A Presentation Service

for

Rapidly Building Interactive Collaborative Web Applications

A thesis submitted to the

School of Computer Science

University College

University of New South Wales

Australian Defence Force Academy

for the degree of Doctor of Philosophy

By

Michael Joseph Sweeney

31 March 2008

Certificate of Originality

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person, nor material which to a substantial extent has been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by colleagues, with whom I have worked at UNSW or elsewhere, during my candidature, is fully acknowledged.

I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.

Michael Joseph Sweeney

Abstract

Web applications have become a large segment of the software development domain but their rapid rise in popularity has far exceeded the support in software engineering. There are many tools and techniques for web application development, but the developer must still learn and use many complex protocols and languages. Products still closely bind data operations, business logic, and the user interface, limiting integration and interoperability.

This thesis introduces an innovative new *presentation service* to help web application developers create better applications faster, and help them build high quality web user interfaces. This service acts as a broker between web browsers and applications, adding value with programming-language independent object-oriented technology.

The second innovation is a generic graphics applet (GGA) component for the web browser user interface. This light component provides interactive graphics support for the creation of business charts, maps, diagrams, and other graphical displays in web applications.

The combination of a presentation service program (BUS) and the GGA is explored in a series of experiments to evaluate their ability to improve web applications. The experiments provide evidence that the BUS and GGA technology enhances web application designs.

Acknowledgements

First, I would like to thank my academic supervisor Dr Graham Freeman for his timely advice, support, and patience. Graham was always supportive, and was understanding of my competing obligations to my family and career. I also have appreciated the positive and helpful attitude of all the staff in the Computer Science school at the Australian Defence Force Academy.

I would also like to thank my wife Kerry and children Joanne, David, and Samantha who have been very supportive and understanding during this period. They have had to do without a husband and father on too many evenings, weekends, and holidays.

My brother Dr Peter Sweeney was tireless in encouraging me to finish the work, and provided lots of practical advice on thesis writing over many glasses of wine.

My colleagues at DSTO have contributed with advice which I found very valuable. In particular I would like to thank Dr David Miron, Dr Paul Whitbread, and Dr Iain MacLeod.

This work has been supported by the Australian Defence Science and Technology Organisation (DSTO). DSTO has generously provided study leave, computing resources, conference trips, an environment to run a case study, programming staff support, and scientific library services.

The GGA, GOL, OMA, and S3DA components were built by professional programmers working directly to me. Each component was programmed in Java by engineering staff or contractors to my original specifications. I was the sole customer, had complete design control, and performed all acceptance testing. The programming

staff performed most of the internal applet design and unit testing. I have included attributions in the thesis where programmers or others have contributed in other capacities.

The OpenMap application is open source software managed by the American BBN company. The 3D engine at the heart of the S3DA applet is a Java product owned by Anfy.com.

A special thanks to Peter Hoek, a talented software engineer I have worked with for many years, who displayed endless patience in implementing my unusual specifications and long lists of enhancements.

This thesis is produced using BibTeX (version 0.99c) and LaTeX 2_ε (release 2001/06/01) and dvips (version 5.92b). Figures were drawn with tgif version 4.1, and converted to EPS files. Python programs are designed for version 2.4.2 of the language.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Since 1995, the internet has grown exponentially in both users and active servers. Once an elite network for academics, the internet is now the foundation for many industries, a vital business resource, the common communications tool, the centre of many social groups, and a homework assistant. The internet age has been enabled by the rapid increase in networking, the ubiquitous web browser application, available web server software, and the HTTP, URL, and HTML standards. The World Wide Web (WWW or simply the *Web*) is the emerging dominant information platform that is based on this internet technology.

As segments of business and social interaction have moved to the Web, the quantity of information to be captured, managed, and presented has increased rapidly. Hand-crafted static web pages authored by a web master are no longer sufficient. Demand has driven the IT industry and open source developers to produce tools that enable non-expert users to produce content, and enable IT staff to build and maintain large web sites.

The World Wide Web (WWW) is being used for increasingly complex application delivery. The types of applications being built for the web include e-commerce, personal communication and coordination, data repository dissemination, and business workflow. Software development projects created to build these new applications have to deal with a new set of tools, components, and limitations.

This rapid growth in tools to enhance productivity, capacity, reliability, and security in web sites has caused many problems for software developers. Many ad hoc tools have been quickly implemented as a temporary solution and then built upon as demand has grown.

Standards for languages, protocols, and API access have been evolving rapidly and developers have freely extended or ignored them. Academic contributions to the field have also been lagging behind, leaving developers without guidance on development methodologies, user interface designs, applicable software patterns, and effective architectural frameworks.

This thesis consists of a study in technology, the innovation of a software design concept, the innovation of a new graphics web component, and experiments on the application of these new ideas. The study covers the techniques, technologies, and tools that underpin the development of web applications. The focus is on the shortcomings of the current development environment, and understanding the fundamental architectural themes that web applications are built upon. The new software design concept is a *presentation service* which is designed to mitigate some of the problems experienced in building and maintaining web applications, and encapsulate some of the complex design that web applications require. The new graphics component offers to bring general purpose interactive graphics to any web application. It is a synthesis of many current designs that support graphics in the browser, wrapped in an easy-to-use interface and adding hooks for flexible integration. Finally, a number of experimental applications have been built and evaluated. The mechanisms of how these new concepts work are discussed with an analysis of experimental performance and opportunities for future research.

This chapter introduces the web application subject and current challenges before outlining the thesis aims and method. The original and significant contributions to the field are summarised, and the last section provides an overview of each chapter in the thesis.

1.1 Introduction to Web Applications

The idea for the WWW can be traced back to a document called "Information Management: A Proposal", authored by Tim Berners-Lee in 1989 [Berners-Lee 89]. The original design called for a network oriented primarily toward academic use with no clear distinction between a reader user and a writer user. Software and protocols grew (mostly due to the contribution of key staff in CERN), and these new concepts were propagated for wider adoption in 1994 [Berners-Lee *et al* 94].

In the early years of the internet, content was almost exclusively static text combined

with image files. Since the internet has become available at work and home to many western world people, an increasing number of commercial, government, and community customer services have been made available using the now well-known internet browser user interface. These services are implemented as web applications. They typically receive information requests from customer browsers and use server software to update agency databases, then return pages of information to be displayed in the browser.

The popularity of web-based applications has grown rapidly over the period from 1995 to 2007. The web now has over 65,000,000 active web sites as of December 2007 [Net07], and most of these sites are supported with web application tools to assist users in managing content and deploying applications. The number of users on the web is also climbing. More than 20% of the World's population are now web users, and in developed countries, usage rates exceed 75% [Min08]. Users are also spending more time using the web; in Australia, users now spend more time using the web than watching television [Sultana 08]. This huge number of web sites and active users drives the need for new and better methods to manage web content and web systems.

Most major software vendors offer web application support as standard in new software, and organisations are using this facility to build complex business-critical systems accessible from their intranet and the internet. The applications are closely integrated with the organisation's web site, and typically provide remote staff, customers, and business partners a level of interaction with the organisation's information systems. With a growing trend to move organisation structure and processes into cyberspace, some organisations are run where the web site *is* the organisation (eg: outsourcing companies, Sourceforge development teams, and internet standards organisations).

1.2 Evaluating Web Applications

There have been many studies on the performance of different web technologies and on how to manage these aspects of performance [Abdelzaher & Bhatti 99], but these works focus on the quantitative analysis of resource consumption and scalability. One of the problems in evaluating and comparing web application technologies is the difficulty of assessing their qualitative properties [Wampler 01]. These qualitative properties are often

of more interest to the prospective developer that the quantitative properties. If the number of web requests served per second on the server is poor, a larger server may need to be purchased; however problems in reliability, interoperability, or flexibility may doom a project that is based on this architecture, technology, or framework.

The performance of software is typically measured by the computing power needed to support the internal algorithms, the memory space consumed, and the network bandwidth used. In high-performance applications, or where the transaction rate is very high or resources constrained, these performance measures may be the dominant factor in the decision of whether to use this software; however web frameworks and services are not chosen because of their minimisation of system resources, but their design features that assist product development, application integration, system administration, and the user's productivity.

In most circumstances (looking at the total cost of ownership), the dominant factors in the software domain that affect the organisation the most are:

- The cost and risk in developing solutions using this software,
- The difficulties in building collaborative software that allows staff or clients to work together,
- The lack of flexibility and adaptability in resulting solutions,
- The frequency and severity of system unavailability,
- The ability to integrate with other current and future systems,
- The time required for developers to become proficient with this software,
- The usability (user learnability and efficiency) of user interfaces developed using this software,
- The ongoing cost of being trapped into a particular operating system, programming language, network service protocol, or database,
- The ease in reuse of existing internal and external working components, and
- The cost of infrastructure to support the software under typical and stressful user loads.

These factors are, in part, affected by the properties of the supporting software used to build systems. The other factors such as staff quality, management technique, executive and user participation, and appropriate development methodology are outside the scope of this study.

Web applications are more highly dependent on support software than other software due to the complex and ever changing technology environment, the demand for good web programmers, and short timeframes imposed on development teams. The software supporting web application developers can be divided into developer support, web content creators, and run-time support. Developer support tools are intended to help the developer design and build software (not necessarily web applications) and include modelling software and Integrated Development Environments(IDEs). Web content creators, interactively or via batch mode, build and maintain web sites (eg: Microsoft Frontpage). The last category, and the one I will be examining in this thesis, is run-time support software.

Run-time software support for web applications include frameworks, services, components, and code libraries. This software is used as a foundation or parts of a programmed web application. Examples include the Java environment, the .NET framework, the PHP module, and the YUI Javascript libraries.

The properties of web application run-time support software is shown in Figure 1.1. A link between a stakeholder and a property indicates that the stakeholder benefits from an enhancement to this property. The use of a framework, architecture, component, or technology in the building of a web application will have an effect on one or more properties, and therefore an effect on the efficiency or effectiveness of the work of the stakeholder group. These properties are sometimes called *Quality Attributes*. The set of quality attributes used in this thesis are a combination of those found in software engineering [Barbacci *et al* 95] and those used in software architectures [Clements *et al* 02].

Around the properties are the *Supported Activities*. These activities are not directly linked to a set of properties but located in an approximate position that indicates the type of activity that is dominant in the spectrum of web application concerns.

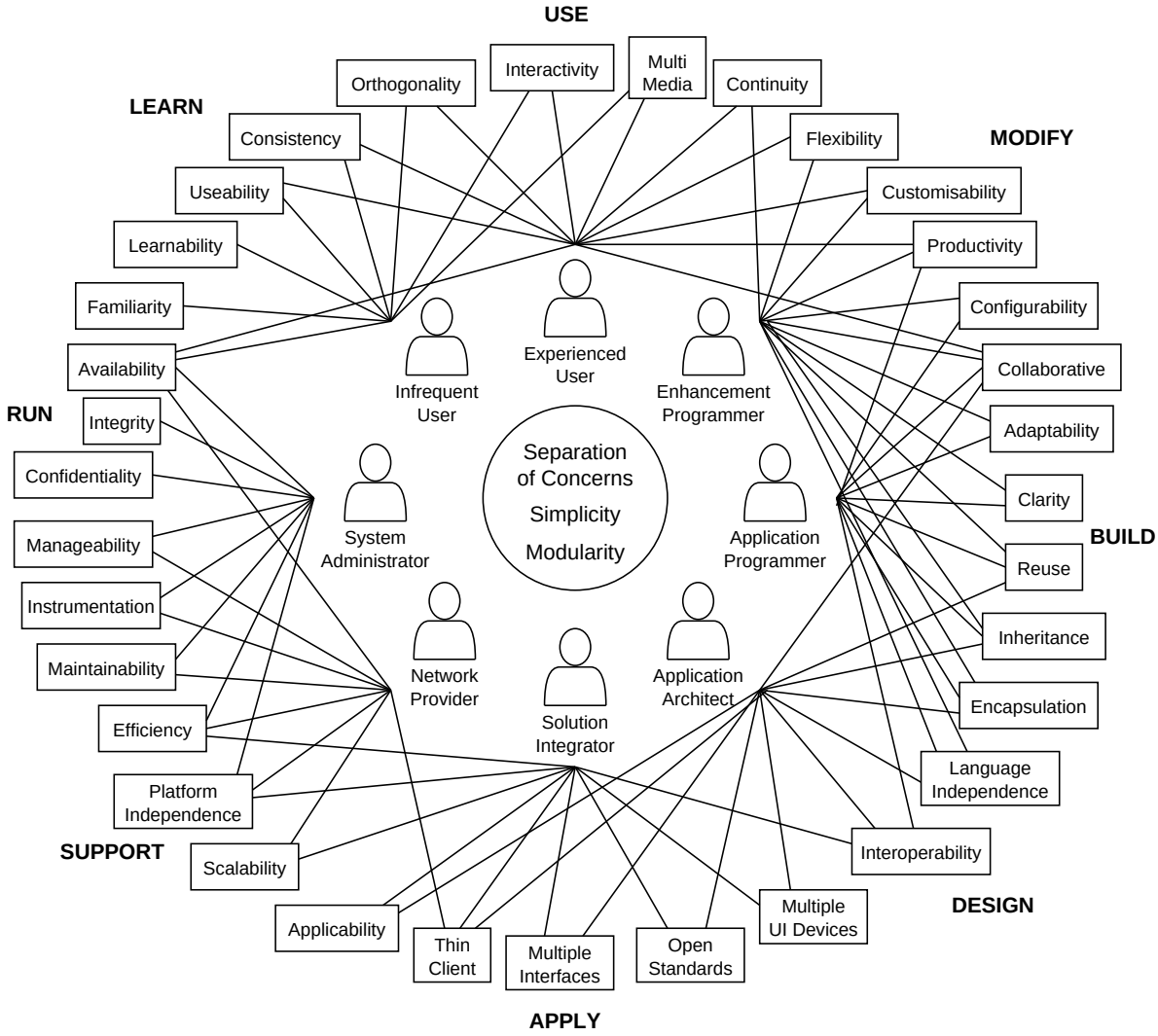

Figure 1.1: Web Application properties and benefiting stakeholders.

The core properties which positively influence all stakeholders are the *separation of concerns*, *modularity*, and *simplicity*. These properties are important in any system, and systems operating in complex environments (such as web applications) *require* these properties to manage the tangle of interacting components, interfaces, and behaviours.

Table 1.1: Properties enhanced by Web Application support software (part 1).

Property	**Enhancement**
Sep. of Concerns	System division of work responsibilities with minimum coupling.
Simplicity	Intuitive, clear, logical, and minimal mechanisms.
Modularity	Division of the system into interacting logical components.

In Tables 1.1, 1.2, and 1.3, the properties are listed with a short explanation of the role the property plays in web application development. Each property may enhance more than

Table 1.2: Properties enhanced by Web Application support software (part 2).

Property	**Enhancement**
Familiarity	Appearance and behaviour similar to established methods.
Learnability	Intuitive functionality usable with minimal training.
Useability	Support of efficient work practise with speed and control.
Consistency	Uniformity of appearance and internal behaviour.
Orthogonality	Independence of function, structure, presentation from each other.
Interactivity	Support for user control of the UI with rapid feedback and updates.
Multi-media	Support for integrated rich text, images, graphics, and media file presentation.
Continuity	Recovery from infrastructure faults and software enhancement with minimum user disruption.
Flexibility	Breadth of functions, structures, and interfaces in the design.
Customisability	Support for users to modify system operation for optimising work.
Productivity	User or developer work efficiency.
Configurability	System versatility by editing configuration without code change.
Collaborative	Facilities to connect user sessions to provide a common workspace.
Adaptability	Able to be easily adapted by programmers for new purposes.
Clarity	Ease of understanding of the design, languages, and protocols.
Reuse	Use of existing resources in new roles.
Inheritance	Building new components by referencing and extending existing work.
Encapsulation	Support for each aspect of the system in components with clear APIs.
Lang. Independence	Freedom of computer language in software enhancements and applications.
Interoperability	Open, flexible, and simple methods to connect with other systems.

one aspect of web application design and use.

Note that these properties are *intrinsic* attributes of the technology or technique. Other relevant properties are a product of the *application* such as workflow, data persistence, value validation, and business rules. Another set of properties that would inform a decision to adopt or purchase a new web application framework, architecture, components, or tools would be the *business* properties. These properties (such as technical support, industry acceptance, extent of documentation, compatible products, and programmer availability)

Table 1.3: Properties enhanced by Web Application support software (part 3).

Property	Enhancement
Multiple UI Devices	Support for multiple UI technologies including desktop, web, email, and mobile.
Open Standards	Use or provision of open published standards for languages, protocols, and components.
Multiple APIs	Facilities for integration at multiple layers in the system.
Thin Client	Minimisation of client platform configuration change to support system.
Applicability	Extent of application roles this software can be used in.
Scalability	Ability to handle large datasets, transaction rate, and user count.
Plat. Independence	Minimisation of requirements for facilities supplied by a subset of available computing platforms.
Efficiency	Conservation of computing power, memory, disk space, and network bandwidth.
Maintainability	Ease of fault isolation and repair without side-effects.
Instrumentation	Monitoring and presentation of system parameters.
Manageability	Facilities to control running systems.
Confidentiality	Protection of information from unauthorised access.
Integrity	Protection of information or system from damage or loss.
Availability	Ability to serve users with minimum interruptions and failures.

must be considered for business reasons, but are not intrinsic properties of the technology.

Effective evaluation of web application environments would include the three core properties and capabilities in some of the other properties. The ideal web application environment would form the foundation of web applications that excel in all of the thirty seven properties shown in Figure 1.1; however, as many of these properties require trade-offs in the real world (such as flexibility vs. efficiency), there can be no perfect environment. My goal in this thesis is to understand the available technology and current research, then develop new concepts which offer enhancements in several of these properties.

1.3 Challenges in Developing Web Applications

One of the main reasons that web application development methodology and support has lagged behind mainstream software engineering is the history of the web as a document publication and dissemination medium.

Previous researchers [Gellersen & Gaedke 99] have described this aspect of the problem:

> *One reason for the lack of a structured approach may be in the Web's legacy as an information medium rather than an application platform. Web-development is seen primarily as an authoring problem rather than a software development problem to which well-established software engineering principles should apply.*

Another problem derives from the complexity in the web environment. The more sophisticated web applications use a mix of Javascript, HTML, style sheets, Java applets, server side programs, databases, email, security, and web server customisation to achieve the required functionality. This combination of rapidly evolving protocols and technology has uncovered a number of development and management problems that are not addressed by traditional software engineering techniques.

The lack of formal development methodology, application layer models, or design patterns applicable to the web application development domain have caused web application developers to adopt ad hoc methods [Gellersen & Gaedke 99]. The vast menagerie of support software have overlapping concerns, different approaches, incompatible interfaces, and immature documentation. Web developers use a set of familiar tools combined with home-grown tools and custom code to assemble and grow web applications. These ad hoc methods lead to inefficiencies, brittle systems, and unreliable operation.

Standard web applications typically use a large number of programs, components, services, and configuration files located within the web server. This custom software uses a tangle of code developed over years to receive user events, apply application logic, and process database transactions. Differing web application architectures and components have trade-offs in scalability, flexibility, performance, interoperability, and vendor lock-in, but the combination of multiple *types* of architecture and components with custom glue code to hold it together and adapt it to the user requirements is a source of failure for many web applications.

In an analysis of the design of web applications [Brown 02], Brown stated in his thesis:

> *A successful Web application design must bring together three major elements: structure, navigation, and presentation. To bring these elements together in a Web application requires a clear and systematic design approach that overcomes design problems related to separation of concerns, design frameworks and patterns, and framework representation.*

In this thesis, Brown went on to point out that intermixing of structure, navigation, and presentation makes it difficult to organise, reuse, and extend the application. There is a need for a comprehensive approach to web application development that is built on object orientation and accepted design patterns.

A standard web component architecture that is independent of language and platform, and uses strong architectural features such as object orientation, middleware, and multi-tier concepts, is yet to be defined. There is currently no language-agnostic framework for re-use and integration of DHTML [Niederst 99] [Goodman 98] content and structure, event propagation (typically get and post requests), and presentation logic.

1.4 Research Aims

This thesis is centred on the user interface aspects of run-time support software for web applications. The aims are to understand the architectural and technological concepts behind the problems of building web application user interfaces, and create innovative architectural and technological solutions that improve on the current state of the art.

The design of SQL database systems was influenced by the need to encapsulate the specialised and shared data handling requirement into a simple data service. The resulting design hides the complexity of indexing, query optimisation, relational joins, disk space allocation, multi-user access, error recovery, and caching behind a simple structured English protocol exchanged over a network connection.

This thesis investigates how the concepts of the data service can be adapted to hide the complexities of web application presentation mechanisms behind a simple protocol over a network connection. The aim is to combine common browser display functions into a

long-running process that receives connections from applications that supply services, and manage user events and UI generation with multiple user interfaces.

To understand the environment and create a presentation service, this study will be guided by the following questions:

- What functions are needed within the service?
- How will multiple UI client protocols be handled?
- How will the service route user events to applications?
- How will presentation rules and data be handled within the service?
- What protocol will applications use to communicate with the service?
- What are the benefits of using the presentation service?
- What new capabilities become possible when using a presentation service?
- What are the limitations of the presentation service?

During the study, it became obvious that server-side innovation would be unable to address a number of common web application requirements. The lack of interactive graphics support, poor integration of graphics with surrounding web components, and page-oriented transactions limiting interactivity and information currency, suggested the development of a general purpose client-side graphics component with strong integration and server-communications.

The study was expanded to also investigate the design of this client-side component. This component extends the capabilities of the presentation service by providing graphics capabilities, client side integration, and asynchronous events and commands between applications and user interfaces.

This study will also seek to address these questions:

- What graphical functions are needed in the UI component?
- How would the component integrate within the browser environment?
- What is the protocol between the component and presentation server?
- What are the benefits in using this over current solutions?
- What are the limitations of the graphics component?

Overall, this thesis aims to answer the question:

Can a presentation service combined with the graphics component provide a simple and powerful framework for implementing interactive and collaborative web application user interfaces?

1.5 Approach

I have approached these problems with a combination of analysis, exploration, and experimentation. My first step was the research of the protocols and languages that make up the web application environment. I then analysed common web frameworks, services, components, and libraries, examining how they operated, and their features and performance. Further study included a survey of current research in web run-time support software. This introduction includes a definition of the properties I will evaluate to assess web application technology.

To better understand the work of the web application designer, I designed and built a number of small web applications and a small web application framework (called WebFrame). This experience provided an insight into the real issues programmers encounter when building web applications for the real world. The complexity of all the protocols and languages, combined with constantly evolving software tools and the lack of software engineering support demonstrated to me that there was significant opportunity to create innovative solutions that would have immediate impact.

My first innovation was the development of the presentation service concept. This service would encapsulate some of the complexity of the UI code and provide a simple API for web applications to use. I designed and built a browser-oriented version of the presentation service called the BUS. The BUS was used in experiments to test a number of internal logic designs, object structures, interfacing protocols, and communication languages. After many trials, I achieved a flexible service that displayed many of the target properties (see Figure 1.1).

A major limitation in the browser user interface was the lack of graphics support. After further research in available technology and methods currently in use, I commenced another development effort. The aim was a general purpose graphics component that would

integrate with the surrounding web page and act as another user interface device for the BUS. The developed prototype was used to implement a number of interactive graphics functions, and was found to not only meet the requirement, but offered many new web application support functions.

To evaluate the BUS and UI component technology, I devised a number of experiments to gather information on development effort and UI enhancements using the new technology. The properties of the resulting web applications and user interfaces were evaluated to determine which properties were improved by the use of this new technology. An analysis of the enhanced properties provides evidence of the benefits of using the presentation service and UI components in web application architectures.

1.6 Scope

The web application domain is a large area for research. My research is focused on the techniques and technologies in building dynamic interactive web user interfaces. The research will include the protocols and markup languages that web browsers use to interact with web applications, and the current architectures and techniques used to generate displays in user browsers and capture user events.

