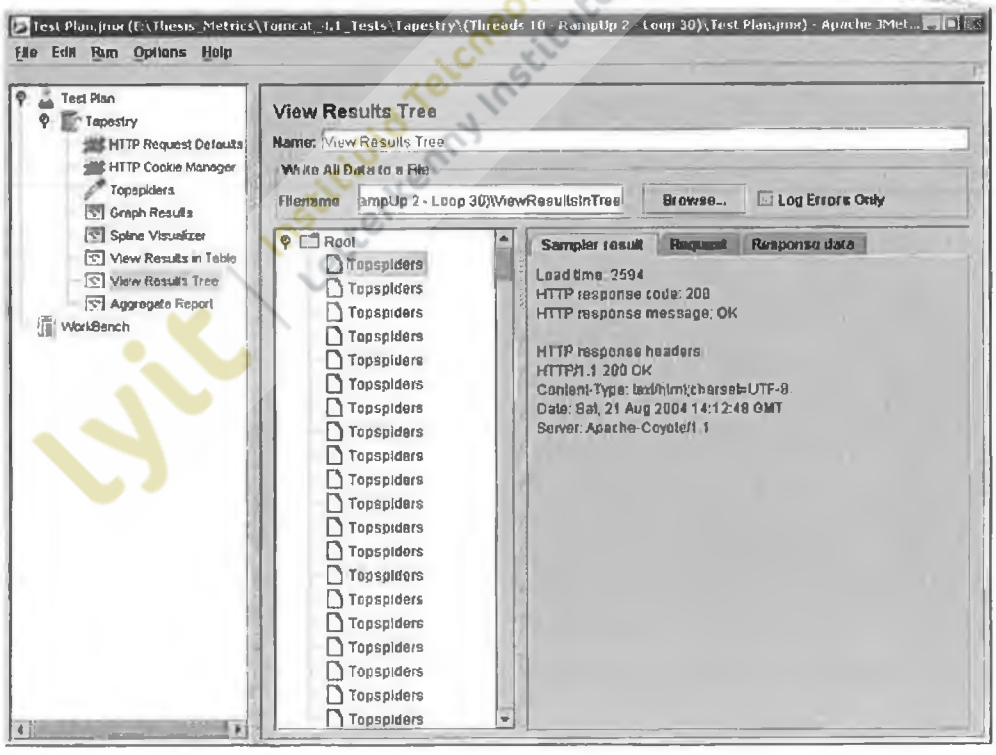


Figure D.24: View results in tree of Apache Tapestry

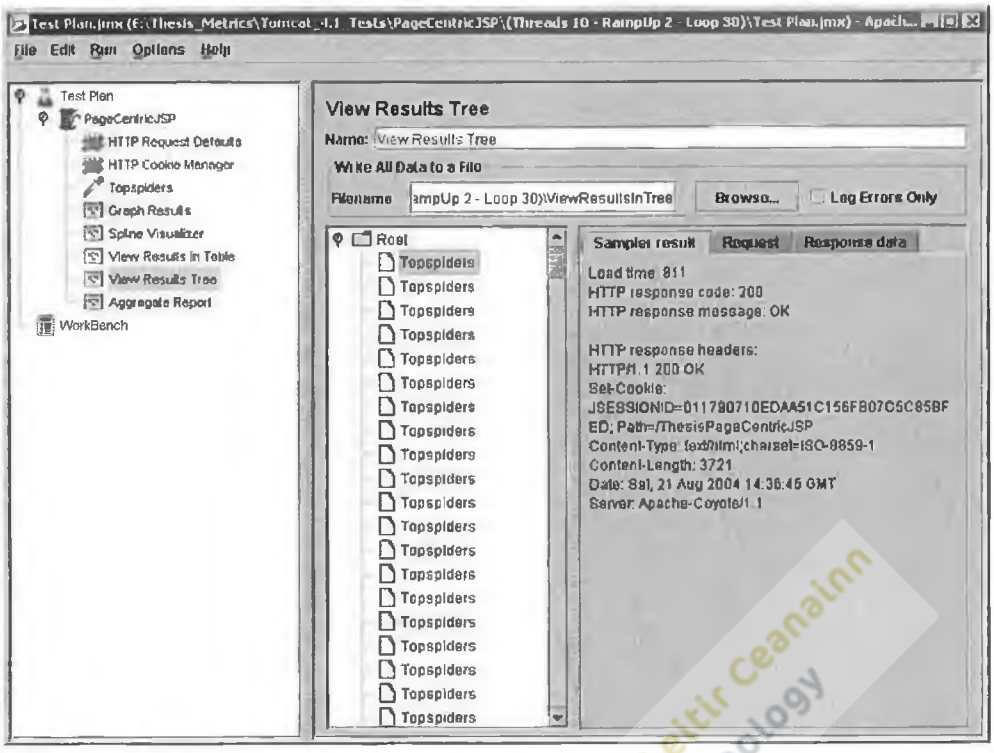


Figure D.25: View results in tree of page-centric JSP

Overall Results

Architecture	minTime	maxTime	Average	Rate	Deviation	Throughput	Median
Framework (JSP)	20	4717	368	20.6	554	1236	261
Framework (MSP)	10	3605	379	20.9	449	1251	280
Struts	20	8062	391	14.6	1064	873	40
Tapestry	30	6409	559	15.9	752	953	441
PageCentric (JSP)	20	10375	377	15.4	1203	923	30

Table D.1: Benchmark two's overall result