There are many other problem areas that contribute to make web application development so challenging. These topics are worthy of study but are excluded from my research:

- Web application analysis and design methodologies.
- Development of models and resource generation.
- Site navigation schemes.
- Web page content layout and ergonomics.
- Interactive web resource editing tools.
- Programmer support tools (eg: IDE, RCS).
- Business modelling and code conversion and generation.
- Security (authentication, script/SQL injection, HTTPS).

- Engineering for reliability (eg: Error detection and recovery, load balancing, failover, logging, profiling).
- Usage of proprietary products and protocols.
- Highly interactive UI applications (eg: paint, 3D games, video editing).
- Javascript libraries (Javascript is useful to enhance a UI; however, Javascript is an *enhancement* rather than a *part of* the presentation service).
- Media without embedded hyperlinks (eg: audio, video, file transfers). Binary media is served by the presentation service — not *part of* the presentation service.
- Web service protocols (SOAP, WSDL, UDDI).
- Web service architecture (where used in application and data layers).
- Project management, deployment, and maintenance.

In this thesis, I will use many common terms from computer science without definition, but new terms or terms used in a different context will be explained in the text, the glossary, or the abbreviation list. In particular, I will assume the reader is familiar with the basics of the following topics:

- Internet network protocols (eg: TCP/IP, HTTP).
- Web markup languages (eg: HTML, XHTML, XML).
- Network programming (eg: Sockets, blocking, forking, threading).
- Web navigation (eg: Hyperlinks, URLs, forms, buttons, icons, frames).
- Basic software engineering (eg: Objects, modules, APIs, distributed systems).
- Database server architecture (eg: SQL, schema, client-server, data operations, data storage).

1.7 Innovation

The value in this research work can be divided into three parts: the research and development of a new *presentation service*, the research and development of a new *general purpose*

graphics component, and the *experimental evaluation* of a new web architecture based on these two innovations.

The presentation service is a unique contribution to computer science. There are some elements in common with database management systems and the X Windows system, but the central concepts are my original work. Some early client-server research (such as [Cassell 94]) explored the design of a *presentation server*, but this new work explores a network component that offers services to both user interface components and dynamically connecting applications, providing a *separation of concerns* between the users' UI tools and the application development issues.

The Browser-based User-interface Service is a prototype based on the presentation service concept. The BUS is a long-running service that supports multiple user interface devices binding dynamically to a user interface adaptor, and application components binding dynamically to an application handler. The BUS encapsulates user interface protocols, logic, and functions, simplifying the application design and improving the functionality of web applications.

The GGA graphics component is also a unique contribution to computer science. There is some superficial similarity between the GGA and applets such as the ROSA applet [DM 02] and the CSIRO SVG Viewer [Robinson & Jackson 99]; however the GGA applet offers asynchronous communications with a server using English-like commands, a rich variety of event reporting (including drag and drop), and layer and group management. This flexibility and ease of integration makes the GGA a very useful component that will enhance the properties of many web application designs.

The employment of the BUS and the GGA require a rethink in the design of web application architectures. To evaluate the enhancements to resulting web applications, I devised and conducted six development experiments. Three of the experiments involved the design and testing of new UI components (the GOL for GIS integration, the OMA for GIS encapsulation, and the S3DA which encapsulates a 3D world engine).

The properties of the developed applications and user interfaces were evaluated and found to be significantly enhanced by the application of this new technology. The simplicity of the resulting applications and the high functionality of the user interfaces provides evi-

dence to support my claim that the presentation service concept and generic-graphics user interface components provide an enhanced architecture for the development of interactive collaborative web applications. This unique technology has the potential to increase web developer productivity and enhance the user experience in the huge web application software industry.

1.8 Thesis Structure

This chapter has introduced the web application, and the problems developers have in building them. The aims of the thesis were then stated. The methods used to investigate the thesis problem and develop innovative solutions were discussed, and the scope of the study defined. Finally, the original and significant contributions of this thesis were presented.

In Chapter 2, I will describe in more detail the issues that affect web application developer productivity and the quality of resulting applications. A review of major web application architectures, technologies and tools is then presented, illustrating the diversity and complexity of the web application environment. I will then describe some of the key software engineering techniques that are often applied to the web application domain, and summarise a number of web application research projects. At the end of this chapter, I include a discussion of three related web technologies: SOAP services, rendering with XSLT, and the new Web 2.0 technologies.

The concepts and technologies underpinning the new *Presentation Service* design are explored in Chapter 3. In this chapter, I describe how the service encapsulates user interface functionality, and how the service is designed to interact with user interface devices and application components. The key design concepts of the XML API, inversion of control, management of presentation objects, dynamic client binding, and separate data context are explained. This chapter is closed with a statement of the properties of the presentation service concept, and how these properties assist the web application developer.

In Chapter 4, I describe the lack of support for interactive graphics in current web technology, and introduce a novel general-purpose graphics component called GGA. I describe the key GGA design parameters before explaining how the component operates. I have

supplied example displays and examples of interface communications to demonstrate the utility of the component and its simple yet powerful interfaces. The enhancing properties of the GGA are presented with links to the web application developer problems they address. I have included a review of related research and opportunities to extend on this work at the end of the chapter.

Chapter 5 builds on the presentation service concepts described in Chapter 3. In this chapter, I introduce and explain the design and operation of a prototype presentation service called the Browser-based User-interface Service (BUS). I describe the internal structure, transaction patterns, and the XML language developed to communicate with web application components. The BUS concept is then evaluated by analysing the benefits and limitations in the web application developer context.

In Chapter 6, I present a number of experiments using the BUS and UI components. I explore the experience of developing web applications using this new technology, and evaluate the small web application code and user interface capabilities to determine the advantages and limitations of the technology. The significant findings are presented at the end of the chapter.

In the conclusion in Chapter 7, I summarise the work in the thesis and review the original content. In this final chapter, I also state the significant contributions and potential for impact in the web application architecture domain. Lastly, I propose several possibilities for future work that have emerged from my research.

A glossary of specialist terms has been included in Appendix A, and a helpful list of abbreviations can be found in Appendix B. Appendix C describes the syntax rules and extra information about the DMT template language. The Active Expression syntax introduced in Chapter 5 is described in Appendix D. A bibliography is listed at the end of this document, containing entries for all important resources referenced in this thesis.

Chapter 2

Background

2.1 Introduction

There are a number of diverse web application architectures in use. The standard method triggers the browser to request a static resource from the webserver which delivers it from the server filesystem. Basic dynamic interaction is achieved through the return of pages filled from an attached database. Some user interface functions (such as an outline browser) or application logic (such a field validation) can be performed within the browser by the addition of Javascript which operates inside the browser. For integration with multi-page application logic and datastore access, the webserver may be integrated with servlet or CGI programs [Tittel *et al* 95] that process the submission of user-completed forms. A processor may also be attached to the web server which preparses special HTML files that contain application logic embedded in extension tags. In the same way, complete programs may be closely bound to the web server with APIs such as NSAPI, ISAPI, mod-perl, or pyApache. Another option that exhibits good performance but poor interoperability is the custom web server with integrated single-language support. Another approach is the large single applet based application that uses the web server for deployment but otherwise uses very little web functionality. The applet often is the only object on the page and communicates with services on the web server host with JDBC, CORBA, RMI, or URLs.

Web application developers have a large number of software packages available to increase their productivity and quality of developed applications. These packages assist the developer in different ways. Packages may be page editors, template based page generators, ex-

tension modules, frameworks, code libraries, and web server application platforms. There are categories of web applications that employ these packages in different ways. These web application types are discussed in Section 2.2.2. This chapter analyses these web technology platforms and investigates research in the complex domain of web application architecture.

I will discuss the relevant characteristics of a selection of the most popular development methods first. Next, I will discuss the advantages and limitations of a number of web application development environments. Finally, I present a survey of web engineering research prototypes and analyse how these prototypes address the problems I have identified.

2.2 Inside Web Applications

Web applications are differentiated from other applications by the use of a remote web browser as a user interface. Additionally, the web application needs to act as a web server, or work through a web server. The web application will typically run on the same host as the web server because web servers are not designed to initiate other network services to convert protocols. Instead, they support the web application via a custom extension API, fork a separate process which will then executes the web application, or return Javascript or browser components to the client that implement the application functionality.

2.2.1 Web Applications Operations

To interact with the web application, the user clicks on a hyperlink or button in the client browser. This action triggers a potentially complex sequence of software events, then the client browser will update the user's window with web application supplied media. The URL (Uniform Resource Locator) coded into the hyperlink or web form contains the required information to identify the target web server, web application, and user data parameters.

The components of the URL are shown in Figure 2.1. For web applications, the HTTP protocol [Fielding *et al* 98] is usually used, though a secure HTTPS version can also be used to encrypt client-server communications to protect confidentiality and data integrity. The web server hostname is used to uniquely identify the target web server in the network

where the web application can be accessed.

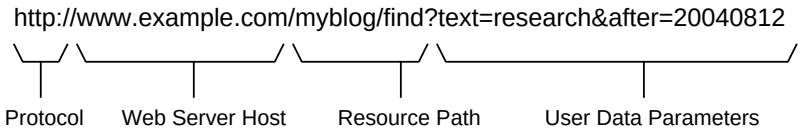

Figure 2.1: Components of the URL.

First, the client host establishes a TCP/IP network link with the server host (see Figure 2.2). The Transmission Control Protocol (TCP) offers reliable point to point communication over network sockets, but until the development of HTTP 1.1, the connection was only used for one transaction then discarded. Subsequent user clicks require the connection to be re-established. This limitation is addressed in HTTP 1.1 where the TCP/IP socket can be maintained over several HTTP transactions, avoiding the overhead of connection and initiation of new sockets with each transaction.

Once the connection is made with the web server, the client transmits a HTTP (Hyper-Text Transfer Protocol)[1] request to the server containing the resource path and user data parameters. The most common request types are `GET` to fetch new content without requesting any changes, and `POST` for submitting requests to take some internal action on the submitted data (such as buying an item or subscribing to a journal). Other requests such as `HEAD` and `PUT` are also used but are less common. After the client sends the request, it waits for a server response.

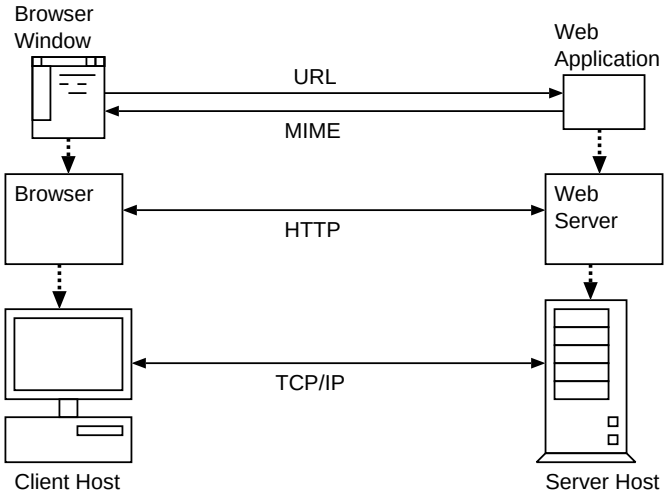

Figure 2.2: Web Protocol Layers.

[1]The HTTPS protocol has a more complex sequence of transactions between client and server, but as this is handled by the browser and web server, and is largely invisible to the web application, we will use the simpler HTTP protocol here.

When the web server receives an HTTP request, it examines the resource path to determine which action to take. This action is described in the web server configuration files which are maintained by the web server administrator. If the resource path indicates a static resource file, a separate web server thread or process is assigned the task of sending the file to the client given the path of the file and the socket to the client. This leaves the web server free to rapidly respond to other requests while other threads feed content to the clients at network speeds.

If the resource path indicates that a web application is to handle the request, the web server calls the web application with the path, header details, and user data. The web application processes the user request then sends a response to the client containing MIME (Multipurpose Internet Mail Extension)[2] encoded data. As web applications use the same URL request, MIME output format, and transaction sequence, client software and intermediate services (such as proxy servers and firewalls) do not require changes to handle web applications — the transactions appear to be identical to web requests for static content.

The content that is returned to the client can be of many types. The web page in HTML format with embedded links to GIF, PNG, and JPG images is the most common content used as static resources or returned by web applications. Other static resources (but rarely dynamically synthesised by web applications) are plain text, document files (PDF), flash animation (SWF), file archives (ZIP and TAR), Microsoft office files (DOC, XSL, and PPT), movie (MOV, MPG, and AVI), and sound (WAV and MP3) content.

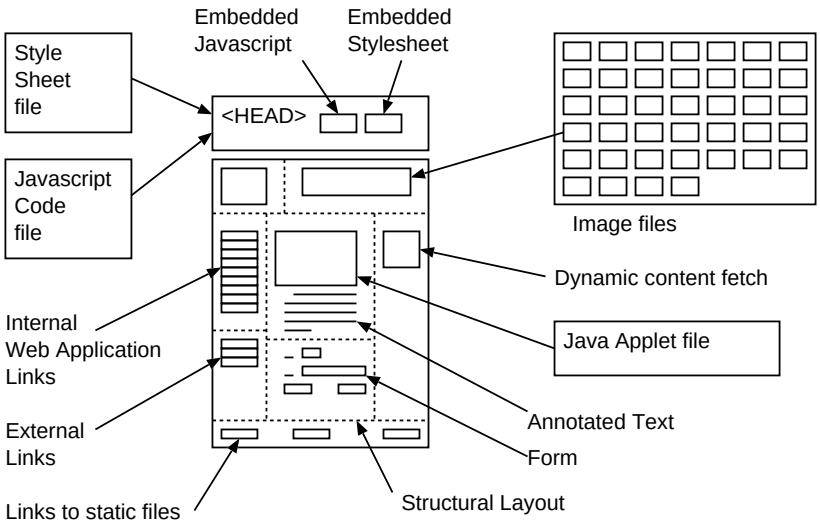

Figure 2.3: Web Page Components.

[2]The MIME format was initially developed to encode multiple media segments in email messages and was then adopted in the web environment for the same purpose.

The dynamically generated HTML (HyperText Markup Language) page is the centre of web application development. The HTML page communicates information to the user, contains mechanisms to capture user information, and provides links to other information in the user context. Information communication is supported by structural tags, embedded graphics, and text annotation. User information can be captured using a form or selection of a hyperlink in a list of options. Other links allow the user to navigate to another part of the web application, change interaction focus to another web application, or request static content. The creative combination (see Figure 2.3) of these elements implements the user interface [Musciano & Kennedy 02].

2.2.2 Web Application Models

Web based applications are currently built using a number of different models. These models illustrated in Figure 2.4 show the distribution of custom code (shaded parts) that implements the web application inside the architecture.

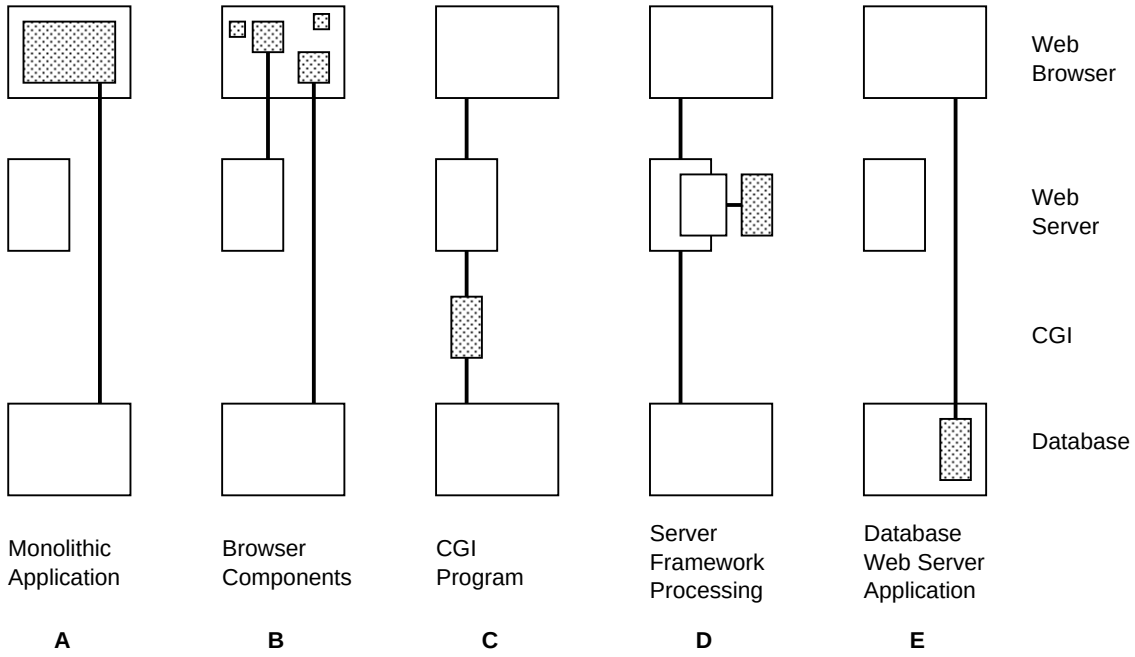

Figure 2.4: Web application models.

The A type uses a large client component such as a Java, ActiveX, or plug-in to embed functionality and connect to remote data stores for client-server applications. The network is typically used only for loading from the centralised software archive and for

database transaction transport. This application type is common on intranets of large organisations. The user interface, application logic, and data services are combined into the same monolithic program.

The B type is often used for lightweight applications such as calculating the cost of products and services from an interactive form. The application flow, rules, and computation code is embedded in the page markup language. Once the form has been edited for values acceptable by the user and the embedded logic, the form may be submitted to the web server which will pass the data to more sophisticated applications for further processing. Complex AJAX applications also follow this model, using code in the web page to dynamically load content and other code on demand. The user interface, application logic, and data are embedded in the web pages.

Probably the most common type on the open internet is type C . A Common Gateway Interface (CGI) program [Ramu & Gemuend 00] is a program spawned by a web server for each URL request, takes a URL argument string and encoded form input on the standard input, executes application logic, and outputs MIME content (typically HTML) on standard output. The CGI programs can be written in any language but the most common are Perl [Wall & Schwartz 92], C, and Python [Lutz 96]. One of the most common functions of CGI programs is to take a set of variables generated by a user with a form, store the data in an RDBMS, query the database for data to populate a new page with, and generate that page. The user interface is mostly generated with strings in procedural code with application logic, and data queries are combined in the same code to perform updates and fill dynamic parts in pages. To help with managing the separation of the UI from code, there are many template HTML generators for each language.

Another common type is D. Typical implementations of the D type are PHP, SSI, PyApache, Perlmod, NSAPI programs, and Java Servlets. These applications are similar in construction to CGI programs, but the server executes the code internally which gives an increase in performance at the expense of server stability due to errors in application code. Another species of this type is the application environment or engine that acts as a web server. These style of application is often a complex cooperating set of components and infrastructure operating in a single-technology web application environment. These environments are intended for large scale developments and can be deployed with several

different web servers, or their own internal server. Development suites such as Websphere and .NET are examples of this model.

For data intensive applications, the E type is used. An RDBMS product will supply a web-server front end to the database. Markup code is stored in tables, and application logic is implemented in stored procedures or custom 4GL programs. These environments are optimised for large-scale complex data displays and large-scale data entry operations. Web page and data caching combine with low overheads due to presentation, business logic, and database running in the same machine (sometimes in the same process), to exhibit excellent performance. The limitations of this model are reduced interoperability and flexibility, and difficulties in using the most appropriate components and development environment in the system design.

Small web applications may use only one of these models, but large applications use a number of these models in complex inter-related designs. The larger frameworks and web application environments include software to support most of these models, although there is a large variety in the extent of support, the functions implemented, and the interfaces available.

2.2.3 Examples of Web Applications

I have selected two moderately large web applications to analyse. I have chosen open source projects so I am able to access the source code. Both TikiWiki and Plone have large numbers of deployed systems and are still in active development. I have measured the amount of code and types of files to estimate the amount of development and complexity of the web application software. These two examples illustrate the size of web applications.

The TikiWiki Web Application

The TikiWiki software is a Content Management System (CMS) developed using an open source model over several years. It is based on the PHP language and MySQL database. The release chosen for analysis is version 1.9. The line counts in the following section includes comment lines and blank lines.

The application consists of 1384 PHP code files containing 326000 lines averaging 235 lines

Table 2.1: File statistics for the TikiWiki PHP web application (version 1.9)

File Type	Extension	Count
Include	inc	9
Code	php	1384
Template	tpl	567
Database	sql	34
Javascript	js	64
Icon	gif	958
Icon	png	176
Others	–	691
TOTAL		3883

per file. The code uses some 1100 include calls and 651 require calls to reuse common files.

The application contains 567 template files that use the *Smarty* template engine. The templates have 47000 lines together averaging 82 lines per file. The templates also have reusable components requiring 479 `include` calls.

The application also has 12000 lines of Javascript in 64 files, and 4600 line SQL file to initialise the 193 tables in the database.

The counts of files in Table 2.1 show the distribution of the 3883 files in the software package. The application makes use of the Pear database service, Smarty templates, the Adodb database interface to manage the architecture of such a large web application. TikiWiki also encapsulates the Galaxia workflow engine, which also adds to the size and complexity of the application.

The Smarty template uses `assign` calls to link data with an internal variable name. For example:

```
$smarty->assign('varname', $data );
$smarty->assign_by_ref('varname', $data );
```

The linked data can be the name of another template (eg: mymenu.tpl) to adapt the display using the referenced component. In TikiWiki there are 7115 calls to `assign` data to a variable, of which 283 are template component replacements. The actual template rendering is performed with a call like:

```
$smarty->display("templatename.tpl");
```

In the application, there are 1277 calls to render a Smarty template.

```
$url = 'tiki-view_forum_thread.php?comments_parentId='
      . urlencode( $threadId )
      . '&topics_threshold=0&topics_offset=1&'
    . 'topics_sort_mode=commentDate_desc&topics_find=&forumId='
      . urlencode( $_REQUEST["forumId"]);
```

Figure 2.5: Example of complex URL building code (from tiki-view_forum.php in tiki v1.9).

One of the most common code segments in the TikiWiki application is the creation of a URL from constants and variables. A typical example is shown in Figure 2.5. Extensive use of this repetitive type of code with complex syntax involving a range of carefully placed punctuation characters and URL-escaped variables increases error rates and maintenance problems.

The TikiWiki application is a large and complex application. The Smarty template library and Pear database library are used to enhance the web application, but complex, tangled, and repetitive code sections are common. Large web applications are difficult to design, build, and maintain.

The Plone Web Application

The Plone web application is a CMS built on the Python Zope web framework. The 91000 lines of python code reside in 803 files. There are a total of 513 template files containing 38000 lines. In Table 2.2 we see the distribution of file types in the 2536 files in the Plone application.

The Plone application is built with the Zope, which is included in the downloaded archive. Zope includes a web server, web server adaptors, template engines, an object database, and management functions.

There are 210000 lines of Python code in 1045 files implementing Zope. There is an additional 34000 lines of *C* code in 69 files that implement speed sensitive functions. Another 204 files contain 14000 lines of template code. The framework includes 438 files containing documentation.

Table 2.2: File statistics for the Plone web application (version 2.0)

File Type	Extension	Count
Python code	py	803
Template	dtml	224
Template	pt	241
Template	zpt	48
Javascript	js	37
Stylesheet	css	16
Document	stx	56
Document	rst	14
Document	txt	100
Icon	gif	171
Icon	png	82
Others	–	744
TOTAL		2536

Table 2.3: File statistics for the Zope web framework (version 2.7)

File Type	Extension	Count
Python code	py	1045
C Code	c,h	69
Template	dtml	204
XML	xml	56
Document	stx	190
Document	html	146
Document	txt	102
Icon	gif	97
Others	–	659
TOTAL		2568

The counts of Zope files are shown in Table 2.3. The Zope framework is the most mature of all Python web application frameworks, but it often criticised for its complexity (as hinted at by the 438 documentation files). Together, the two-layer Plone and Zope CMS require over 5000 files containing 380000 lines of code.

2.2.4 Problems in Building Web Applications

Large web applications typically grow from smaller web applications. The application might start out as a simple CGI script on a shared server, and be continually modified as the demand for the service grows. The number of users and the amount of content can expand rapidly (eg: the *Slashdot effect*). Programmers frantically extend databases, add

new features, glue on authentication, page templates, usage reporting, access control, RSS feeds, and dozens of other capabilities. New sections may be built within a framework (intending for the older code to be migrated in the programmer's free time), useful CGI programs installed, other web applications used as sub-services, and gateways to non-web applications. This tangle of code, displays, and data is held together by configuration files, hyperlinks, shared database tables, and custom protocols and mini-languages. Such a web application is illustrated in Figure 2.6.

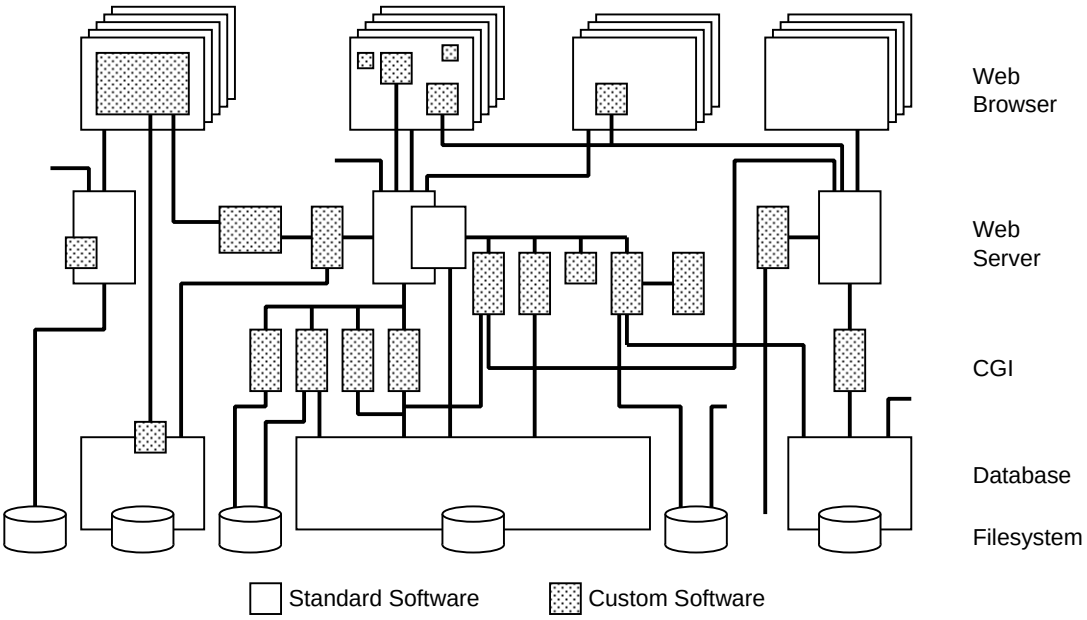

Figure 2.6: The complexity of real-world web application architectures.

This complex architecture is not the result of design. It is the product of rapid growth, the complexity of the technology, unclear user requirements, and a lack of standard architectures, patterns, and interfaces from software engineering.

The lack of architectural guidance and frameworks has encouraged web application designers to build designs that are based on previous experience or available tools. This has led to problems in maintaining many home-grown frameworks, protocols, services, and code libraries. This custom code is often written for an outdated version of a web protocol and single operating system or product, which limits integration options and interoperability.

Most web applications are designed to perform one task and integrating the functionality or merging the output with another application is difficult without deliberate design. There is currently no standard environment to ease the application interoperability task.

As many applications are designed without a scalable architecture, increasing throughput often means a significant redesign of the application software. Many applications use a set of CGI scripts, servlets, or PHP programs each of which generate a user interface, implement application logic, and execute transactions on the database. As applications grow this architecture becomes unmaintainable, and inconsistencies and subtle errors become common.

Tangled code also causes problems with the consistent look and feel in the user interface. Programmers build their own interface generation code within the application software and use hand coded HTML pages when possible. The different icons, navigation methods, layout, link conventions, and site structure increases the mental load on the user.

Web application software is hard to build, test, and maintain as the web environment is complex and always changing, making developed content hard to reuse. A multitude of modules, components, and interfaces are available for developers in each of the main languages of Java, C#, Python, Perl, Ruby, C, C++, and PHP but this does not solve the problem. Different approaches, non-standard interfaces, and complex functions create problems in reusing components within applications. Reusing components between languages and platforms has not been achieved.

Web application solutions are typically targeted at a single RDBMS product, a server side language processor, a particular web server, or a particular server or client platform. The choice of solution constrains the organisation to applications supported by the product and limits interoperability and migration to other environments.

The web protocols and technology offer little support for software engineering during development or maintenance. The ability to re-use developed HTML, Javascript, stylesheet, and applet content is vital for efficiency, consistency, speed of development, error isolation, and change control. Using an object oriented [Booch 91] web object structure allows the developer to use inheritance to build more specialised objects from base ones, aggregation to compose an object hierarchy, and encapsulation to hide data and methods within objects. Although object oriented languages are widely used (such as C++, Java, C#, and Python), there is little use of object orientation in *web engineering*, even though it is accepted as a vital concept in the rest of the software engineering field. The web protocols do not use object concepts, web servers do not use reuseable object structures, and

developers find it hard to make application code communicate objects to other services based on a different language or platform (although CORBA[Obj95] and more recently SOAP[Karmarkar *et al* 07] have been used with some success).

Intrinsic Web Application Limitations

The page-oriented nature of HTTP ensures that a page will remain visible in the browser even if the network connect is broken, the server crashes, or the user leaves the browser window open for days or weeks. The user may even be able to fetch the page from the local cache if the network or server is unavailable. This feature seems like a benefit; however this creates severe problems for the web application designer. A web application may be asked for a resource that has not existed since it was deleted weeks ago, so the programmer must design the application to respond to the user's request in a logical and consistent manner. This may require the display of an error message or the display of an updated or closely related resource.

This problem [Theng & Thimbleby 98] is typically referred to a the *Lost in Hyperspace Problem* or the *Time Travel Problem* because this error is caused by a request from a URL that existed previously in time. User bookmarks (or "favourites" in IE) only amplify this effect, as users may click on a bookmark to a dynamic resource from years before, which will be very confusing to the business logic in the web application.

The most significant danger from this circumstance is the re-issue of a GET request that has side-effects in the web application. The re-issue of a POST request will prompt the user for confirmation before continuing, but a GET request will be transparently sent to the server. A bookmark with the URL to a GET request such as:

GET /docfixer?filter="all"&action="delete"

could delete all the documents in storage, which may have been useful during the testing phase but has immediate and catastrophic consequences when selected (even accidentally) during operation. This is an extreme example, but other subtle effects are almost as dangerous. Competent web developers ensure that only idempotent requests use the GET method, and others use the POST method. The REST design pattern (see section 3.3.1) is very effective at mitigating the time travel problem, but requires a significant change in

application architecture with which many tools are not compatible.

A related problem which is common to all multi-user systems is *conflict resolution*. A user may view a page that indicates a resource is available, yet when a form is submitted to perform an action on the resource[3], the resource may have been deleted or changed by another user. The web application must recognise the conflict and resolve it (typically by displaying the conflict to the user and offering a choice of recovery actions). The impact of this conflict is exacerbated by complex transactions involving internal state. For example, the user may not wish the transaction to proceed if one item of the order is no longer available or has changed specification.

The page-oriented transaction protocol does not support real-time adjustment of display control widgets or provide real-time notification of a change of state in the web application on the server. The creative use of Javascript (see AJAX in section 2.3.4) can alleviate this problem with asynchronous data exchanges with the server, but no standard exists for real-time interactivity.

The browser is designed to render HTML using internal algorithms that implement the semantics of the HTML language. This reduces the UI programming load on the programmer by supplying many native UI components which are standard across all browsers[4]; however this automatic layout with native components limit the options for the web developer.

2.3 Web Application Design Techniques

2.3.1 Web server applications

An early method of building web applications involved the close integration of the custom code with *hooks* provided in the web server itself. The HTTP request and resource processing and transmission can be intercepted at any point to replace standard methods or insert additional methods. The Microsoft web server offers this feature through MSAPI, and the Netscape server uses the NSAPI.

[3]This resource may be goods at an auction, a car, a document, or a task. The action may be to buy, sell, hire, review, delete, or read. I shall simply refer to a *resource* and an *action* in this section.

[4]Browsers do not in fact follow standards exactly, and much programmer effort is required to ensure applications display and behave in the correct manner in each major browser release.

Building applications into the server with the API allows applications to extend web server functions such as authentication, access control, resource pre-processing, and logging, as well as typical business functions. This web application method results in fast and highly flexible applications, but offers no support for architecture, distributed components, multiple independent developers, or multiple languages. Developers are expected to take control of the web server and manage all aspects of server operations and extensions.

2.3.2 CGI programs

Some of the first Web Applications were built using simple programs called using the CGI protocol [Coar & Robinson 99]. The CGI program is executed as a separate process every time the web server receives a URL request from a client that uses a URL path that corresponds to that program. A string containing a set of name–value pairs is supplied to that program as the standard input stream (stdin) or as a command line argument. The program takes application-defined actions on the input data and returns output data to the web browser in the form of HTML page, GIF image, or other browser-understood format.

One of the biggest problems in developing CGI programs is the computing cost and response delay caused by the spawning of a new process to handle each web request. The web server typically forks, executes the CGI program in the child process, and passes request information to the program on its stdin file handle. The program executes, returns a HTTP header and MIME content on stdout, and then terminates.

2.3.3 CGI Scripts

Short programs written in a scripting language is a fast and simple way to build small web applications. Interpreted languages such as Perl and Python are ideal for this and are used widely. The flexibility and rapid development speed of the script-language solution is offset though by its additional load on the server and delay in user interface response. For a script-based CGI program, the web server has the extra overheads of initialising a scripting environment, loading script modules, and possibly dynamic compilation. Scripting programs also run slower and consume more computing resources.

In some ways a script program is more reliable that a compiled program due to the automatic memory management, exception control, and the absence of the common null-pointer error; however, the dynamic typing and object polymorphism used in scripts can mask subtle design and coding errors that might only become visible in production use.

Security is also a concern in interpreted scripts, as user entered data can be executed in a poorly written application, allowing hostile programs to be run using the host process user id and resources.

```
user_age = eval( page.age )
if user_age < 5 : theme = play_school
```

If page.age contains `"3"` or `"2007-1968"`, the program works correctly, however a user value of `"os.unlink('program.database')"` may cause severe problems. This *script-injection* attack vulnerability is a major reason against using scripting languages for internet-based (public access) web applications. Script applications have to be carefully designed to keep user-defined data separate from data which may be executed by the interpreter.

2.3.4 AJAX

The AJAX (Asynchronous Javascript and XML) approach is designed to mediate between the user and web services via scripts that execute within the client browser. The AJAX code is embedded with the web page or attached as a library, and performs actions as requested by user and system events. The primary aim of this technique is improved usability through higher user interface response.

User interface widgets, simple business functions (such as field validation), and context dependant data fetches improve interactive experience and minimise page transactions. Data is mostly transfered in the XML format, however alternatives such as JavaScript Object Notation (JSON) [JSONspec 06] and MIME types are now popular alternatives.

This technique has roots in 1996 and 1997 with the `IFRAME` and `LAYER` DHTML components, that were able to dynamically fetch content under Javascript control. The dynamic content technique evolved through several generations before reaching its current `XMLHttpRequest` form.

This usage of the web page as a “mini-application” brings a number of problems. The user is often unable to bookmark a particular information configuration, and must navigate through the dynamic page upon each visit. A related effect prevents a user from returning to the previous information view by using the ”back” button (instead, the browser returns to a previous ”mini-application” or launching page). Users may also become frustrated with delays in content update due to hidden client-server transactions. Search engines will also have trouble with indexing page content, as the engines use *static* content – not dynamic content.

2.3.5 Server-side template scripting modules

The web server can be configured to pass specially designated files through an associated script processor. The processor searches for encoded sequences in the text content, and substitutes a computed result in its place. The computed result may be a content fragment from the file system, a formatted data item from a database, or content created by a transform based on the coded sequence - such as a graph, bar chart, or formatted string. Typically, page names with a recognised *active* extension (such as .php .jps .asp) are routed to the appropriate engine for processing before returning the result to the requesting browser.

2.3.6 Database with server

Database vendors offer database management products with built-in web servers. The application functionality and user interface templates are stored in database tables and dynamically assembled to produce business applications. These products are designed to excel in data intensive applications as data transactions operate at high speed due to the close binding of the database, application code, and user interface components.

2.3.7 Java Servlets

Servlets are small application programs or components written in Java and configured with the web server to service requests from client browsers or software, in a similar way to CGI programs. Servlets can use Enterprise JavaBeans for rich functionality and JDBC for connection with databases.

2.3.8 Applets

Web applications are often built into a Java applet that executes on the client machine. The client applet will have its own multi-page GUI (using a Java library such as AWT or Swing) and conduct transactions with a remote database using JDBC in the familiar client–server pattern. The main web advantage being exploited is the freedom from managing the client machine software configuration - as it is downloaded from the server on each invocation. As the client CPU performs most of the business computation, the server has a lighter load.

Microsoft's ActiveX components are very similar to Java applets, and provide complete application development support with access to remote databases and integration with the client Microsoft operating system. These components are limited to Microsoft browsers, protocols, and operating systems so have constraints on portability and integration hooks, though they offer a rich functionality in a 100% Microsoft environment. These components are available to provide extra simple widgets up to complex client-server applications.

2.3.9 Client scripting

Simple application functionality can be built using client side scripts in the HTML pages. These scripts are mostly written in ECMAscript (also known as Javascript) and implement functions such as:

- Data field validation,
- Mandatory field completion checks,
- Dynamic response of the user interface to entered choices,
- Conditional content depending on browser type and version, and
- Graphical changes dependent on pointer location.

Client scripting is mostly used in conjunction with other techniques to improve the user experience by making the user interface more attractive, interactive, and functional. The JSON data format was created specially for Javascript clients to communicate with server processes. A simple JSON [Sim] is also available with direct mapping to Javascript objects and Python objects.

Some of the more common Javascript client-side libraries are:

- Mochikit [jsMochi 06]
- Dojo Toolkit [jsD]
- Prototype [jsP]

2.3.10 Web application frameworks

The web application framework is a cooperating suite of software components that support the common web application functional blocks. Frameworks typically bind to a web server (or implement an internal web server) and link to one or more web applications via developer configuration changes. Common components in a framework include session management, authentication, data validation, execution of application code in threads, database access (often with object-relational mapping), template based rendering, AJAX extensions, and error recovery. The framework operates by incoming web requests, interpreting the URL and associated data, and launching the appropriate application code to handle the request. The application would execute using extensions to base components and interact with framework services, before return MIME content to the framework for routing back to the requesting browser. Some frameworks are lightweight with one template component, a database interface, and a web server binding. Others are very complex involving hundreds of components and many framework interfaces and protocols. Very large and complex web frameworks with extensive support packages and tools could be called a *Web Application Software Environment*.

2.4 Web Application Software Environments

An option for large-scale web application development is the Web Application Software Environment. These software suites attempt to provide a complete solution to the web developer's requirements. They incorporate frameworks, communication protocols, common-use functions, reusable components, and development support tools. The advantages for developers include pretested inter-operating components, extensive documentation, user interface consistency, and a large community of users. The disadvantages are: complexity,

mandatory framework and design patterns, lack of interoperability with other technologies and environments, system and product dependencies, flexibility limited to current component function, reliability dependent on provided infrastructure, and useability of the user interface limited to current display components.

The environments are popular in large commercial development teams, as they provide a common infrastructure, protocol and toolset that all developers can build with, avoiding specialist technologies or products that isolate developers and create a maintainability problem when key skilled personnel leave.

Some of the main web application environments are described below and a discussion of the use of these environments follows.

2.4.1 Java

Struts is a Java framework for building web applications. It provides: a *request handler* to allocate web requests to a URI, a *response handler* to manage a resource that processes requests and generates a response, and a *tag library* to create dynamic web forms. Struts is designed on the Model-View-Controller (MVC) interaction pattern, and builds on J2EE technologies with servlets, Java Server Pages, and AJAX. *Stripes* is another Java framework that extends Struts with validation, file upload support, page wizards, and configuration improvements.

Shale is a new Java-based web application framework that weaves several strong Java technologies into software that is designed to succeed Struts. This framework is implemented as a modular set of services which enable flexible development of service based architectures. It uses a servlet container infrastructure, Java Server Faces (JSF), JSP Standard Tag Library (JSTL), several Jakarta Commons packages, and other powerful Java software.

Java Server Pages (JSP) are HTML or XML pages that have Java code sections that implement dynamic behaviour at display time. The first time they are viewed, the JSP is transparently compiled into a Java servlet (see section 2.3.7), and executed by the server Java Virtual Machine (JVM). This compiling phase introduces a delay on the first use, but users can expect servlet performance on subsequent use. The classes used in JSP

are JavaBeans (EJB), and the rich variety of EJBs (such as Object Relational Mapping (ORM) to a database) allow most classifications of web applications to be constructed. Web applications that have a design based on many template pages with small amounts of embedded business logic are well suited to the JSP development style.

The JSF is an application framework that includes a controller servlet, JSP templates, state maintenance, field validation, and support for custom components. It builds on many of the concepts in Struts and Swing and is IDE friendly. JSF was intended to complete with the Microsoft .NET environment.

Another Java framework is *Spring*. This framework uses the MVC design pattern to separate database, business logic, and presentation. The configuration is maintained in XML files that include page flow and navigation rules.

The Apache Software Foundation project *Wicket* is a lightweight web application framework that has a number of novel features that make it attractive to developers needing an object oriented approach with built-in design pattern support. Wicket comes with authentication, authorisation, a test framework, and a set of plain Java classes to inherit from. Applications run in a servlet container using XHTML compliant templates and plain Java code.

The *wingS* [Engels *et al* 07] framework brings the models, event, and listener features from Swing into the web environment. A servlet API governs user interaction, and Swing-like components are called via a session control layer when triggered by user events. The wingS framework is novel in that it successfully adapts a *desktop* Java framework to the web environment.

There are many other web tools and frameworks developed under the Apache Software Foundation. This Apache software includes major packages such as Jakarta, Tomcat, Axis, Geronimo, Turbine and Velocity.

The huge range of Java-based web application tools and frameworks offer many ways to build web applications. The majority of this software is mature and well designed, and in use in thousands of projects around the world. One of the limitations of the Java technology is complexity — Java has migrated from a useful tool to help develop web applications to a career choice. The complexity requires full time dedication and immersion to capi-

talise on the benefits of the technology, raising a culture of mono-technology specialists that place little emphasis on interoperability, simplicity, and elegance[5].

The Java framework software has constantly evolved through releases, redesigns, and inter-species competition to a current level of holding the standard in large scale web application development infrastructure. Its very success however has created a wall to adoption with the steep learning curve of hundreds of thousands of pages of documentation and hundreds of APIs. Java has a dominant position among many professional developers yet many smaller and simpler web application technologies thrive.

2.4.2 Macromedia

The Macromedia technologies provide a rich graphical UI for the user with high interactivity, attractive display widgets, animation, graphics transforms, and a wide variety of fonts. The custom client display engine, server infrastructure, and special-purpose client-server protocol is an alternative to the standard HTML over stateless HTTP designs.

The Macromedia web applications use Flash MX for user interface display and interaction and ColdFusion MX for server interfacing. The developers use DreamWeaver MX as an IDE to build the applications. The applications are rendered in the browser using Shockwave (SWF) media files which have embedded ActionScript code to implement application behaviour at the client side. Shockwave applications do not use HTML to interact with the user, but its own UI components much like a Java Applet does. Because of this *heavy client* design, most of the application processing is built into the SWF client which improves UI response time, frees the server from most application processing (improving server scalability), and reducing network utilisation. The clients communicate with the ColdFusion server via Action Message Format (AMF) transactions.

The server is built using Cold Fusion Components (CFC) that provide server-side functions such as database access. These components are created using the Cold Fusion Markup Language (CFML) and use the *Flash Remoting* interface to connect to clients, and the Cold Fusion MX infrastructure for basic services. Servers can also be implemented in Java or .NET technologies.

[5] In this software context, I intend elegance to mean having the properties of balance, efficiency, cognitive resonance, and linguistic clarity (ie: to the experienced eye, it looks *right*).

Adobe AIR

Adobe have created a new product that is based on a rich application platform for the desktop that is an alternative to browser-based tools. The AIR platform supports an enhanced set of SWF protocols and integrates with the local filesystems and desktop, incorporating standard desktop application functions such as cut-copy-paste and drag-and-drop. AIR applications are packaged and digitally signed, then installed on the client machines.

The AIR concept offers re-use of developers' skills and existing tools (in the flash domain) and integrates well with the Microsoft Windows desktop, but the technology has split from the standard browser-based web application architecture. AIR applications lack the ability to be integrated into other web applications and miss the benefits of server-based software, session, and data management.

2.4.3 Microsoft

Microsoft's tool for delivering dynamic web content is called Active Server Pages (ASP). The ASP pages are text files that are preprocessed by the Microsoft Internet Information Server (IIS) and sent to client browsers in the standard HTML format. The active part of the page can be coded in Visual Basic or Microsoft's JScript and connect to databases with ODBC. Hypertext pages can be dynamically created by server-side scripts, scripts within hypertext pages can add functionality, and embedded ActiveX components (which work similarly to Java applets) provide additional interactivity. ASP.NET Web Parts connects to SQL server and Internet Explorer sessions to provide a GUI development environment for web application construction.

Microsoft's current flagship web application environment is *.NET*. This environment provides the infrastructure to support a cooperating system of web components using presentation generation, business logic, and database access technologies tied together with the COM+ proprietary protocol. The components use the Microsoft C# language to extend supplied classes that provide common web application functionality. Applications are designed and implemented inside Microsoft Visual Studio, a sophisticated single-technology Integrated Development Environment (IDE) that uses wizards, drag and drop designers,

and interactive debugging. Some of the more powerful abilities of .NET include: database interfacing provided by the *System.Transactions* service, authentication implemented with the *Membership Provider*, page flow specified with the *Wizard Control*, and page content and behaviour encapsulated in a *Master Page*.

Provided that developers are willing to build complete web applications using the .NET architecture and infrastructure, the environment offers many components, services, structures, and tools to assist the web development team in rapidly building sophisticated web applications. The obvious limitation of this approach is the complete dependence on a single technology and product provider, and the imposed restrictions on interoperability with other non-Microsoft web application technologies.

2.4.4 Limitations in using Commercial Environments

Many development environments have been developed and sold to assist web application developers. This software is designed to be a product that generates profit for a company, so the architecture is constrained to "lock in" customers to the methodology, training, support contracts, design patterns, interfaces, and protocols embedded within the vendor product range. This insular and myopic approach to web application software offers some benefits in "one stop solutions" but limits the flexibility, integration, and interoperability of applications. Productivity is also adversely affected. Developers must use components that are not well suited to the requirement as this is the only option provided by the vendor. Errors in closed software can only be fixed by the vendor, so developers must wait for the error to be fixed, or substitute a "work-around" which is usually brittle and hard to maintain.

2.5 Open Frameworks

2.5.1 LAMP

Many web applications are built using what has come to be known as LAMP technology (Linux OS, Apache web server, MySQL database, and Perl, Python, or PHP code). All of these systems are open source and designed to work together. In addition, there are abundant extensions and tools to support the web application developer.

2.5.2 PHP

The PHP (PHP Hypertext Processor) module is one of the most popular open source web application development tools. There are over two million Apache servers running on the internet with an enabled PHP module [Sec02] as of May 2002. The PHP module is often used as part of a LAMP web application environment, which is a combination of free and open components that form an effective infrastructure for web applications.

The PHP program that implements the business logic is embedded in sections of extended HTML pages. The web server passes these pages to the PHP module for execution when the user sends a request for this URL. The PHP module executes the code sections within the HTML and replaces the code with HTML output — which is often fetched from a database. A rich array of functions, network protocols, content generation, and database interfacing are built in for application construction, and over 50 code libraries are available for many application support needs.

To improve speed, run-time compilers are available that compile and optimise PHP code on demand, then cache the high-speed code for subsequent access. A database access library called Pear provides some separation between the business logic and data access protocols and formats.

The Smarty [Maia 02] Template Engine employs a strong separation between business logic and presentation logic by using a PHP template library to handle HTML generation from within PHP applications. For speed and efficiency, Smarty templates are compiled into PHP code, may then be run with a PHP accelerator, and is cached for fast loading. Smarty can be extended with add-ons and plug-ins, but it is limited to PHP applications.

2.5.3 Python Web Application Frameworks

Python is a language that has deep support for web application frameworks, and Python developers have produced several mature frameworks and many experimental frameworks. The included web-support modules in the Python software package make it easy for developers to experiment with new web framework concepts, and this has been the genesis of many of the frameworks below. The abundance of many external web application support libraries has been suggested [Gregorio 06] as another reason for the many Python web frameworks currently available.

Django

Django is a popular and mature framework that manages UI templates, URL dispatching, caching, database operations, and many lesser functions. Web applications are built in Python using the defined APIs and scripts. Django can be configured to use many different types of web server front ends, and relational database back ends. It also has modules available for administration, syndication, authentication, and other common requirements.

Pylons

The Pylons framework complies with the new WSGI specification, which allows it to support middleware components at many layers in the web application processing chain. Database interactions are controlled through the SQLAlchemy or SQLObject ORM modules. The output content can be generated by a number of template-based modules:

- Mako,
- Genshi,
- Jinja,
- Kid,
- Cheetah,
- or other generator compatible with the Buffet middleware.

The framework is adapted for use with Javascript libraries (such as Mochikit and Prototype) for AJAX-oriented applications.

TurboGears

TurboGears [Ramm *et al* 06] had its first release in June 2005, and rapidly became popular as a useful and stable web application framework. It was designed in reaction to the multitude of complex and hard to maintain frameworks that tangle application logic, user interface code, and database code. TurboGears uses the MVC pattern to separate its three main components: the SQLObject database interface, the Kidd templating engine, and CherryPy web server interface.

Zope

Zope is an open source web application environment produced by Digital Creations and is one of the oldest and largest of the Python frameworks. The Zope package includes an internet server, a transactional object database, a search engine, a web page templating system, a management tool, and extension facility.

An internal multi-threaded web server called ZServer is included in the Zope framework, but Zope can also bind with an external web server using FastCGI or Persistent CGI (PCGI). The Apache web server has a FastCGI module which can send requests to the Zope application server without the overhead of forking a new handler or initialising a program. The PCGI option is a Python module that manages the running of the Zope server which is less complex that FastCGI, but still requires a new handler to be forked for each request.

Webpy

The web.py framework was developed by Aaron Swartz to fill a need for a very simple but robust and fast framework. It is WSGI compatible, integrates with the Cheetah template engine, and uses the error page generator from Django. Incoming URLs are mapped to class and method names, which are called in a separate thread. This high transaction-rate Reddit web site runs this framework.

Webpy is designed to be lightweight yet it includes its own Python-like language parser. The parser is designed so that script-injection attacks can be prevented and developers can use a familiar language for template behaviour design.

Karrigell

The Karrigell framework [kar08] takes a lightweight but full-featured approach. It includes a simple non-blocking Async web server but also has adaptors for Xitami and Apache. It uses the simple KirbyBase database or the Python GadFly SQL database to manage data. The framework manages sessions, basic HTTP authentication, and error reporting. A unique feature is the four different methods that can be used to implement application logic and presentation — application code and presentation markup can be combined in

different styles (see Figure 2.7) depending on the preference of the developer and the complexity of the application.

```
1:  <h1>Squares</h1>
2:  <%
3:  for i in range(10):
4:      print "%s :<b>%s</b>" %(i,i*i)
5:  %>
```

Figure 2.7: Example of the *Python in HTML* template code style.

The many other Python web frameworks include Web2Py, CherryPy, Myghty, Cymbeline, CleverHarold, Quixote, Skunkweb, Spyce, Divmod, PEAK, Snakelets, Albatross, Wasp, Aquarium, Spark, HTMLgen, Python Service Objects, and Webware.

The Python WSGI Standard

There are currently dozens of web frameworks based on the Python language. The larger and more popular frameworks have been described above in this section. This menagerie of frameworks has split the Python development efforts, causes confusion in users, and acts as a barrier to the reuse of tools, components, and modules.

The WSGI protocol [Pyt06] is a set of interface specifications which divides the web application control flow into a number of distinct layers. Software developers can build software to fit into a layer, and system integrators can select that software for the selected function in the WSGI *stack*. There are middleware modules available [Ste05] for sessions, authentication, validation, URL routing, and presentation generation. Most large Python frameworks have updated their codebase to be compliant with the WSGI specification.

The WSGI is a Python-only architecture, but it suggests a way forward in separating concerns in the web application design. This design pattern would provide a way for web application developers to interchange, insert, and upgrade functions in the web application flow minimising the risk of interface non-compliance and increasing modularity for enhanced maintainability.

2.5.4 Ruby on Rails

The Ruby language originated in Japan in the early 1990's, and gained a following among programmers due to its combination of Smalltalk's conceptual elegance, Python's ease of use and learning, and Perl's pragmatism [Walton & Hibbs 06]. The Ruby On Rails software is a web framework that is intended to reduce the number of lines of code programmers have to write, and work without the need for verbose configuration files. The number of lines of code and complex configuration files were seen by the designers of Rails to be an impediment to the development and maintenance of web applications [Black 06]. The website [Rub07] describes rails as:

> *Rails is a web-application and persistence framework that includes everything needed to create database-backed web-applications according to the Model-View-Control pattern of separation. This pattern splits the view (also called the presentation) into "dumb" templates that are primarily responsible for inserting pre-built data in between HTML tags. The model contains the "smart" domain objects (such as Account, Product, Person, Post) that holds all the business logic and knows how to persist themselves to a database. The controller handles the incoming requests (such as Save New Account, Update Product, Show Post) by manipulating the model and directing data to the view.*

Another writer [Rustad 05] describes the benefits of developing in Rails as:

> *Rails prefers explicit code instead of configuration files, and the dynamic nature of the Ruby language generates much of the plumbing code at runtime. Most of the Rails framework has been created as a single project, and application development benefits from a set of homogeneous components.*

In the Rails MVC architecture, the model layer is managed by an ORM called *Active Record*, which handles the mapping of object to and from relational database entities. The corresponding controller layer is called *Action Controller* and the view is called *Action View*. The controller and view functions are closely bound in Rails, which reduces the lines of code that implement business logic components with display responsibilities but negates the *separation of concerns* and *loose coupling* software engineering features of MVC.

One of the strengths of Ruby on Rails is all the web servers it is designed to integrate with. It is preferred to use the Ruby web server: Mongrel, but can also work with Lighttpd and attach to Apache, LiteSpeed, and IIS servers through FastCGI. For local prototyping, a simple Ruby webserver called WEBrick can be used.

There is also a strong preference in Ruby on Rails towards code generation. A generator is used at the beginning of a project to build the code directories and stubs, then again during each development iteration to create new files that support changes in the database. Code generation is rarely used in frameworks as it constrains the developer in how the web application is designed and maintained. If a desired feature is not allowed for in the generated framework, the generated code must be changed — this can cause unforeseen side-effects in other parts of the framework and may be overwritten by the code generator in the next development iteration. Although useful for beginners, and development of small applications that follow the framework designer's intent, the drawbacks for professional web development are substantial.

2.5.5 Mason

The Mason web framework became a popular way of building small Perl web applications in 2000 and 2001. It is a platform-independent Perl based framework that supports web development through URL request routing to components and output generation with templates. Blocks of HTML and Perl can be configured as reusable components. In the example Mason "component" shown in Figure 2.8, a local variable assignment is made in line 1, followed by HTML text with an embedded variable reference in line 3.

```
1:  % my $greeting = 'Hello';
2:  <h1>Message</h1>
3:  <p><% $greeting %> World.</p>
```

Figure 2.8: An example Mason component.

The Mason documentation [Web05] provides a description of the framework:

> *Mason's various pieces revolve around the notion of "components". A component is a mix of HTML, Perl, and special Mason commands, one component per file. So-called "top-level" components represent entire web-pages, while smaller components typically return HTML snippets for embedding in top-level components. This object-like architecture greatly simplifies site maintenance: change a shared component, and you instantly changed all dependent pages that refer to it across a site (or across many virtual sites).*

Other Perl frameworks include Catalyst, Maypole, and Jifty.

2.5.6 FuseBox

Fusebox [Quarto-vonTivadar *et al* 05] is an application framework that builds on ColdFusion and PHP. It uses components called "circuits", and a pipe-and-filter architecture. Configuration and application behaviour are described in XML, which includes variable controls, loops, and branching syntax. FuseBox consists of the Runtime, Loader, Transformer, and Parser modules. It is a combination technology that in some ways offers the best aspects of ColdFusion and PHP; however, no new approach is offered to tackle the more difficult problems of the web application designer, such as collaboration, maintainability, encapsulation, or reuse.

2.5.7 ClearSilver

ClearSilver is a high speed templating engine written in C. It merges templates and structured data files dynamically to generate web content. The ClearSilver software does not offer the developer much more than high-speed template-based rendering, so web applications that use ClearSilver will have to be combined with other software to support all the required functions of the application. Another limitation is the custom Hierarchical Data Format (HDF) specification for input data files and the custom template language.

This huge range of web application frameworks and environments each have advantages for the developer; simplicity, completeness, integration hooks, flexibility, range of components,

or compliance with standards for example.

In the next section, we will explore some research projects that have investigated aspects of the web engineering problem and have produced an experiment or concept demonstrator.

2.6 Web Application Research Projects

The WOOM (Web Object Oriented Model) [Coda *et al* 98] has been developed to add a layer of abstraction to the web application design process. This system divides web sites into resources, elements, sites, servers, links, and transformers. A framework is also provided to convert a model into an instantiation in the web environment. The goal is to bring an abstraction layer, separation of concerns, modularity, and flexibility to web application engineering.

It has been observed that building web applications is a complex and time–consuming process [Schwabe *et al* 01] and new design techniques are needed to manage the design complexity and facilitate reuse [Gellerson *et al* 97] of design. The OOHDM–Frame has been developed [Schwabe *et al* 01] to determine key architectural components and design structures that lend themselves to reuse. This model separates application behaviour, navigation modelling, and user interface design — offering design reuse across these three domains. Schwabe et al. have found that systematic reuse of design parts is a key approach for maximising reuse in web application development.

A common technique is a dynamic HTML page generation using a template and application supplied variables. There was several early generators such as HTML++ [Schranz 98], then a multitude of popular but incompatible languages such as PHP, ASP, JSP and dozens of minor dialects. New template languages are still being developed (Google cTemplates [Goo06] for example).

The Hera [Houben *et al* 05] model driven design methodology uses a presentation generator to build web information systems from RDF resources.

The Strudel [Fernandez *et al* 00] site implementation tool offers a structured query language joined with a rudimentary HTML templating system to build data–driven web sites.

The WebRB visual programming environment [Leff & Rayfield 07] allows developers to

visually create and edit relational algebra components and relationships in a single tool. The declarative language representation of the application is then executed, implementing a data–driven web application. This design has a steep learning curve and limits interoperability and integration.

Web applications can also be structured around creative Javascript libraries. The KnowNow [Zhao *et al* 02] architecture uses dynamic interaction with the server via an event router that links the HTML user interface with server logic.

Another solution investigates cooperating agents [Ciancarini *et al* 98b] coordinating in *PageSpace* to execute application logic, dynamically connect to external services via gateway agents, and interact with the user with user-interface agents. PageSpace is based on the combination of Linda TuppleSpace and Java Applet technologies. Browsers that load the base applet are able to host application agents which can interact with the user, communicate with other agents in the same browser or remote browsers, and exchange transactions with the server. This is a novel solution that provides an excellent infrastructure for collaborative applications, however reliability, scalability, and security remain challenges for this style of architecture.

WebComposition [Gellerson *et al* 97] uses Web Composition Markup Language(WCML) which is used to define web components, properties, and relationships. The WCML is based on XML and uses prototype-instance reuse. Supplied components support the set, get, and generate methods, and can be extended with the use of prototype inheritance. When the site is built, components are fetched from the attached RDBMS, assembled into pages, and then stored in the web server file system for access by the web server when HTTP requests are received. This is an innovative approach to the management of large static web sites, but does not address the dynamic aspects of the web application development domain.

Another object oriented web component environment designed for re-use and management is JESSICA [Barta & Schranz 98]. The system uses a custom language and Java based compiler to generate a web site from a set of JESSICA templates and objects.

The WebBroker is a process which runs on the web server and uses XML-RPC, DCOM, RMI, and CORBA to communicate user HTTP requests to distributed software objects.

The CGILua [Hester *et al* 98] is a web development tool that is implemented as an extension language which is embedded in HTML and processed by a CGI program. It has a simple language including regular expression matching, database access, and dynamic linking to C++ libraries.

The W3Objects [Ingham *et al* 98] is a distributed object service based on RPC communication and a TCL-like scripting language. The objects have a management interface and multiple views.

An early attempt [Brown & Najork 96] at distributed collaborative network of components used *Oblets* written in the special purpose language *Obliq*. The Oblets were configured and programmed to provide application functionality by communicating between browsers, allowing users to collaborate dynamically. This was a peer to peer collaborative application architecture that used browser components for architectural building blocks.

All of these research projects mentioned are designed to provide a technique or technology to assist web application developers in building professional applications efficiently. They each propose a different solution that addresses a different subset of the problem space.

2.7 Related Technologies

2.7.1 SOAP Web Services

The most popular and formal set of protocols used to implement web services are based on the Simple Object Access Protocol (SOAP) [Karmarkar *et al* 07]. Web services are processes that advertise an API and set of functions for other software to use via web protocols (typically HTTP). Web services do not interact with browsers or users, but web applications may use or be based on web services, and display content that was indirectly supplied by web services.

The SOAP protocol defines an XML envelope specification around an XML payload that can be transmitted to another process using a variety of transport mechanisms. SOAP messages are commonly sent over a TCP/IP socket as part of a transaction exchange, but are sometimes sent over SMTP or other asynchronous message orientated protocols. The content of SOAP messages can be encoded using a number of XML dialects and structural

extensions. Name spaces are defined with the XMLNameSpaces [Layman *et al* 06] protocol which provide context for tag and attribute names. A richer data structure specification is provided by XMLSchema [van derVlist 02] where new and compound data types can be defined for validation, interoperability, and interpreter designs.

Web services that use the SOAP protocol can use the Web Services Description Language (WSDL) [Ryman *et al* 07] to describe the functions and parameters of the web services. This description can be used by clients at run-time to dynamically locate and call a service with the required function. A web services broker uses UDDI [Clement *et al* 05] to exchange WSDL with clients ans services, and can redirect clients to needed active services based on service requirements.

The set of SOAP-related protocols bring distributed systems architecture into the web applications domain. The sophisticated data format specification, technology-agnostic transport, service advertisement, and dynamic client-service binding lay the foundations for large-scale enterprise web systems. In many ways the sophistication and complexity of this suite of protocols has limited its acceptance. Incompatibilities between web service software due to incomplete implementations, mistakes in software construction, and custom extensions to protocols have made the goal of transparent interoperability hard to achieve. This complexity is not only a hindrance to the building of robust software systems, but requires highly skilled and educated developers who have devoted time reading extensive documentation and gathering expertise in these protocols.

Alternatives to the SOAP protocol for web services are XML-RPC, Java RMI, Microsoft COM+, Corba, DCE, and plain URL input and MIME output over HTTP.

2.7.2 Rendering User Interfaces with XML and XSL

Another method of dynamically generating web content is via transformation of XML structures into XHTML [Ishikawa *et al* 07] markup code using XSL [Berglund 06]. The XSL style sheet defines the mapping of XML elements and contents into the XHTML stream, and XSLT [Kay 07] can be used for more complex transformations. The mapping uses a series of rules that search for *patterns* in XML elements and apply *actions* to produce the final XHTML content. This technology was introduced in 1998, and was used in some

frameworks [Kristensen 98]; however most dynamic web content is generated by processes that use methods separate from the XSL standards.

Similar operations can be applied to incoming XML to produce SVG [Ferraiolo *et al* 03] content. This use of style sheets could map an XML fragment containing data to be displayed in graphical form into a spatial map, a business chart, a diagram, or other vector-based graphical output.

2.7.3 Web 2.0

In recent years, there have been innovative developments in information tools focused on using the properties of the Web. These developments are often called Web2.0 technology, however this term is widely abused [O'Reilly 05]. The keys to this new web technology is *user participation* and *user owned data*. The Web2.0 sites offer *interactive* functionality, providing a useful service for clients, and linking client information together using social network techniques.

Developers designing applications for Web2.0 are using new tools and technologies to build better applications in less time. New web applications and services are mostly built on frameworks now. It is difficult to justify custom designs when there are so many highly functional frameworks with pre-tested components and trusted security. There are also many content management systems (CMS) packages now available that can be used without further development by user communities needing to create and manage structures of web content.

The increased use of Javascript has improved web page appearances and dynamic update behaviour. By using Javascript, user interface components can quickly validate user data, expand showing more information, implement menus, perform animations, and load new data on demand. The development of Javascript components that operate like a word processor application (eg: FCKedit and TinyMCE) has eased the migration of users changing from desktop computing environments to user services on the Web. The alternative *light markup languages* are non-trivial to learn and lack many of the features of a WYSIWYG editor[6].

[6]It could also be argued though that the use of light markup promotes consistency of appearance and reduces frivolous text decorations.

Writing glue code (typically Javascript) to make two or more web 2.0 applications with lightweight APIs work together is called a *Mashup* and can be used to combine multiple web applications. Another option is hosting custom web applications in a large web application environment managed by an external organisation. The Google engine (Gears) and the Amazon E-Commerce site offer a simplified API for developers to use for embedding custom applications that use a common web infrastructure and data storage engine.

Older web applications arranged content in a hierarchy and limited connections with content to explicit URL hyperlinks. The Web2.0 methods of navigating through information and linking content with external resources include tagging, RSS feeds, and mapping toolkits. Users prefer to use their own terminology for keywords (tags) describing content, rather than navigating a taxonomy of terms to select the most appropriate one. This has been called tagging by *folksonomy* and results in multi-dimensional navigation paths. Changes in Blogs or other content of external Web Applications can trigger syndication (RSS) messages to be sent to subscribing software, signalling new material. This notification by opt-in protocol has transformed the web of static hyperlinks to a *Live Web*. Mapping services offer tools for users to add data to the map environment and an API to integrate the service into other tools. The mapping functions allow linking by geospatial proximity and the simple visualisation of content locations.

Many web companies are now offering desktop-like productivity tools that have the added benefits of secure off-line storage and collaborative features. Many sites are now offering free storage for user data. The benefits are availability from anywhere and the protection of a large data centre, but questions remain on who owns the data and the user's rights to privacy. Several *webmail* services provide an email address and email program functionality. A central service that manages email is attractive to users because of the global accessibility, ease of configuration, and integration with other services. The Google organisation has experimented with many innovative user tools based on web technologies. Pure web applications such as the calendar, document editing, web page editor, and spreadsheet show how users can potentially use web-based user services for all their computing needs. Web based commercial transactions have become robust and ubiquitous. Large e-commerce web companies such as Ebay, Amazon, and PayPal are constantly improving services to users, exploiting emerging web technologies to enhance payment handling,

the user experience, and reputation management. There are many options on integrating new web applications with these services. The support for *issue tracking* has become an important pillar of collaborative work in the Web2.0 environment. The ability for a user to log an issue and track its progress to resolution provides a sense of shared ownership in the enterprise. It also assists the problem solvers in understanding current user issues with the service, and helps manage progress towards the solution.

The increased popularity of communications services is a significant characteristic of the Web2.0 community oriented nature. The communications methods are stll dominated by email, but other methods have secure places in the Web2.0 environment:

IRC and IM: Inter-Relay Chat and Instant Messaging are communication tools that allow groups of people to exchange short text messages in a defined *chat room*, much like a face-to-face conversation.

Online real-time 3D games: The high penetration of broadband Internet into homes across the Western World has fuelled the growth of *multi-user games*. The games use the Internet for passing message traffic between a game server and many game clients, but have little to do with the Web itself.

Subscriptions: Information services strive to make content attractive to their user demographic, and then offer the material as a subscribed service, linking the user to the website with a loose producer-consumer relationship. This subscription provides the website with a channel to invite the user to explore other content and services, and generate revenue through placed adverts.

A main component of Web2.0 is the support of social networks. These network links are within sites and between identities in external sites. The links are based on a *shared community of interest* and may be permanent or transient. Some sites concentrate on supporting the communication and expression of users. The well known *social network* sites (eg: Facebook, MySpace, Orkut, Friendster, LinkedIn) provide spaces to express interests and support communication and communities of users with similar interests. Other sites are based on bookmarking. These types of sites (eg: Technorati, Reddit, del.icio.us) make it easy for users to manage their bookmarks for URLs on the web. Additionally, other related web resources can be suggested based on the correlation of a user's URLs with

other user's URLs. Another form of social network is centred on a media type. Some Web organisations have built a large user base from providing user value in handling media. The Flikr, YouTube, and Picasa web sites are examples of user media services that do one thing, and do it right.

The discussion forum and blog can also be considered to support social networking. The community forum is a threaded group discussion with a complete message for each entry (unlike a phrase in IRC). The forum may be used for information sharing or the reach a decision, and can be kept as an online knowledge resource. A social network based on blogs is called a *blogosphere* and support communities of interest by linking blogs. A blog is basically a chronologically ordered online journal; however the real benefits of the blogging environment are the RSS feeds that notify interested parties, the comments that can be added to blog entries (forming a distributed community discussion between a loose network of blogs), and the creation of *Permalinks* (permanent URIs) for each entry for reliable referencing.

Another feature of the Web2.0 environment is the availability of global knowledge. The best example of this global knowledge store is *WikiPedia*, a web-based encyclopedia that uses a huge community of contributors and editors to store and link knowledge. It uses a *soft security* feature that allows any user to change any content, but provides simple tools for editors to check, change, or reverse additions. This resource can be integrated and linked into web applications. Smaller sized knowledge repositories can be created using commonly available wiki software. Rapid content creation and linking can be performed with this style of collaborative web application. Simple text can optionally be enhanced with special punctuation sequences to build readable hyperlinked pages, lowering barriers for new web users and increasing productivity of information workers that require simple management of notes and knowledge capture.

An essential part of the new Web is attracting users with web services containing high-value content and simple functionality. Successful sites will also have simple methods for programmers and content aggregators to make use of site services. Web sites will tend to be in constant refinement based on continuous feedback from user behaviour, and supply a service in a simple and intuitive user interface.

The Web 2.0 way of thinking defines the environment for web applications in the near

future. This suggests a number of desirable properties of new web application development software:

- To attract users, new web applications will require high quality user interfaces. The UI must be consistent and intuitive, yet be able to sustain constant refactoring.
- A website in constant development will need to be based on software that is modular, reusable, flexible, configurable, and implements a separation of concerns.
- To attract programmers who will link with our web service using a variety of methods, the software should have a variety of lightweight integration options.
- New services will need to display and interact on a variety of devices, so support for multiple UI technology and transaction patterns is a growing requirement.

Many of these properties align with the web application properties given for evaluation in the introduction (see Figure 1.1), providing a method of validation, and some indication that these properties will also be important in the next generation of web application technology.

2.8 Summary

We can see from this chapter that web application development has seen a lot of innovation, software experiments, hundreds of new protocols, standards, and interfaces. Despite this, the web application designers job is harder than ever — the complexity of the tools combined with the complexity of the web environment and engineering requirements is difficult to understand, communicate, and manage.

In the next chapter, I propose a technological solution that has the potential to encapsulate some of this complexity in a reusable service. This service is called a presentation service, and can increase the productivity of web application developers while increasing the manageability of the product and the quality of the user experience.

Chapter 3

A Presentation Service

3.1 Introduction

We have seen that the web application developer is faced with a number of problems when designing, implementing, managing, and extending web applications. Most of these problems are with managing the software that controls the browser user interface. The other parts of the software that implement business logic and interface to data stores are better supported by techniques and products, although subtle influences caused by web application design choices can complicate these parts too. This chapter discusses these problems, proposes a solution based on a separate presentation service, and describes the architecture and high-level design of the concept.

3.2 Web Application Problem Space

Contemporary web application designers are faced with a complex array of limitations, pitfalls, and productivity drains that reduce development effectiveness and increase risks.

User Interface Entanglement: Many of the existing tools and frameworks are designed for code that shows close coupling between the user interface and application logic. This often seems to be the best design choice as it combines the application code and UI code that work together. The problem only manifests itself when the code grows larger and the developers realise that the UI code is spread around the whole

codebase, introducing subtle incompatibilities, inconsistent appearance and function, and making maintenance and enhancement a difficult task.

Browser Compatibility: Each developer in the web environment must contend with multiple browser vendors, release versions, and platforms, each having subtle incompatibilities and non-perfect standards compliance. The developer must not only learn and keep up to date on a multitude of languages and protocols (such as HTML, CSS, HTTP, XML, XSL, URI, and Javascript), but needs to code for each browser possibility and test to ensure the intended code works for each possibility.

Complexity: Developers are also required to build code connecting to other user interface technologies (ie: flash, bitmap libraries, geospatial clients), and maintaining skills and running code in multiple evolving technologies has high developer costs, introduces extra complexity into the project, and increases the risks of project failure and application faults.

Collaborative Sessions Support: Most applications benefit from collaborative work functions however application developers are required to build collaborative capabilities in ad hoc ways, customised for each application, or put off these advanced features for later versions. Multi-protocol collaborative sessions are even more complex and attempts at implementing this has been only partially successful.

Inconsistent UI: Software with user interface code interwoven with application code tends to suffer from inconsistency in appearance (such as a button labelled `OK` somewhere and `Accept` somewhere else) and behaviour (such as a `Delete` button clearing field contents on one screen compared with deleting the all the screen items and the record on the database elsewhere). Inconsistency affects user trust in subtle ways and can lead to product or service failure.

A related inconsistency problem happens when two or more teams develop applications for the same user base and build the user interface within the teams. The result is often both inconsistent user interfaces and poor integration.

Stovepiped Applications: When user interface design is closely bound with applications, the resulting software is difficult to integrate. A single hyperlink or button may open the related application on its front page, which hardly satisfies the user

need to see information from several applications merged on the same screen with options for actions in either application. An integrated user interface requires intimate collaboration between teams and increased work, or interfaces and software to provide the capability.

Redundant Work: User interface code that is closely coupled with application code cannot be reused in later projects, despite the user interface requirements being very similar. Many libraries are available that provide some of these reusable user interface functions though they can only support one language or platform, and are very complex to learn and use. Simple multi-language cross-platform user interface software is difficult to develop.

Vendor Lock-in: Current web applications are mostly built using Java, .Net, PHP, or other environment that ties the application to the environment, unless a custom interface is developed to support special transactions with a different environment. This single environment is mostly not a problem as this arrangement is more efficient in developer skills and reduces complexity. Unfortunately the single environment prevents applications having the best functionality from several tools based on different environments. A method of combining the capabilities across environments has to be found before these benefits can be realised.

Page Oriented: Web applications are built on the HTTP protocol which uses synchronous page oriented transactions. This coarse grained protocol limits interactivity with large and slow page loads for each mouse click, and consumes network bandwidth. There have been many web components and libraries that have been developed to ease this problem by supporting many small transactions and updating small parts of the page as required [Zhao *et al* 02], but a general purpose robust and simple solution remains elusive.

Complex Extensions: When user interface code is spread through the application, the ability to adapt and extend the user interface and the business logic are both affected. It is harder to find the right part of the code, and a change to the user interface may have an unexpected effect on the business logic, and a business logic change may cause a side-effect in the user interface. The perceived efficiency in keeping both types of code together may be attractive at the start of the project; however the cost and risk is much higher over the life of the project.

Diagnostics: To exacerbate the situation, the closely coupled code also makes it hard to find errors, fix, and test them. Web applications are rarely built within a testing framework, so simple diagnostic tests such as inspecting the web server log and inserting print statements into code are used. These simple methods are not efficient, especially in large and complex software. When correcting errors in current typical web applications, changes may also have unforeseen side effects, leading to application faults or additional software repairs.

Lack of Graphics: Web applications developers have used many ad hoc techniques to add maps, diagrams, and business charts to the browser user interface, but the work is hard to generalise and adapt to different requirements. The resulting functionality is often simple and clumsy despite the large complex code used to implement it. The smooth rich graphical environment that desktop applications excel at is extremely difficult to design into web applications, and usually not worth the effort for less than major projects.

UI Coding Effort: The effort and expense in redeveloping the user interface handling code for each application or project could be reduced if a general purpose software package could be designed that could handle most of the user interface complexity internally and expose a language-independent platform-independent simplified API.

These web application design problems have been addressed by many different technologies, protocols, frameworks, and products, yet each offering not only solves only a subset of the problem space, but can also introduce additional problems (such as language or platform lock-in and complex new interfaces to learn). The web application design domain requires a new way to manage web user interfaces and simplify the task of designing and implementing new web applications.

3.3 Design Concepts

The software engineering world uses many design concepts to aid in the creation of complex software. These concepts categorise sections of the software system and extract the core section properties and transactions between sections. Each of these concepts adds a different view on the software design problem.

The main web engineering concepts are described by Fraternali in a survey of web development tools [Fraternali 98]. In this survey, the design dimensions of Structure, Navigation, and Presentation are emphasised. The critical properties of component reuse [Lee & Shirani 04], three tier architecture, and user interface useability were also identified.

The benefits of separating presentation code from business logic has been described in Sun's Enterprise Applications blueprint document [Sun02] as:

- Minimises impact of change–Business rules can be changed in their own layer, with little or no modification to the presentation layer. Application presentation or workflow can change without affecting code in the business layer.
- Increases maintainability–Most business logic occurs in more than one use case of a particular application. Business logic copied and pasted between components expresses the same business rule in two places in the application. Future changes to the rule require two edits instead of one. Business logic expressed in a separate component and accessed referentially can be modified in one place in the source code, producing behaviour changes everywhere the component is used. Similar benefits are achieved by reusing presentation logic with server-side includes, custom tags, and stylesheets.
- Provides client independence and code reuse–Intermingling data presentation and business logic ties the business logic to a particular type of client. For example, business logic implemented in a scriptlet is not usable by a servlet or an application client; the code must be reimplemented for the other client types. Business logic that is available referentially as simple method calls on business objects can be used by multiple client types.
- Separates developer roles–Code that deals with data presentation, request processing, and business rules all at once is difficult to read, especially for a developer who may specialise in only one of these areas. Separating business logic and presentation allows developers to concentrate on their area of expertise [Parr 04].

3.3.1 Engineering Techniques

There are a number of software engineering techniques we can consider. These techniques are popular in mainstream software development but often have only partial adoption by the web application design community.

Object Orientation

Most of the useful measures of effectiveness in software systems, such as reuseability, flexibility, reliability, maintainability, and useability have proved difficult to measure. In 2005, researchers reported [Darcy & Kemerer 05] on a number of efforts to correlate various object orientation attributes of software with fault rates, reuse, and maintainability. There appears to be no conclusive evidence that would provide absolute design rules; however internal object cohesion and a lack of coupling were associated with better designs. Surprisingly, inheritance was not seen as a significant factor in the resulting software quality (although it may have a positive affect on other parameters - such as productivity and reliability).

One of the simplest and useful methods of reuse is through prototype based inheritance [Ungar & Smith 87] [Taivalsaari 96]. The WebComposition [Gellerson *et al* 97] prototype found this inheritance method was lighter than class inheritance, but still allowed abstract objects which could be extended.

Three Tier Design

By encapsulating user interface functionality in a separate service, the applications need only contain business logic. The common data, communications, and user interface functions are separately designed, implemented, and managed using the appropriate data, communications, and user interface services. The addition of a presentation layer creates a true three tier application architecture (see Figure 3.1).

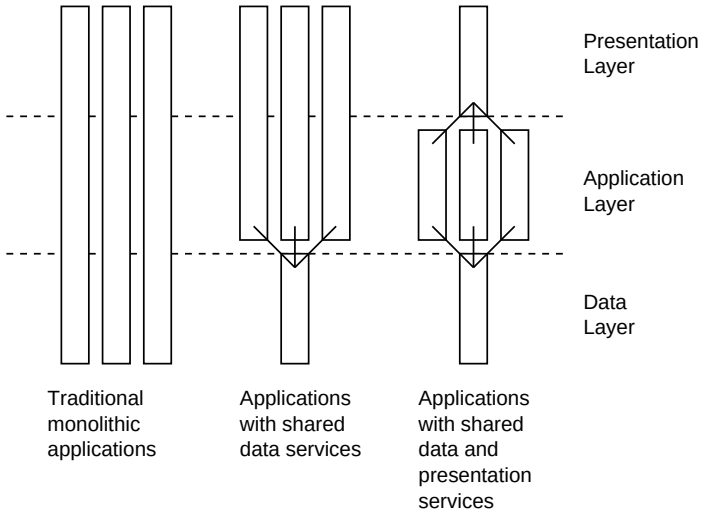

Figure 3.1: Shared services in a three tier model.

Model-View-Controller Separation

Contemporary web application frameworks often use a Model-View-Controller (MVC) [Cox & Novobilski 86] design. This pattern makes a logical division in the application between the presentation (view), business logic (controller), and the database (model). This separation of concerns assists developers in managing the complexity of the design and implementation [Krasner & Pope 88]. In the web application domain the *view* layer is implemented by a template driven generator or an object serialiser. The *controller* is typically a multi-threaded request handler managing simultaneous request-response cycles through implemented business logic and rules. The *model* may be specialised to implement page navigation and state persistence, but is commonly designed as a standard application database.

The MVC pattern is an important design technique to use in the presentation service. The service will provide a universal view layer, and also capture user events and transmit them to the controller (which would be inside the application).

Inversion of Control

Many web applications use a technique called *Inversion of Control* (IoC). An application uses this method to bind itself to a web request handler and then it passes "control" to

the handler and waits for requests. The handler will process an incoming request, and if the URL matches a pre-configured pattern, the web application is sent the request, and the application assumes control until the web content is returned to the handler, where the web application will then "sleep" again.

The IoC technique is applicable to the design of the applications connecting to the presentation service. The applications will be primarily reactive, and spend most of their time idle, waiting for a user event. The presentation service acts as a server to both UI clients and application clients.

Usage of Patterns

Architectural and design patterns provide a pool of proven solutions to many recurring design problems [Buschmann *et al* 96]. The presentation service can make use of many of these established patterns to achieve a high level of functionality with minimum risk of complications or failure. We will be able to use the *Broker* pattern to provide services while hiding unnecessary complexity. The *Blackboard* pattern can be employed to store objects within the service that can be updated or used by multiple cooperating interfaces. The modified client-server architecture can be based on the *Client-Dispatcher-Server* pattern which protects clients from communications and protocol complexity. The challenge is to use patterns in appropriate places within the architecture and combine them into an effective whole.

Representational State Transfer

Representational State Transfer (REST) [Fielding & Taylor 00] is a design technique for web applications to exchange data based on the GET, POST, PUT, and DELETE methods of the HTTP protocol. This protocol uses data in the form of documents, and limits the influence of state [Fielding 00]. The REST design can be contrasted with the more common Remote Procedure Call (RPC) technique. A web application designed with REST will have a unique URI for each resource held, a representation (or format) of each resource, a set of methods that can create or act on a resource, and a set of status codes that could be returned. This allows application resources to be bookmarked, guards against resubmitted forms, and defines an effective API for other software to use. The

REST design technique can be used with any web application design and works with most implementation technologies.

When using a REST-based application, the calling software need only know the URI of the resource to be acted upon. The methods are the familiar GET, POST, PUT, and DELETE operations which mirror the Create, Retrieve, Update, and Delete (CRUD) operations used in database applications. To use other other styles of API (such as SOAP), the caller needs to understand the methods available, the arguments for that method, the data structure schema, the transaction pattern, and the conditions for exceptions and how they are handled. The power of the REST technique lies in its simplicity and reuse of the browser-based method of resource interaction [Bianco *et al* 07].

Frameworks

A popular definition of a framework [Johnson & Foote 88] is:

> *A framework is a reusable, semi-complete application that can be specialised to produce custom applications.*

However, in the web context, this definition should be extended to *an executable empty web software environment that can be extended to meet application requirements by additions of custom code and configuration.*

In describing object oriented frameworks, experienced designers [Fayad & Schmidt 97] identify the key properties: modularity, reuseability, extensibility, and inversion of control. These properties allow the developer to create software with higher productivity and quality. To achieve these benefits, a project built using a framework must consider a number of related issues:

- the effort in developing the framework itself,
- the learning curve of developers who are to use the framework,
- the complexity of integrating the framework with other tools, interfaces, and code,
- the problems in coordinating changes in the framework and dependent application software,

- difficulties in debugging complex software with a shared flow of control,
- the balancing of run-time efficiency and flexibility of the framework, and
- maintaining compliance to evolving internal and external standards.

An effective framework has several simple APIs for applications to use to integrate components, configuration, and custom code. The key attributes for a framework are the separation of concerns, and simplicity. The purpose of the framework is to hide the complexity of operating a style of software while automating the repetitive and common parts.

The Database Service Model

Can we learn from another software concept used to mitigate a similar set of problems? The purpose of a Relational Database Management System (RDBMS) is to hide the complexity of data management from applications, provide optimised data handling services, enable applications to share data, and support data management. It supplies this capability via an ASCII text command set (SQL) over a network socket connection to multiple simultaneous clients.

The presentation service (PS) concept mirrors the RDBMS concept. The PS hides the complexity of user interface connections and protocols, enables the sharing of sessions, enables applications to share user interface resources, and supports UI management. Multiple simultaneous applications access the PS over a network. A similar example can be found in an SMTP server which encapsulates the email communications service functionality.

The discussed design concepts are methods of managing complexity using abstractions. The presentation service concepts will be presented in the next section, where we will see how these design concepts have been folded together into a new style of software.

3.4 Synthesis of the Presentation Server

There are subtle differences between software architectures, frameworks, toolkits, libraries, products, components, and services. They are all software concepts and artifacts intended to assist a software developer in building applications with less effort and less risk; however they are used at different levels of abstraction in the development process.

A software service is one or more separately executing processes that make logically separable functionality available to multiple software clients via a publicly described protocol over a communications channel. A software service is intended to make available some shared resource or commonly required application functionality while hiding implementation details.

The presentation part of applications deals with the dynamic generation of media designed for display to humans, and the capture of human generated input device events. The media generation uses programmer-defined presentation rules to process selected raw data into user information displays, and add information entry, application navigation, and data transaction components. Generation is controlled by the types of interaction the user requires, the user profile configuration, user security permissions, and system state changes. User activity within the user interface (such as mouse gestures) result in an event being reported to the remote application with information on the *type* of event, the *identity* of the component the event was focused on, and the *identity* of the user.

To develop a separate presentation layer for web applications, we will need to explore how the browser communicates with application software. The browser initiates a transaction as a result of the user opening a web page, clicking on a button, or clicking on a hyperlinked text or image. A TCP connection is established with the server specified in the *host* part of the URL, and the HTTP protocol is used to request (GET) content or send (POST) data in key-value pair format. In both cases, content is returned to the requesting browser in MIME format that can contain references to other content expressed in URL strings (typically images, style sheets, Javascript packages, or frames). The MIME data (most often HTML, GIF, or JPG) is then displayed in the browser surface, presenting information and interaction options for the user. When the server receives a GET or POST transaction from a browser, it will try to locate the local resource described in the *path* of the URL.

If this resource is a static file, it is returned to the browser client. If the resource is active (a program, script, or module), the resource will be executed with the key-value data as input and the output stream is then redirected to the requesting browser's TCP socket. In either case, the local browser attempts to render it according to the client configuration and content handling software.

The presentation service is unusual in exposing two separate API types and acting as a value-adding broker between them. The service offers connections and adaptors for multiple user interface client types and an application interface for many simultaneous applications that are configured to use presentation services. The client and application connections may be long lasting and supporting many transactions, or transient and supporting a single transaction (such as an HTML web page GET request).

The applications API uses an inversion of control technique. The presentation service exposes a network TCP service to applications like a server; however after an application connects and registers, the transaction style inverts and the application then functions as a server and waits for user events from the presentation service. Of course, an application may have other interfaces that it responds to too (perhaps even a local host user interface). Asynchronous messages from the application may also occur due to timing logic or a change of state caused by software external to the application.

It is important to differentiate here between the concepts of a service-like application and the traditional workstation single-user program. The workstation program maintains state for only one user and may be started and stopped as required. The service-like application described here would be normally long running and support multiple simultaneous sessions. All user transactions that require this application functionality would be routed to this application, where the event is processed in the context of the relevant session, and actions performed on data stores and user interface(s) via the presentation service. This approach can offer an increase in space efficiency (the code is only loaded once and kept in one memory image), time efficiency (the code is server based and is always running), but may present problems in scalability if a single host is required to process hundreds of simultaneous user events. Server based applications need to be carefully designed if CPU or IO intensive algorithms are required in a large system.

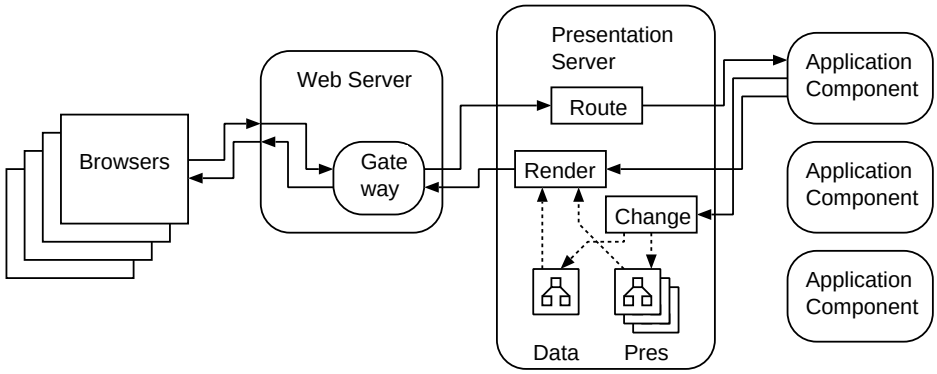

Figure 3.2: Architecture of a web application framework using a Presentation Service.

The architecture of a web application framework that employs the presentation service concept is illustrated in Figure 3.2. After registration (where an application nominates a unique name to the presentation service used for UI event routing), the service accepts presentation components and data components that determine how the user interfaces will be rendered. These components are organised in a hierarchical data structure within the presentation service and can be manipulated by a set of object oriented messages. The presentation components are template fragments for a UI command language, and use active fields to dynamically synthesise UI update streams using designated data components. There can be many presentation component types, but only one data component type.

The components in the presentation service should be easy to understand and use. Each component has a tag that reflects the type of object (such as `table`), an optional list of attributes in key-value pairs, and optional sub-components and string data. This object design simplifies conversions to and from XML, and matches the concept of an object in object oriented programming, thereby reducing the learning load on developers. Attribute values and string content can be labelled as dynamic with some simple transparent syntax so that the content will be interpreted as instructions at render time. A number of special objects (such as looping, conditional, and data selection) are also available which are acted upon by the renderer and not emitted to the UI command stream. These simple component structures and features are capable of supplying much of the user interface synthesis requirement for applications.

Components are created and manipulated inside the presentation service by commands sent from attached applications. These components may be inserted by the applications, created by inheriting existing components, changed by updating attribute values, moved between parent objects, and deleted. The presentation service does not enforce when component commands are issued, but applications would be expected to write most presentation components and foundation data components on attachment, and then update components due to state changes from user interface or other events.

When an application needs to update one or more user interfaces, the components inside the presentation service are updated and a `render` command is issued. This command identifies which presentation object sub-tree to render, which data sub-tree to make available, and which connection or session is to be sent the render streams. The presentation service then builds the user interface stream according to the presentation object structure and dynamic fields, recursing through presentation and data trees until both are exhausted. The output streams in the format compatible with the remote user interfaces are then sent to a single connection, or to all connections in a session depending on the application's command.

We have seen how presentation service components are used to build user interfaces, but how are user interface events handled? Captured user interface events from linked user interface clients are transformed into a standard format and routed to the appropriate application based on the default for that client, or a nominated application name that has been registered by an application. These events either describe a raw user interface event, request an update to a data object, or request a change to the user interface. Raw events describe an action happening to a defined user interface object, leaving the application to derive meaning using its internal state. An update request presents an object defined by key-value pairs to be used in a transforming or selecting operation based on the applications interpretation of the request type. User interface change requests use the key-value pair data to express the requested change to the application (such as presenting the next record or displaying completely new information). These events do not *require* answers to the UI through the presentation service (except for CGI types that require returned content as defined by the HTTP protocol) as the user interface to application protocol is left to the application system designer; however standard user interface patterns of user event capture leading to user interface update is provided for.

3.5 Presentation Service Functions

Given the above outline of a presentation server, how can we design it to meet the requirements of the web application developer without introducing design artefacts that would cause unnecessary new problems. One limitation of any new service will always be the new API language and transaction pattern. One of the goals in the design will be to minimise the complexity of any new language, and simplify transactions patterns. By reusing familiar syntax (such as XML, and familiar transaction styles, developer cognitive requirements will be reduced.

User's browsers attach to the web server and send through a URL, which can be viewed as a user event. An adaptor then converts the event into a PS compatible message and sends it to the PS on a network socket. The PS routes the message to a connected application via another network interface, and waits for a response on that connection.

The application is required to understand the user event message and respond with a command that causes the PS to return a user interface page to the user's browser. Business logic within the application may cause many operations before this happens, such as updates to a database, queries from a database, updating data structures within the PS, or communicating with other services, file systems, and applications. The minimum requirement for a PS compliant application is the acceptance of user event messages and transmission of user interface update commands via a network socket.

The protocol used for application interfaces is based on XML, and consists of structured messages for event description, object operations, and user interface update requests. User events that are sent to the application are descriptions of data entry, navigation directives, option selection, and UI object manipulation. The application uses object operation commands to create, delete, and modify objects with the PS as a response to user events or at other times due to other communication links or changing file system state. The application also sends user interface update messages to the PS to request changes to one or more user interfaces.

Once the PS has been given the command to return a user interface to the user by the application, it dynamically assembles the specified UI from components and data, and returns the web-standard MIME data to the requesting client. The web server then closes

that connection with the PS and the client and waits for the next request. Applications can also receive and transmit other messages independent of PS transactions at any time.

Inside the PS, user interfaces are assembled as required from presentation components and data components. These components are *objects* with types, attributes, and sub-objects, structured into a tree. Some objects are created from configuration files when the PS is started, and others are created by applications dynamically, and applications may change, copy, or delete their own objects at any time. Applications may also reference or copy another application's presentation and data objects (ie: rendering external presentation components with local data, or rendering local presentation components with external data).

To create a user interface (such as HTML), a presentation object is *serialised* in a *data context*. The data context is simply a pointer into the data object tree, and defines what data will be made available to the presentation object and sub-objects for user interface generation. The serialising object outputs initial UI content, then iteratively gives control to its sub-objects (with a possible *data sub-context*), before outputing final UI content and returning control to its encapsulating object or caller. The range of standard rendering objects are expanded with special objects which implement data context change, simple decision logic, and looping constructs. These special objects do not themselves generate UI output, but control the rendering of sub-objects and content. In addition, raw content and attribute values can have *active* content, which is computed at render time using the current data context.

This transaction pattern and UI generation method is not only effective at generating HTML – the primary web application output language, but any markup language. The concept can be extended to generate any output content by defining a serialisation method for the new presentation objects. This rendering method could equally well build dynamic PNG images, VRML worlds, or RTF documents.

The PS can serve many simultaneous UI connections with embedded child components, and many applications, while sharing presentation and data objects, managing multi-connection sessions, and exposing a management interface for dynamic status, usage statistics, and control. As the interfaces to the PS are ASCII and/or simple XML over network sockets, any UI technology and any application on any platform written in any language will be able to use the PS.

3.6 Architecture Features

The presentation service architecture is designed around a number of strong software engineering concepts. These concepts are employed in many parts of the software engineering world, but have been slow to penetrate the web application development environment. Many web application technologies, techniques, frameworks, libraries, and toolkits employ *some* of these engineering concepts but fail to consider all the facets of the web application environment. The presentation service is a combination of the best of contemporary software engineering concepts, adapted to the web application environment, and designed for simplicity and effectiveness.

Hides UI syntax: The encapsulation of UI complexity minimises developer learning requirements. Having less complexity to remember reduces the cognitive loading of the developer, and allows his or her mental efforts to be expended on unique problems of the web application. The delegation of the UI "problem" to UI experts also minimises the frequency of errors in UI generation functions. Common UI functions are available for use without the need for these functions to be rebuilt in every web application project, saving the developer time. Developers are shielded from protocol and UI device diversity and upgrades, as the dependent UI functionality is managed by the presentation service.

Separate long running server process: A long running process spends less time loading, initialising, allocating memory, and connecting to other services. This is more efficient than restarting processes for each user event, session change, or service call. A single service handling multiple clients will also use less memory than multiple processes with one client each. There is also a benefit in user response time, because the responder does not have to load and initialise before answering the request. A long running process can support collaborative applications, as user information can be shared without complex inter-process protocols.

A single long running service does have some limitations. If the software fails, all user connections will be lost. The service will be a single point of failure. A successful service will use internal exception management, guard against memory leaks, and validate incoming data structures and values.

As a single service manages all user connections in a single process, load sharing over multiple processors and multiple hosts is difficult to manage. Scalability will be a problem as the concurrent user count grows, particularly if transactions are expensive in CPU time, memory, or network bandwidth.

Dynamic update of the content and applications: By designing the service so that applications and users may connect and disconnect dynamically, flexible architectures can be achieved. If the update of service content, the user events, and the application messages are designed to be asynchronous, user interface responsiveness will be improved, and applications become easier to write. Dynamic updates also assist rapid application development methodologies. New content can be inserted by programs, and tested in the user interface without halting the service.

Minimum thin client requirement: By using standards such as HTML and CSS to build the user interface, clients can use standard browsers without configuration changes, large downloads, new software, or changes in operating system. Using a standard browser as a thin client frees system managers from maintaining software on client machines.

Hierarchical object trees: Programmers are familiar with hierarchy trees for information organisation (eg: file systems and organisation structure). This structure is also a perfect match for XML elements and sub-elements. Trees are good for partitioning data and managing access control.

Object oriented: Object orientation is accepted as a strong engineering technique to manage complexity, and can easily represent real world concepts. The OO technique emphasises re-use. The re-use advantages in the presentation service are:

- Consistency of UI
- Productivity
- Reduced testing
- Greater stability (pre-tested components)

Platform independent: This independence to hardware and operating systems means the service will be installable on diverse machines within organisations. It also makes the service immune to hardware, OS, or network migrations and updates.

Language independent: Applications can be developed in any language with XML and TCP socket support. This allows developers to use the most familiar and efficient language. Specialist languages can also be used for special purposes and specialist libraries can be used with its required language.

Multi-user sessions: In the presentation service, multiple users may appear as one user for simpler application design. Actions in one UI may transfer to all other UIs in the same session, implementing collaborative sessions. Users joining a session can have their UI updated to the current cumulative state.

Multiple transport protocols: A range of adaptors can be used with the UI interface side of the presentation service, allowing many protocols to be connected and used (such as HTTP, WTP from mobile phones, email via POP3/SMTP, and strings over TCP sockets).

Multiple dynamic content protocols: The content types that can be supported over these transport methods can be extended too. As well as HTML and XML, markup languages such as SVG, VRML, RSS, WML, and SOAP could be used. With the appropriate media generators, other media content such as Flash animation, Sound, Voice, PDF, CSV, and CSS could be managed and rendered.

XML API protocol: XML has the benefit that it is human readable for ease of learning and ease of debugging. It is also is a universal standard and many tools and languages support it. The separation of syntax from semantics protects the service from change or errors. Unwanted, new, or optional attributes and elements can be ignored.

System Integration: The service design uses URL based calls returning dynamic MIME content, which is supported by most software. This interface can be used by a browser, or another application. It forms an application interface for connecting other applications. The application interface can connect to any number of small and large applications. This combination of connection options provides flexible facilities for integration.

Powerful object operations: The service implements add, delete, move, copy, and object change. Objects are chosen using an XPath search, which selects objects by combinations of the select operators:

- Tag type
- Element attribute existence
- Element attribute value
- String content
- Element content
- Numerical index or range

The objects support a method of prototype based inheritance, and use overloading of object attributes to specialise objects. Dynamic binding of data structures to presentation objects when rendered creates the dynamic user interface.

Separation of concerns: Graphic designers can change the UI without programming skills, and load the new objects into the PS. Developers use the UI and session support to simplify applications. System administrators use the control interface to monitor and manage systems, and system integrators can link other applications to PS systems without code changes.

There are some intrinsic limitations to the use of XML as an API protocol:

- Attributes cannot have compound values (but references allowed).
- White space may not be preserved.
- Serialisation and parsing is slow.
- HTML entities are poorly supported in XML representation.

These architectural features match many of the desired web application properties shown in Figure 1.1. The presentation service supports many of the required functions and attributes in web application designs.

3.7 Summary

The presentation service removes the common user interface handling code into a separate service, which is optimised for this work. The concept is familiar within the domain of data management, but the employment of this concept within the user interface domain is innovative and delivers design improvements. If applied to web application design methodologies and frameworks, this combination of engineering concepts will assist with productivity, integration, management, user interface quality, and reliability.

Chapter 4

Interactive Graphics for the Browser

4.1 Introduction

The requirements of web applications are becoming more complex but the software designer has very few tools to assist in development. The HTML specification has no support for graphical displays such as maps, diagrams, and graphs so the software designer is forced to use custom specialised software or inadequate parts of the HTML protocol to implement this part of the web application user interface.

This chapter introduces the Generic Graphics Applet (GGA) which is intended to be a flexible general-purpose interactive graphics component. The GGA manages a large collection of images, vector objects, and text. These graphics primitives are organised into layers and groups, allowing complex interactive displays to be created on the applet surface. It can be used to build interactive maps, diagrams, and business charts into web applications.

4.2 The Browser Presentation Environment

As web applications become more complex, the software designer is faced with the difficulty of building an application user interface (UI) within the limits of web protocols and

technology. Many varieties of software platforms have emerged to ease web application development [Fraternali 98], however little has been achieved to bring the interactive display properties currently available on the desktop into the browser based UI. The user of the browser UI suffers delayed responses, lack of interface feedback, limited widget range, limited layout options, and frequent screen refreshes to transmit and receive state information from the server [Rees 97].

The web UI designer is limited to "whole of screen" transactions by the page-oriented nature of the HTTP protocol. Interactive interfaces that support desirable features such as fast feedback, direct manipulation, and graphical visualisation methods [Shneiderman 97] cannot be supported without bypassing the web protocols with complex custom software such as browser plug-ins or special Java applets. Such custom software is typically built for a particular purpose and uses a proprietary protocol to communicate with a remote server, causing problems with integration, interoperability, and reuse of software.

Desktop UIs use a familiar set of interactive display tools that are consistent in the objects and relationships they show, and the behaviour they exhibit. For the purposes of this work, the eight common displays of text, table, image, form, outline, chart[1], map, and diagram will be used.

These fundamental components of displays are described in Table 4.1 with a summary of the HTML supporting elements. These theoretical components have a number of common properties:

- Each of the displays can be embedded in others of the same or different type. For example, a table may contain text and images, a diagram may contain charts or tables, and form may contain a sub-form or a map for location choice.
- The presentation of the data is separate from the content so that different displays can be used to provide alternative views of the same data. For example, an aircraft flight plan could be viewed in the geospatial context with a map, in data format with a table, as a diagram showing temporal inter-relationships between entries, or a hierarchical format organised by flights segments and subordinate waypoints with an outline display.

[1] In order to reduce confusion about the difference between a business graph and a nodes-and-edges graph, the terms *diagram* and *chart* are used within this chapter.

Table 4.1: Application Display Types.

Display Type	**Definition**	**Examples**	**HTML Support**
Text	A linear word or object stream	Memo, Email, Letter, Comment	b, i, p, pre, sub, sup, cite, etc.
Table	A rectangular array of objects	Train Timetable, Calendar	table, tr, td, th
Image	A sampled projection of an object or space	Photo, Sketch, Scan	img, map, area
Form	A set of name and attribute fields	Address, Login, Data entry	form, input, select, textarea
Outline	A vertical indented hierarchy	Table of Contents, File Manager	ul, ol, li, dl, dd, dt
Chart	A graphical representation of quantities	Bar, Pie, Scatter plots, Histograms	None
Map	A proportional spatial representation	GIS, Building, Plan, Aerial photo	None
Diagram	A graph of nodes and edges	Block diagram, Trouble-shooting diagram, Pert chart, Flowchart	None

- Components within the display can be linked to perform operations on other displays or display components. For example, a file manager application may be configured so that a directory entry in a hierarchical outline is linked to an associated table that shows the contents of the directory.
- The attributes that define the *appearance* of the component (such as line thickness, font name, or column width) are independent of the data attributes.
- Each component contains a structure of smaller components. A table contains a heading and rows. An outline contains a list of indented elements. A diagram contains nodes and edges. An image component can areas of high resolution sub-images and have highlight areas with annotation.

Other display types such as movie, 3D environment, graphical control, and dialog box do not share these common properties. These display types are not directly considered in this work; however, they can easily be connected to a web application without being fully integrated into the interactive display set. Emerging 3D environments show potential for bidirectional interlinking and customisation, however they have yet to mature.

Of the eight displays, only five can be implemented directly with components available in

HTML [Ishikawa *et al* 07]. The chart, map, and diagram types require additional client-side or server-side development to be part of the browser UI.

4.3 Current Techniques

The three most common methods currently in use for generating map, chart, and diagram displays in the browser are :

- Configuring a program on the web server to generate the graphic content, then write the result to an image file which is sent to the browser. To provide the interactive behaviour, Javascript, style sheet overlays, image tiles, or image maps are used. This complex solution is often inefficient (transferring large raster images often takes longer than a vector data set containing the same information), and does not provide immediate UI feedback and direct manipulation behaviour.

- Creation of a custom applet or other browser component which is intended for that single display purpose and must be changed to fulfil a related task. The applet is usually not provided with an application programmer's interface (API) with which it can be integrated into a new application. Further, the protocol it uses to communicate with the server is rarely made public which stops web application developers from using the applet with their own data sources and applications.

- The development of Javascript code that dynamically builds a graphic display inside the web page. Some highly functional and complex solutions have been developed using this style (such as maps with GoogleMaps [Goo08] and charts with PlotKit [Plo06]), but these efforts use extensive code and many "browser tricks" to perform graphical functions. The result is a satisfactory user interface but developers have problems with extending functionality, integration to other components, flexibility, and maintainability.

An approach to rendering *general purpose* graphics to the browser was developed into the High Performance JavaScript Vector Graphics Library [zor06]. The developer of this library describes his approach to the problem:

In HTML there are no such elements as oblique lines, circles, ellipses or other non-rectangularly bounded elements available. For a workaround, pixels might be painted by creating small background-coloured layers (DIV elements), and arranging these to the desired pattern.

This library allows developers to use Javascript functions to draw coloured vector graphics in any container on the HTML page. Extra programming is needed though to build the raw vector objects into charts, map, and diagram displays.

```
 1: <script type="text/javascript">
 2:   <!–
 3:   function myDrawFunction()
 4:   {
 5:     jg_doc.setColor("#00ff00"); // green
 6:     jg_doc.fillEllipse(100, 200, 100, 180); // co-ordinates related to the document
 7:     jg_doc.setColor("maroon");
 8:     jg_doc.drawPolyline(new Array(50, 10, 120), new Array(10, 50, 70));
 9:     jg_doc.paint(); // draws, in this case, directly into the document

10:     jg.setColor("#ff0000"); // red
11:     jg.drawLine(10, 113, 220, 55); // co-ordinates related to "myCanvas"
12:     jg.setColor("#0000ff"); // blue
13:     jg.fillRect(110, 120, 30, 60);
14:     jg.paint();

15:     jg2.setColor("#0000ff"); // blue
16:     jg2.drawEllipse(10, 50, 30, 100);
17:     jg2.drawRect(400, 10, 100, 50);
18:     jg2.paint();
19:   }

20:   var jg_doc = new jsGraphics(); // draw directly into document
21:   var jg = new jsGraphics("myCanvas");
22:   var jg2 = new jsGraphics("anotherCanvas");

23:   myDrawFunction();

24:   //–>
25:   </script>
```

Figure 4.1: Example of usage of the Vector Graphics Library (from [zor06]).

An example of the use of this vector library is shown in Figure 4.1 where standard calls to graphics functions mask the poor browser graphics support. The visible vector objects consist of hundreds of overlaid colour `<DIV>` elements carefully placed to form the required

shape. These objects are static once rendered, cannot be moved or adjusted, and there is currently no practical way to capture mouse events on these objects.

Current techniques for implementing interactive graphics in the browser environment fall far short of the capability available to desktop application developers. The next section describes an innovative solution to the interactive web display of maps, diagrams, and charts which is called the Generic Graphics Applet (GGA).

4.4 The Generic Graphics Applet

4.4.1 Requirements

The design of the GGA [Sweeney 00b] must provide a flexible light-weight web page component that can be used to display a broad range of interactive displays. The display will be composed of combinations of graphics primitives which are placed on an arbitrarily large canvas.

A number of design attributes and features were identified that would need to be implemented to achieve the aim of the Generic Graphics Applet. These attributes were chosen as they individually and collectively enhance many of the key web application properties displayed in Figure 1.1. These attributes are:

- Small, flexible and simple,
- Human readable (non-binary) APIs,
- Interactive with event reporting to server application,
- Full range of 2D graphics primitives,
- All objects support targeted hyperlinks,
- Web page integration API and web standards compliant,
- Configurable through embedded commands in the applet element,
- Support flexible group assignment and usage,
- Built as layers with commands to manipulate order,
- Object identifiers to be server addressable,

- Draggable option with drag and drop messages,
- Fixed display objects for user controls, and
- Flexibility through attributes and configurable defaults.

The interactive requirement of the design will require a communication channel to an application. The application will be hosted on the web server or a separate server and respond to dynamically attaching GGA components. This application will build graphics displays by sending a rendering command to the GGA. User actions to objects within the GGA should be reported to the application so applications can implement business logic, possibly providing user feedback by updating UI graphics.

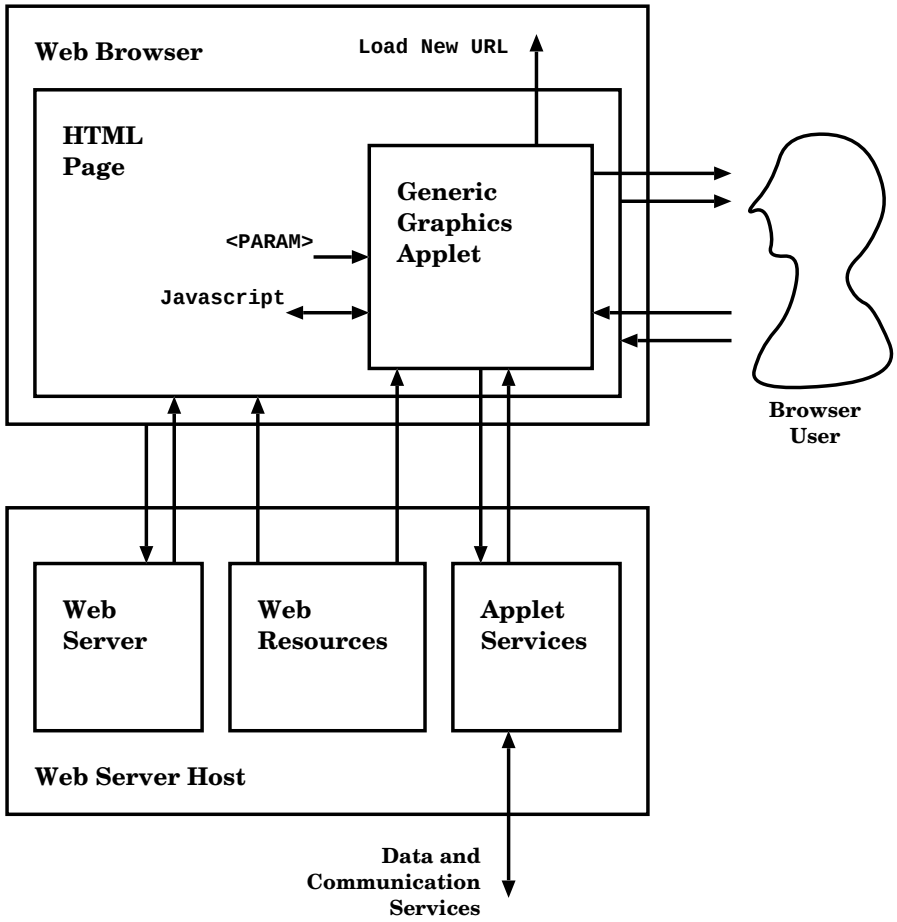

Figure 4.2: GGA Architecture Diagram

Figure 4.2 shows the architecture of a GGA-based web application. The GGA is a component in the browser user interface that presents graphical information to the user and captures user actions on graphics objects. Graphics commands can also be embedded in the page in `PARAM` elements, which are executed directly after the applet completes load-

ing. To provide web page integration capabilities, the GGA exposes a number of internal variables and two methods to DHTML Javascript code in the browser. The GGA will also have the ability to issue a URL request and have the response display in another window in the browser.

The useability of the GGA is a prime design parameter. To ensure adequate response time for user operations and information display, optimisations such as simultaneous image loading (with threads), local image caching, fast internal data structures, and block command transfers should be used. Responsive panning and object dragging will enhance the direct manipulation experience, even if the connection to the server is slow.

4.4.2 Implementation

The GGA was developed in Sun's Java 1.5 but only uses features available in Java 1.2 [Gosling & McGilton 96] to be compatible with the majority of browsers without any required extensions or plug-ins. It is deployed in a single JAR file, and is less than 100kB in size. The applet is rapidly downloaded and initialised in web pages, creating a minimum of delay in the browser user interface. Once downloaded, further use fetches the file from the browser cache, reducing load delay to zero.

Every object created in the GGA is assigned a layer. The applet by default will load each successively defined object over the existing objects; however, the layer ordering can be changed at any time with the `change layer` command. Layers are a core concept in the GGA: they determine what objects can be seen and which objects are triggered by a mouse click.

The GGA component uses three communication methods, which may be used separately or in combinations. The GGA can build graphics displays from embedded commands inside *param* elements inside the applet's *object* element. Another option is the use of events on the web page to control the graphics content of the GGA using Javascript (also known as ECMAscript [ECM97]). The Javascript can be attached to many types of event handlers supported by the web page model (such as the `onClick()` attribute of many elements). The most powerful method is a TCP socket connection to an application on a remote server. A simple plain-language protocol is used to send and receive messages from the

application. The remote application is able to build graphics displays with named objects, and respond to user mouse actions on these objects. The names used in these objects are separate from XML Id names and named elements in the surrounding DHTML.

4.4.3 Command Language

The requirement for simplicity and readability drove the API choice toward a structured English style dialogue. The use of line-feed terminated command and event strings was used over a standard TCP connection for communication between the GGA and the server. This method was chosen over RMI [RMI97] and CORBA [Obj95] as the target program only needed a standard network library and the ability to handle strings to function as a GGA server. Any language from Cobol to Basic to TclTk to Smalltalk to Perl could be used to implement a server application. The increased flexibility of the strings over TCP solution allowed flexible parameter types as they are transferred in an ASCII string representation. The human readable format also promoted ease of learning and debugging. A complication arising from this choice was the need for a sophisticated language parser in the GGA; however this development cost would only be paid once, allowing many future web application developers to make use of the clear and simple syntax.

```
load [draggable] [fixed] <url_img> at <x,y>
   using <id> [<notify>] [<urllink>]
draw [draggable] [fixed] [<prop>] line <x,y> to <x,y>
   using <id> [<notify>] [<urllink>]
draw [draggable] [fixed] [<prop>] circle at <x,y> radius <r>
   using <id> [<notify>] [<urllink>]
draw [draggable] [fixed] [<prop>] rectangle <x,y> to <x,y>
   using <id> [<notify>] [<urllink>]
draw [draggable] [fixed] [closed] [<prop>] polygon <point_set>|<delta_set>
   using <id> [<notify>] [<urllink>]
   <point_set> ::= 3{<x,y>}
   <delta_set> ::= at <x,y> 3{<dx,dy>}
write [draggable] [fixed] [<prop>] text <string> [with <fontspec>] at <x,y>
   using <id> [<notify>] [<urllink>]
draw [draggable] [fixed] [<prop>] ellipse at <x,y> major <w> [minor <h>]
   using <id> [<notify>] [<urllink>]


where:
   <notify> ::= notify [click] [doubleclick] [grab] [drag] [drop]
   <urllink> ::= add link <url> [into (self|blank|parent|top|<name>)]
```

Figure 4.3: Specification of GGA object creation commands.

The object-creation parts of the language are shown in Figure 4.3. An example of the GGA command language is shown in Figure 4.13 in Section 4.4.6. These commands instruct the GGA to load and display an image, draw a shape, or render some text. The `x` and `y` values specify the place to put the object on the canvas. The applet starts with position `(0,0)` of the canvas in the top left corner of the GGA display. The `fixed` keyword attaches the object to the applet's surface rather that the canvas. This means the object will not move when the canvas is scrolled or panned; this is very useful for creating UI controls. The `draggable` keyword indicates that the object may be picked up with a mouse click and moved to a new location by the user. Each object must have a unique ID name specified when it is created. This name is used for reporting events on the object, and identifying the object when an operation on it is requested. This name is unrelated to the Id name of XML elements used in BUS transactions or the names of elements in the DHTML environment.

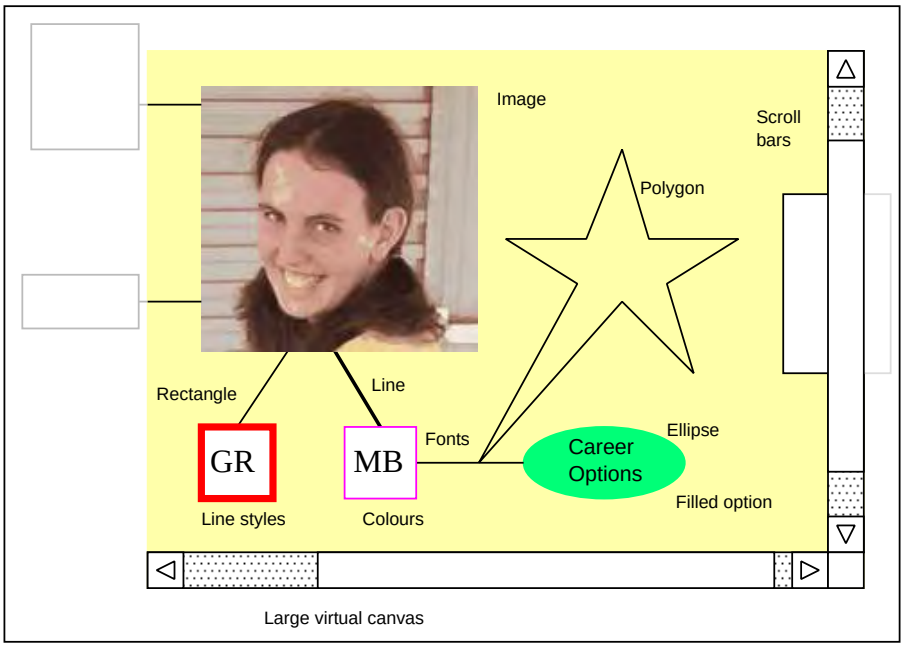

Figure 4.4: Objects created and manipulated inside the GGA.

The types of objects and attributes are shown in Figure 4.4. The objects are shown within a scrollable viewport on a large virtual canvas. Each object maintains its own identity and can be manipulated by the server in groups or independently.

Other properties such as linecolour, linewidth, fillcolour, and flag for filled shapes may be specified. Bounds on how far an object can be dragged is specified with xmin, xmax, ymin, and ymax. Each object can be configured to report on the user mouse events that

occur such as click, doubleclick, grab, drag, and drop. Additionally, each object may have a nominated URL that is opened when this object is doubleclicked by the user. Text objects have specialised properties such as style, fontsize, fontname, and justification settings.

```
set attribute 1{ linewidth <int> | linecolour <color> |
   fillcolour <color> | filled <bool> |
   xmin <int> | ymin <int> | xmax <int> | ymax <int> }
set font [ style <styleopt> ] [ fontsize <int> ] [ fontname <nameopt> ]
   [ justify left | right | centre ]
set notify ( all <bool> ) | 1{ <action> <bool> }
set outline shape rectangle | line | circle | ellipse | freehand | text


where:
   <styleopt> ::= plain | ( [bold] [italic] )
   <nameopt> ::= serif | sansserif | monospaced | dialog | dialoginput
   <action> ::= click | doubleclick | grab | drag | drop | annotation
   <bool> ::= true | false
```

Figure 4.5: Specification of GGA default setting commands.

At any time, a command to change object defaults can be received. These `set` commands change the default properties that will apply to new created objects. In Figure 4.5, we see the range of default properties that can be set. The `set outline shape` command is special as it does not change defaults, but changes the behaviour of mouse gestures. The outline mode can be set to a rectangle, line, circle, ellipse, freehand, or text. The text mode is unique in allowing the user to type in a word or phrase *into the mouse pointer*, then every click can report this "text annotation" on this object to the application.

```
group [ with drag ] <id> is 1{<id>}
ungroup <id>
remove <id> from group <id>
add <id> to group <id>
```

Figure 4.6: Specification of GGA group commands.

The group capabilities (see Figure 4.6) allow an object to be a member of one or more (possibly overlapping) groups. Actions resulting from a command with a group name in the ID field affect every member of the group. A special group name *all_objects* is always available and includes every object in the applet. The `with drag` modifier causes all objects in a group to be dragged if one of the objects is dragged. This feature is very useful for building compound objects.

```
remove <id>
show <id> [with <alpha> visibility]
hide <id>
move <id> (to|by) <x,y>
change <id> layer ( to top | to bottom | up | down |
   to above <id> | to below <id> )
```

Figure 4.7: Specification of GGA object manipulation commands.

A command can be sent to manipulate existing objects by addressing the change operation to the object's ID name (see Figure 4.7). Changes to a group ID name will affect all members of the group. With these object manipulation commands, objects can be destroyed, moved, and hidden. The display layer order of an object can also be changed in absolute terms or relative to another object. Objects may also have a transparency property changed which will give the object a see-through effect.

```
pan (n|ne|e|se|s|sw|w|nw) by ( <int>% | <int> pixels )
center on <id> [at <x,y>]
loadURL <url> [ into (self|blank|parent|top|<name>) ]
```

Figure 4.8: Specification of GGA set view commands.

In Figure 4.8, the `pan` and `center` commands that alter the user's view are specified. The `pan` command moves the applet's view port in one of eight directions by the specified amount. The `center`[2] command scrolls the viewport so that the object with the requested ID name is in the centre of the view. The `loadURL` command is a powerful feature that sends a URL request to a web server and returns the result in another frame or browser window. Like other commands, `loadURL` can originate from the applet's parameters, Javascript via the `gga.cmd()` method, and the application via the communications socket. This flexible method of calling web URLs is a powerful integration feature for dynamic asynchronous web application designs.

4.4.4 GGA Generated Messages

The GGA can emit messages through the application socket. These messages are typically reports on user actions within the applet so that the application can trigger business logic

[2] As is common in the computer science world, the American spelling for words often "leak" into British and Australian software.

and change state. This event reporting mechanism is an important part of the GGA's interactivity support functions.

```
object <id> had <user_action> at <x,y> [ text "<string>" ]
   user_action ::= click | doubleclick | grab | drag |
      drop on <tgt_id> |
      anchor | stretch | pin | annotation
message "<msg>"
invocation <methodstring>
ERROR: <msg_string> failed because <reason>
```

Figure 4.9: Specification of GGA event messages.

There are four types of messages (see Figure 4.9) that the GGA can send to an application. The object event message tells the application that the user performed a mouse operation on the object with the ID name in the message. The logic within the application will determine the meaning of the event from its knowledge of what that object represents and the current internal status. The `text` clause will be added to events that originate from a user *annotating* an object with a word or phrase.

The `message` type of message typically originates from the surrounding DHTML via the `gga.sendmsg()` method or from an embedded applet parameter command that uses the `sendmsg "<string>"` command. The string will have a regular structure that the designer of the application has defined, so that an understanding between the message sender and message receiver is assured. An example would be the use of a DHTML button to send a verb-noun pair such as "display aircraft" or "zoom 200%".

In order to maintain internal state, the application is sent an `invocation` message each time the browser environment sends a command to the applet via the `gga.cmd()` method. Errors in GGA processing of commands are sent to the application in an error message. It is the responsibility of the application to take action on these errors or write them to a log.

There is a `sendmsg "<string>"` command too; however it is not useful for the application (as it would just send a message back to itself), or in the DHTML code (the `gga.sendmsg()` method is intended for this purpose). The `sendmsg "<string>"` is designed to be used in the parameter commands to communicate context from the web page to the application.

4.4.5 Embedding the GGA in a Web Page

The GGA is a standard applet and is used in a web page by specifying the applet attributes in a HTML *object* tag. The GGA, like other applets, occupies a rectangular space in the web page, obeys web page content flow rules, and can be addressed and manipulated via Javascript using the Document Object Model (DOM).

Figure 4.10 shows the typical method of embedding the GGA applet in the web page. All parameter entries are optional. The TCP port that the applet is to connect to is given in the `PORT` parameter tag in the applet specification. A set of command strings can also be given in the parameter tags. These parameters are executed in order, computed by the integer as part of the parameter name (eg: CMD1, CMD5, CMD6, then CMD12). Note that command numbers can be skipped — this allows the insertion of other commands later. Parameters can set graphics defaults, create graphics objects, manipulate graphics objects, and send message strings to the server application.

```
<object name=GGA
  classid=URL
  codebase=/hiat/applet/gga
  archive=gga.jar
  code=dsto.himalaya.HimalayaApplet.class
  align=baseline|center|left|middle|right
  width=INT height=INT hspace=INT vspace=INT
  >
  <param name=PORT Value=11907/>
  <param name=HORIZONTALSCROLLBAR value=true/>
  <param name=VERTICALSCROLLBAR value=true/>
  <param name=DEBUG_EVENTS value=true/>
  <param name=DEBUG_SOCKET_IN value=false/>


  <param name="CMD1" value=GGACMD+";"+INT/>
  <param name="CMD2" value=GGACMD+";"+INT/>
  <param name="CMDn" value=GGACMD+";"+INT/>


</object>
```

Figure 4.10: Specification of GGA applet element options.

4.4.6 Demonstration

Several demonstrations of the application of the GGA to support web applications are described below.

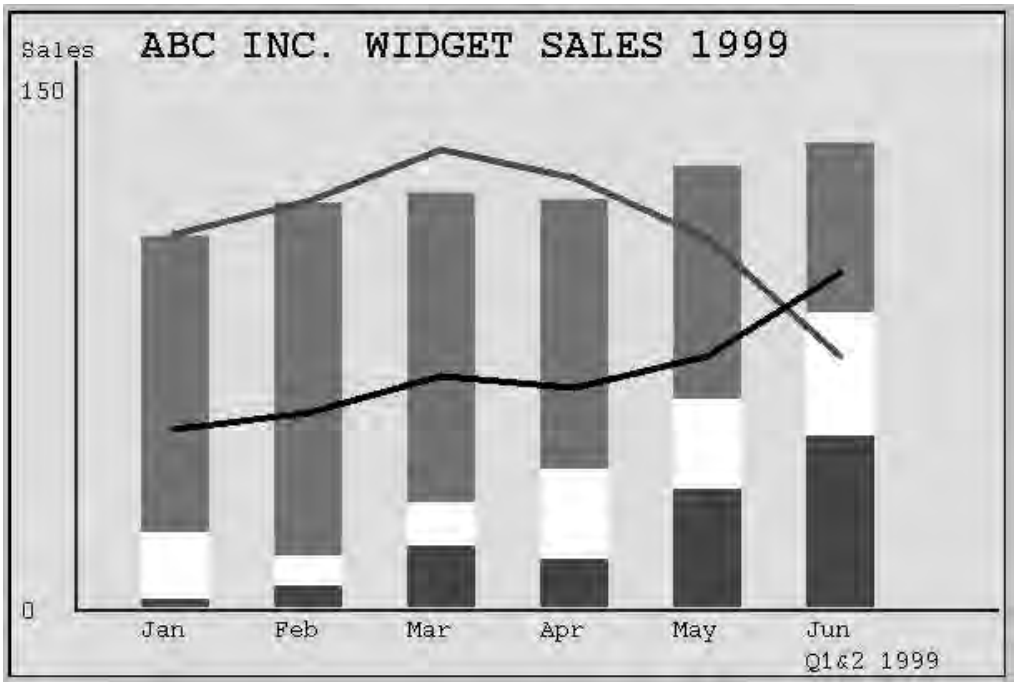

Figure 4.11: Business Chart Display Example

In Figure 4.11, a simple business chart has been generated by a server application using the GGA. The server application has produced the surrounding dynamic HTML page and configured the embedded applet to connect to itself when loaded. The application uses combinations of dynamic web pages to display standard HTML content, and the GGA to supply an interactive user interface for charts and other graphical content.

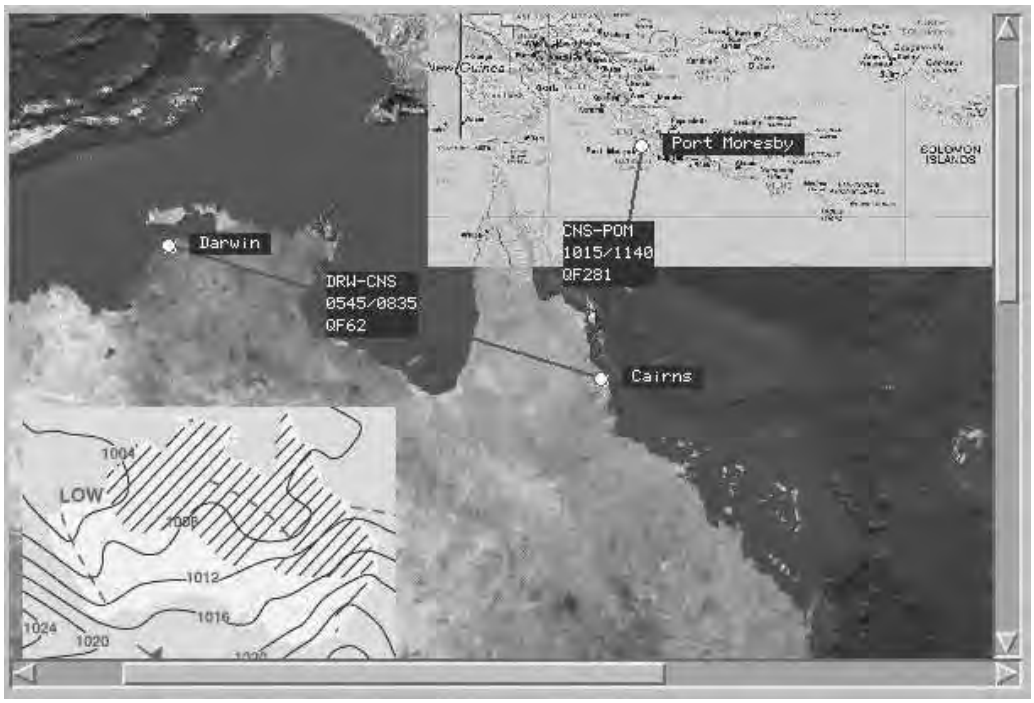

Figure 4.12: Mapping Display Example

The mapping display example in Figure 4.12 shows the user interface of a simple travel planning application based on the GGA. The application has built a map background by specifying two map images and added a weather map image. The map images have been scaled to be compatible with a single resolution, and metadata provides information on where each map should be placed to join correctly with other maps. The GGA adds the maps at the correct pixel locations, using layer controls to ensure smaller higher resolution maps appear on top of larger low resolution maps.

Figure 4.13 shows a snapshot of the GGA commands that were used to build the map example in Figure 4.12. Local image URLs were used in this instance but remote URLs fetched through a local HTTP proxy work equally well.

```
1:  load /hiat/map/world/map30s150e60deg1024pix.jpg at 0,0 using map1
2:  load draggable /hiat/map/world/png_small2.jpg at 340,17 using map2
3:  load /hiat/map/weather.gif at 100,310 using map3
4:  set attribute linecolour white filled true fillcolour 333333 linewidth 1
5:  draw linecolour red linewidth 2 line 442,293 to 466,155 using line1
6:  write linewidth 0 text "CNS-POM\n1015/1140\nQF281 " at 420,200 using t1
7:  draw linecolour red linewidth 2 line 186,214 to 442,293 using line2
8:  write linewidth 0 text "DRW-CNS\n0545/0835\nQF62 " at 280,230 using t2
9:  draw linecolour red fillcolour white circle at 442,293 radius 4 using dot1
10: write linewidth 0 text " Cairns " at 455,286 using city1
11: draw linecolour red fillcolour white circle at 466,155 radius 4 using dot2
12: write linewidth 0 text " Port Moresby " at 479,148 using city2
13: draw linecolour red fillcolour white circle at 186,214 radius 4 using dot3
14: write linewidth 0 text " Darwin " at 199,207 using city3
```

Figure 4.13: Example GGA Commands

The lines, text, and city circles are placed interactively by the server in reaction to events generated by user mouse actions. Other events were also generated by messages initiated by Javascript enabled buttons in the surrounding HTML page. Javascript messages to a GGA instance can originate from many DHTML sources such as buttons, hyperlinks, icons, timed calls, images, and the page itself.

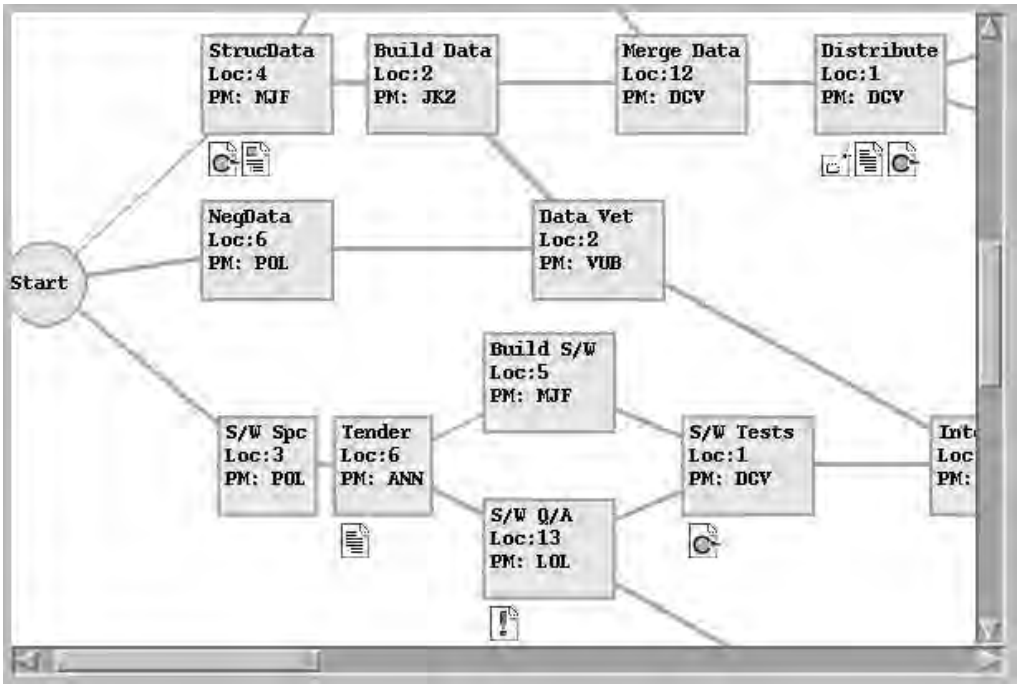

Figure 4.14: Graph Display Example

Many information displays consist of a layout of vertices and edges. These include organisation charts, trouble-shooting charts, flowcharts, flow diagrams, block diagrams, schematics, and network diagrams. The Graph Display Example in Figure 4.14 shows how a display of vertices and edges can be implemented using the GGA.

In this diagram example, the application initially draws the network of edges between pairs of node locations. Then, for each node, the application sends commands to build the node representation. A filled rectangle (or a circle depending on node properties) is placed at the node location. The text associated for each node is placed over the rectangle and a number of small image icons is then placed beneath the node rectangle. The shape, text, and icons of each node are linked with a drag-group association so they are dragged and addressed as a single logical object.

The supplied interactive options in this example are a sample of the interactivity that can be achieved with the GGA: The nodes can be dragged to another location so the user can optimise the graph layout. A double click on a node shape will open a browser form containing the node data for editing. A double click on a node icon will open a different browser window using the URL from the icon attribute. The user can pan across the diagram using the built-in scrollbars. A *Start* button in the web page sends a command to the GGA to centre the display on the Start node. The user can select a number of

nodes to list in a table by outlining the nodes with a bounding box and clicking on the *Display Selection* button in the web page. These interactive aspects of the user interface are described in table 4.2.

Table 4.2: Interactive properties of the Graph Display example.

Object	**Mouse Action**	**Application Semantics**	**Application Behaviour**
Node shape	Double click	Open node data in browser window for editing	When the data is saved with the *submit* button, all GGA displays in the session are transparently updated.
Node icon	Double click	Open URL in named window	This is a native GGA feature. The application only assigns a URL to an object. It is not involved in opening the resource.
HTML button	Click	Send message "go to Start"	The application responds to the "go to Start" message by sending a `center on Start` to the originating client.
Scroll bar	Click or drag	Pan this application view	The server does not control this action. Each user may view a different part of the diagram.
Display	Drag out bounding box	Select nodes	The application finds the nodes within the box and stores the *selection* linked with this client.
HTML *Display Selection* button	Click	Send message "display selection"	The application will send a `loadURL` request to the client GGA to load a browser window with the *selection* of nodes in this client browser.

These interactive properties are not implemented in the application with hundreds of lines of code. They are directly supported by the GGA design.

4.5 Technology Evaluation

To evaluate how the GGA performs, we check which properties of the web application environment would been enhanced by the use of the GGA features (see Table 4.3). The GGA contributes strongly to simplicity due to its straight-forward APIs and clear encapsulation of graphics functions. It is also strong in integration properties with its multiple APIs and URL support, with independence from platform and language. It's user interface properties are probably GGA's strongest features, so multi-media and interactivity are significantly enhanced.

Table 4.3: Evaluation of GGA Web Application Properties.

Property	**Enhance**	**Features**
Sep. of Concerns	$\surd$	Separate DHTML, applic, GGA dev domains
Simplicity	$\surd\surd$	Simple language, events, DHTML API, UI
Modularity	$\surd$	Reusable light component
Familiarity	$\surd$	Drag and drop, graphics
Learnability	$\surd$	Direct manipulation
Useability	$\surd$	Better understanding of data using graphics
Consistency		
Orthogonality		
Interactivity	$\surd\surd$	Asynchronous protocol
Multi-media	$\surd\surd$	Graphic display
Continuity		
Flexibility	$\surd$	General purpose graphics
Customisability		
Productivity		
Configurability		
Collaborative	$\surd$	Application can connect multiple GGAs
Adaptability	$\surd$	Adaptable to any graphics purpose
Clarity	$\surd$	Readable protocol language
Reuse		
Inheritance	$\sim$	Objects inherit from defaults
Encapsulation	$\surd\surd$	Graphics handling layer
Lang. Independence	$\surd\surd$	Any application language
Interoperability	$\surd\surd$	Connects through APIs, URLs
Multiple UI Devices		
Open Standards	$\sim$	DHTML, TCP socket
Multiple APIs	$\surd\surd$	3 APIs
Thin Client	$\surd$	Only standard browser needed
Applicability	$\surd$	General purpose
Scalability		
Plat. Independence	$\surd\surd$	Any modern browser without change
Efficiency	$\surd$	Cached, client CPU, light protocol
Maintainability		
Instrumentation		
Manageability	$\surd$	Deployed as standard web resource
Confidentiality		
Integrity		
Availability		

Legend: $\sim$ = partial enhancement, $\surd$ = enhancement, $\surd\surd$ = major enhancement.

The GGA design has a number of intrinsic limitations but none are unexpected or onerous. The command language is another thing for the developer to learn. Its clear readable syntax should make this task easy, but developers will need to be convinced of the net productivity gain in learning to use this new technology. Applets are typically restricted from connecting sockets to any host except its originating web server. This forces the

GGA applications to be deployed to the web server unless security settings are modified on the clients (which violates the run-anywhere without client change requirement). A socket proxy on the web server would avoid this problem at a cost in performance and reliability.

There are also a number of unsolved GGA implementation problems which could be expected to be solved before applying this technology in a production environment. It is unclear under what conditions an applet terminates. Applets seem to *hibernate* when the web page it has executed in is replaced by another web page, only to spring back to life when the `back` button is pressed. Hundreds of hibernated applets inside browsers due to visiting many GGA-equipped pages will have serious memory demands. There are problems with loading large images in applets. This may be caused by an over-zealous Java security manager or a limitation on the JVM memory allocator.

4.6 Related Work

Early work by the CGM Open Consortium with WebCGM [Gebhardt & Henderson 99] produced a CGM (Computer Graphics Metafile) specification and the WebCGM Profile became a W3C recommendation. The Scalable Vector Graphics (SVG) [Ferraiolo *et al* 03] work at the World Wide Web Consortium (W3C) has pioneered the development of 2D vector objects as an integrated web content type. This work is gaining support and is likely to be integrated with the next generation of web browser that is based on XML.

The CSIRO has developed an SVG Viewer [Robinson & Jackson 99] which is implemented as an applet and displays graphics downloaded from a URL. By using Javascript and the Document Object Model (DOM), client side application logic can be built around the SVG viewer. This is a powerful solution and is likely to gain favour for use in static vector display usage.

This work parallels the GGA work in many respects. The main areas where the work differs is in the server interaction and the lightweight client. The GGA is designed to interact with an application component compared to the SVG Viewer which has limited server connection options. The SVG viewer also requires a large Java library to be downloaded then stored on each client where the GGA is self-contained and lightweight.

The work on Displets [Ciancarini *et al* 98a] takes the approach of extending HTML and intercepting the extensions at the client with an applet displet manager that performs pre-parsing of the HTML stream and rendering of extension functionality into bitmaps. The GGA work concentrates on interactive graphics and dynamic content within the graphics applet. The displet work is suited more to extensions for HTML to display domain specific extended HTML rendering.

The ROSA Applet [DM 02] is a Java applet designed to display an image and support a number of button actions and drawing operations. It was originally developed to support geospatial web applications, but over time other features have been added to support other web application requirements. User events due to drawing operations are inserted in an associated HTML form and sent to a server application encoded as a MIME form. Buttons have a variety of settings and also populate form values to be sent to the server. The applet has a distance measurement tool and configurable legend box.

Thousands of other Java applets are available from Java sites such as Gamelan [Gam] and Java Boutique [Jav]; however, very few are general purpose and none offer the functionality, simplicity, and flexibility of the GGA.

4.7 Further Work with the GGA

The GGA uses a English-like text command language which is easy to learn and read, but is non-standard. The Scalable Vector Graphics [Ferraiolo *et al* 03] dialect of XML may offer a more standard method of describing graphics operations, though it is not intended as an interactive protocol and would still need to be extended to support functionality equivalent to the GGA.

Current browsers do not print canvas based applets well. More work is to be done understanding the limitations of browser printing and modifying the GGA to print inside the web page.

More testing needs to be undertaken on using the applet under real internet conditions and with more complex web application designs. Experimental work to date has taken place behind a firewall on a LAN.

The current prototype lacks any security features that are not inherently provided by the Java environment. More work is needed on connecting the applet to TCP ports on remote machines, authentication, and access control (to control the rights of applications to move, change or delete graphical objects created by other applications for example).

Web pages in web applications are mostly dynamic and applet pages which are replaced by another page remain dormant until that page is revisited. Research on understanding the behaviour and memory requirements of applets is a growing need. Hundreds of applets attached to dynamic pages which will never be revisited again would have a large impact on client machine memory usage.

To keep the UI component lightweight and flexible, only basic functionality was built into the applet. To create a map or chart requires computation and command generation in the application process to implement the display. A high-value enhancement would be the development of an extension to support maps, charts and diagrams of different types.

4.8 Summary

The existing HTML protocol supports the basic needs of displaying text, images, outlines, forms, and tables but the more graphical displays such as charts, diagrams and maps are unsupported. Web application developers and programmer product vendors have developed specialised and proprietary methods to display graphical data; however, there is yet to be a general purpose graphical display component.

The GGA is designed to fill this need. The applet is a general purpose component for web application developers to include small interactive graphical components or build complex user interfaces that rival desktop interfaces in functionality. It offers a simple human readable API, a range of orthogonal commands to manipulate the user interface, and comprehensive event management. The applet is lightweight and able to be used in applications without client modification. The one applet can be used to display all graphs, diagrams and maps, and the applet can be closely integrated with DHTML in the browser to achieve a flexible and highly functional user interface.

Chapter 5

The BUS Concept

5.1 Introduction

In Chapter 3, I discussed the problems facing web application developers and proposed a solution based on the presentation service architectural component. This chapter introduces a design of the Browser User-interface Service (BUS) which is based on the presentation service concept.

The BUS offers an object oriented presentation language to build custom web user interfaces that dynamically connect to application components. The components are able to re-use HTML, Javascript, and stylesheet content using prototype inheritance and dynamically bind presentation objects with data supplied in XML format. The BUS is designed to be independent of operating system, web-server, and browser software. The BUS uses XML messages on TCP sockets to communicate with distributed application component processes that implement business logic and connect with data stores. The BUS is a flexible component that is intended to improve consistency and flexibility in web interface design and application maintenance. The BUS is also an integration component that can connect a single user-specific user interface to multiple distributed application components.

5.2 Developing a Browser-based User-interface Service

I examined several aspects of web application architecture to understand the limitations and develop a software solution which would mitigate existing limitations and enhance the potential for web based applications.

An implementation of the *presentation service* was required that maximised the positive properties of web applications. The solution would need to encapsulate user interface functionality and enhance the user interface for users. It must provide methods to integrate and interoperate with other components in web applications, supporting communications with simple readable interfaces. The software must be easy to deploy, configure, and manage. The application interface must also maximise reuse of components and flexibility.

The solution is the *Browser User-interface Service* (BUS) [Sweeney 00a] that runs as a separate process on a server, connects to users via a minimal gateway program and offers a page composition and event application programming interface (API) to application components in a distributed computing system. The BUS prototype is written in Python [Lutz 96] because it offers rapid development features, a rich set of libraries, and high readability (it is sometimes referred to as executable pseudo-code). The user interface designer and application component developer do not have to learn Python as the APIs are designed with a simple object oriented language and XML data messages.

In Figure 5.1, a typical browser initiated transaction is illustrated. Application components (AppCmp) connect to the BUS dynamically through TCP sockets that carry XML [Cowan *et al* 06] messages. The browser communicates user requests to the gateway program which converts the GET and POST request, along with key values from the web server environment, into an XML event and sends this to the BUS. The BUS will use its internal logic to route the event to the appropriate application component and use the returning and stored presentation objects, and data objects to assemble a DHTML (or XML,PNG,WAV,WML) presentation stream.

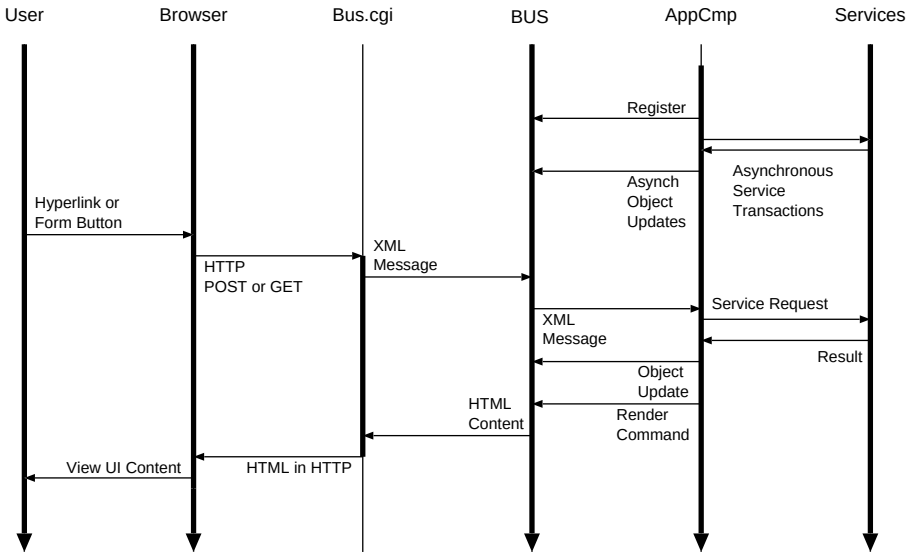

Figure 5.1: BUS to browser transaction chart.

The choice of an XML based application API is intended to help application developers by using a standard syntax with wide support. However, the use of XML based messaging applies a speed penalty to cooperating processes, but the development benefits are more important. Arbitrarily complex object structures can be communicated and altered without requiring re-compilation of either program. Unlike many other communication protocols, this method will also work with any language-platform combination that has a socket library and can process strings.

The use of the common scripting language Python, the CGI protocol, and XML messages on TCP sockets was chosen primarily so the architecture would have the maximum flexibility. BUS-based web applications can be built using any web server product on any operating system. BUS application components can be written in any programming language, and hosted on any operating system. The applications work with over 95% of the installed web browsers on any operating system without configuration change or software installation. This extreme flexibility is of particular use in large organisations without centralised IT control and employing many different computer systems.

This design effectively separates the presentation layer from the application and data layers. As the application logic and state are not located in the browser (unlike an application applet) or the web server (unlike standard CGI based applications), the application components can be shared among multiple sessions on multiple web servers, allowing collaborative applications.

5.2.1 Using the GGA with the BUS

The Generic Graphics Applet (GGA) has been introduced in Chapter 4, where we saw the many benefits of using this flexible UI component in web application designs. Developers using the GGA, however, still need to build code to generate GGA commands and interpret GGA return messages. Effort is also required to implement a design where all GGA connections are accepted, monitored and responded to promptly. GGA applications must also manage both the GGA interactions and web page interactions using the same sessions, state, and data structures. A presentation service has the potential to encapsulate these application requirements with reusable components and enhanced integration features.

The BUS was intended to only demonstrate the presentation service concept as a dynamic web page manager. This role has been expanded to include the integration of the GGA (and other prototype UI components described in Chapter 6). The BUS services GGA connections in a similar way to URL requests, except GGA connections are long-lasting where URL requests only exist for a single transaction. The URL and GGA events are normalised and managed identically in the BUS. A separate object store in the BUS manages the GGA presentation objects in a similar way to the HTML objects. The GGA presentation objects use the same data object structures as HTML presentation objects when rendering user interface changes. This means that data need only be updated in one place, and all presentation objects (HTML, XML, GGA, or other types) use a consistent view of the data when rendering user interfaces.

The transaction pattern for the embedded GGA component (Figure 5.2) illustrates the integration of the GGA protocol with the BUS user interface API. The graphics capabilities and the real-time transactions support truly interactive and effective user interface designs.

The GGA has three APIs for integration with web applications. The BUS is able to work with each of these APIs to support flexible and functional applications. Web pages generated by the BUS may include a GGA applet. This applet may have graphics commands embedded in parameter strings, created from BUS presentation objects. The web page may also have BUS-generated components which are configured to call the GGA methods: `cmd(STRING)` and `sendmsg(STRING)` using Javascript statements. The most flexible and interactive method is the dynamic socket connection to a BUS service using the BUS-allocated port.

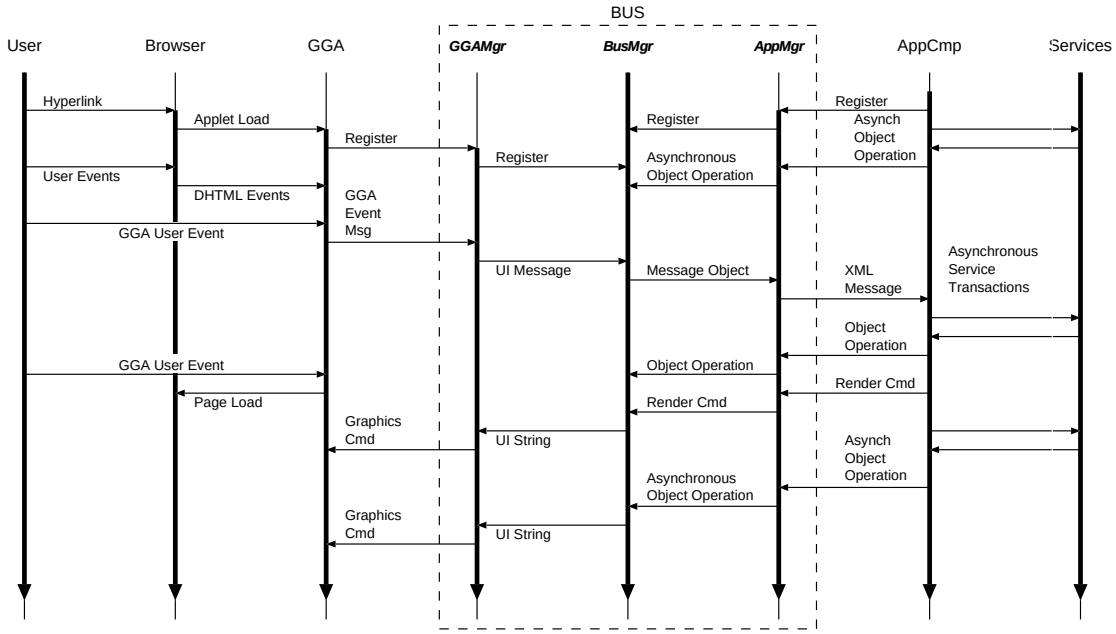

Figure 5.2: GGA transactions with the BUS.

5.2.2 The BUS Design

The architecture of a web application that includes the BUS technology (see Figure 5.3) uses the BUS as a *value adding* layer between application components and the user interface software. Application components and UI components have long-lasting connections through the BUS, and URL-based transactions use single transaction connections through a web server equipped with a lightweight *BUS Gateway* extension.

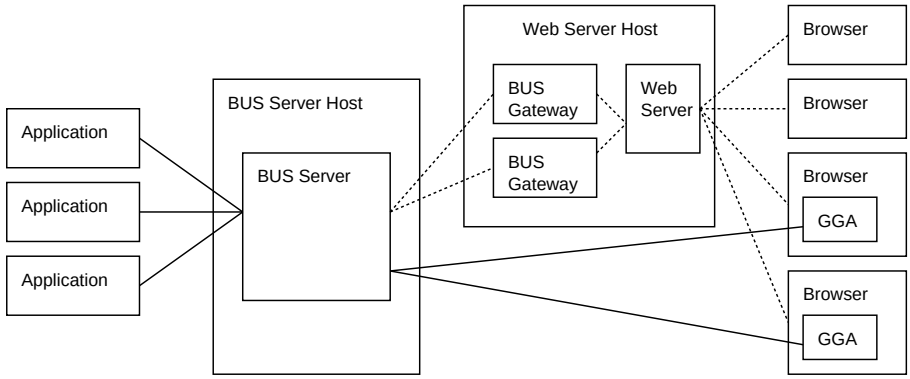

Figure 5.3: BUS Architecture.

The BUS Gateway is an adaptor that routes HTTP requests to a BUS server. Two versions of the BUS gateway have been developed: a CGI adaptor for the Apache web server and a dedicated simple web server. Both versions accept multiple HTTP requests, translate the

request data and environment variables into a BUS XML message, and send it to a known BUS server. The gateway then waits for a MIME-based reply, and sends this message to the client without change before closing client and BUS connections. The CGI adaptor is convenient to install in existing server installations but has the common disadvantages of CGI programs: high resource consumption and transaction latency. The dedicated server is a single process that uses a multi-threaded design to optimise throughput and limit latency, but offers spartan functionality beyond the BUS gateway requirement. Because the gateway works with HTTP and MIME messages, any MIME compliant media can be supported including HTML, XML, SVG, PNG, WML, JPG, PDF, and RDF for example.

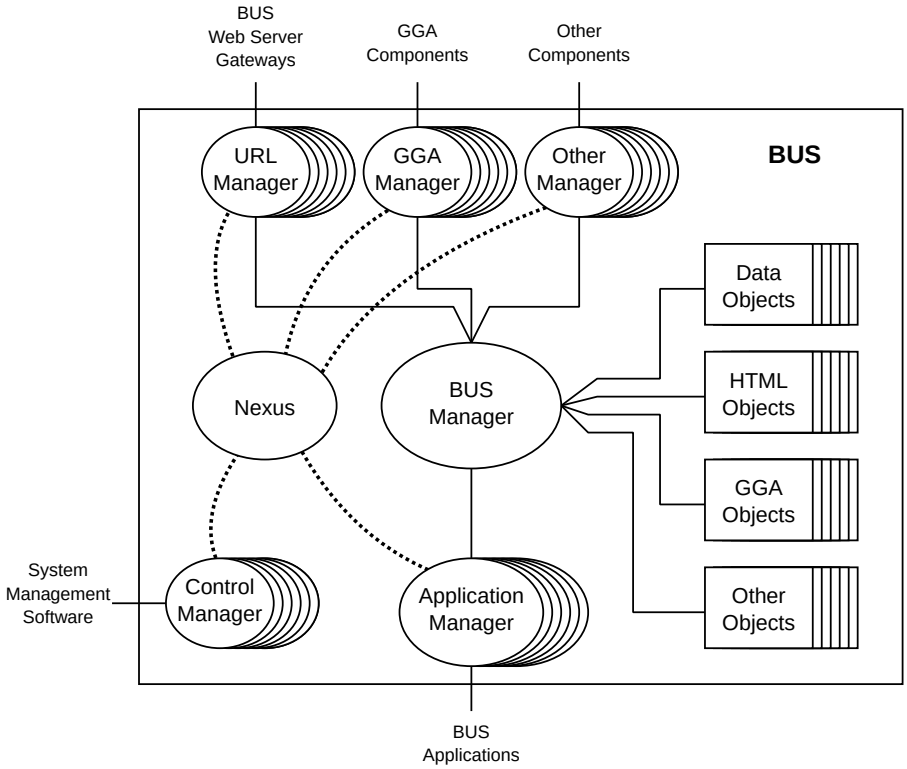

Figure 5.4: BUS Design.

The internal design of the BUS can be seen in Figure 5.4. The *nexus* is a socket-oriented asynchronous server that builds and manages the relationship between a TCP socket and a service object. The *Application Components* connect dynamically and are assigned an *Application Manager* by nexus. User interface components also connect dynamically and are assigned a *GGA Manager* (or another manager if they are a different type of component). As HTTP uses TCP sockets which are created and destroyed for every HTTP request, the assigned *URL Managers* also exist only for a single transaction; however, internal session management in the BUS can simplify application designs by creating a

virtual persistent HTTP user session. A virtual session makes sequential requests from the same client UI appear as a single permanently connected UI.

The single *BUS Manager* holds the control logic that operates the BUS. It receives new objects, object operation requests, and render requests from Application Managers as messages are interpreted from application component communications. The BUS Manager executes the request by manipulating the appropriate object structure, calling a method of an object it manages, or returns an error message to the originating Application Manager.

The BUS Manager also receives normalised user events from user interface managers. If these events are intended for the BUS itself, the BUS Manager interprets and acts on them. If the events include the name of a destination application, the event is routed to the responsible Application Manager for dispatch.

The *Control Managers* are created when an external process connects to the BUS control port. The Control Managers are designed to interpret and act on messages from system management software. When sent a recognised query, the Control Manager will read the requested BUS parameter or object, serialise it, and return it to the requesting software. It is also possible for authorised external software to send commands to the BUS which can change its internal configuration.

In Figure 5.5, the transactions inside the BUS are shown. A new user interface connection is registered to provide a link for responses. After registration new user events in a message object format are sent to the BUS Manager. A chain of events are then triggered which results in a user interface update stream being sent to the source user interface manager, and then on the the originating user interface. A URL Manager will terminate at this point, but long-connection components (such as the GGA) will remain open.

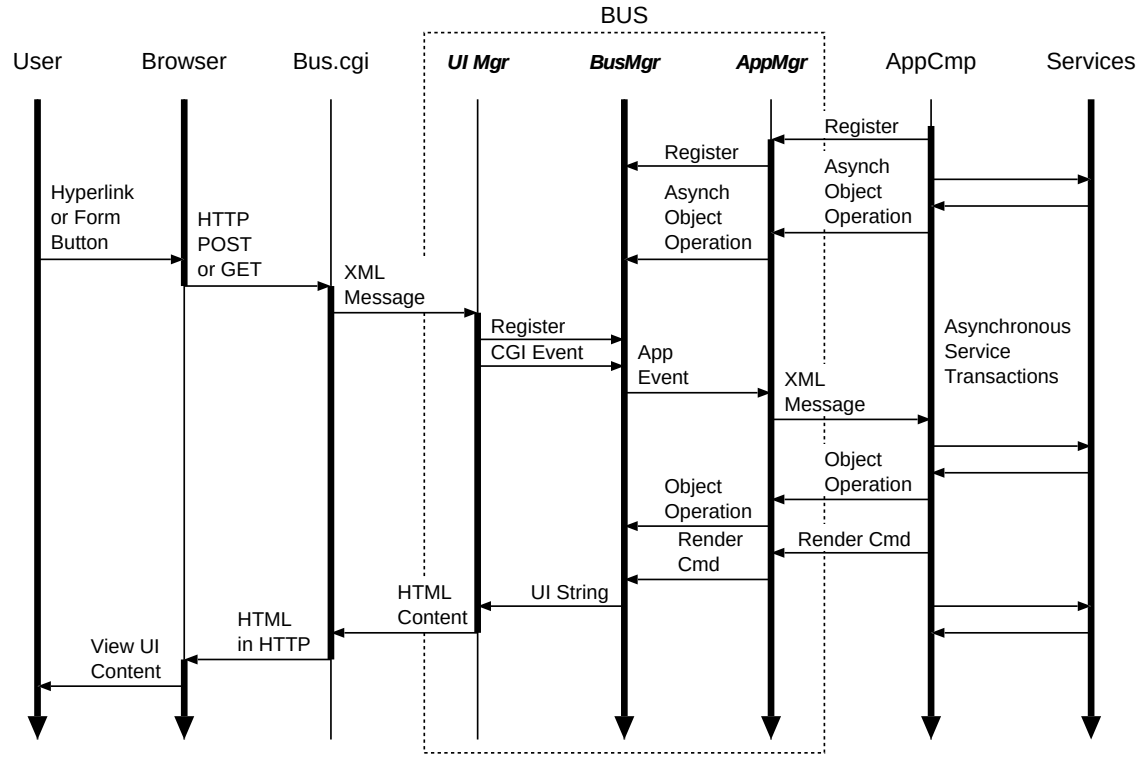

Figure 5.5: Inside a BUS transaction servicing a HTML request.

A browser attaching to the BUS through a gateway will be given an initial page and hyperlinks activated within that page will cause an XML GET or POST request to be sent to the BUS. If the request is for a different view of the interface, the control logic will call the render method of the required presentation object to render the new user interface. If the event requires application action, the XML event message will be constructed and sent to the appropriate application component. The application component may apply business logic, transform data, and use a connected database before responding. A response to the BUS usually requires sending an update for BUS data objects, and possibly a new user interface component (usually using parts of existing components). The final action is often to send a render command (mandatory for URL events) to update the user interface.

5.3 Application Communication API

The application API works on the asynchronous exchange of XML messages. The message design has been kept as simple as possible to aid readability, minimise the learning requirement, and reduce fault finding time. Unlike a SOAP-based message, the BUS message uses no extra layers of XML such as XML Schema, XML Data, or an envelope. A

minimal use of the *BUS* namespace is implemented to differentiate a BUS element from a content object element. The only extra XML technology employed is *XPath*, which is used in messages to describe which objects in the object stores to act upon.

The typical chain of events in a BUS and GGA enabled application is shown in Figure 5.1. Application components open a socket to the advertised BUS port, then send the BUS a registration message with a unique name. The application then uses BUS messages to build presentation and data objects inside the BUS. The application then idles and waits for user events. User events addressed to this application are received, then the application takes action based on its internal logic. The application then sends update messages to the BUS and finally sends a `render` command to update the user interface. The application would then return to idling, waiting for the next user event.

5.3.1 Messages from Applications Components

In Figure 5.6, the set of possible operation requests accepted by the BUS is shown. The most used operations are `add` and `render`. The `add` operation creates or replaces objects in one of the BUS object stores. The `render` operation instructs the BUS to direct the target presentation object to *serialise* itself, recursively rendering through sub-objects, and referencing data objects to be inserted in dynamic values. Note that the application must send the `connect` and `session` values with a render operation so the BUS can determine which user interface (or multiple user interfaces) are to be updated.

An example of the messages sent by the BUS to the application components and sent by the application components to the BUS can be found in section 6.3.1 and section 6.3.3.

```
Application Registration:
  <register name="STRING"/>

Mandatory attributes (except for <register>):
  connect__ = IDSTRING
  session__ = NAME

Action Requests:
  <render ctype=CTYPE dest=XPATH dataset=XPATH/>
  <config NAME=VALUE [offx,offy,pixdeg,...] >
    [<xml object>,...]
  <unconfig/>
  <query ctype=CTYPE src=XPATH/>

Object Operations:
  <comment> CONTENT </comment>
  <build ctype=CTYPE dest=XPATH/>
  <ref ctype=CTYPE dest=XPATH>
    [<xml object>,...]
  <add ctype=CTYPE [src=XPATH] [dest=XPATH] [offset=OFFSET]
      [position=POSITION] [context=keep|follow(def)]>
    [<xml object>,...]
  <move ctype=CTYPE src=XPATH [dest=XPATH] [offset=OFFSET]
      [position=POSITION] [context=keep|follow(def)]>
    [<xml object>,...]
  <del ctype=CTYPE dest=XPATH/>

Attribute Operations:
  <seta ctype=CTYPE [src=XPATH] [dest=XPATH] srcname=CSL destname=CSL
    value=STRING/>
  <rema ctype=CTYPE [dest=XPATH] destname=CSL/>

where:
  CTYPE = html | xml | gga | OTHERS
  XPATH = XML object selection specification
  OFFSET = after(def) | before | replace
  POSITION = first | INT | last(def)
  INT = integer describing position of reference
      (negative indicates reverse index: -1 is last object)
  CSL = comma seperated list of attribute names
```

Figure 5.6: Operation requests sent by Application Components.

The abbreviated XPath language used in the BUS operations is defined in Figure 5.7. The syntax is similar to file path specifications, with added numeric indexing and selection by attribute. For the XPath implementation in the BUS, I added the non-standard *negative numeric index* option for the selection of items located from the *end* of the sub-component list.

```
/ : Root
. : Self
.. : Parent
// : Recursive Search
* : Wildcard
[INT] : Index (base=1)
[-INT] : Index from end (BUS extension)
[name] : Node Select by Child Tag
[@name] : Node Select by Attribute Existence
[name="STR"] : Node Select by Child Tag and Child Content
[@name="STR"] : Node Select by Attribute Value
TAGNAME : Selection of Children by Tag
```

Figure 5.7: XPath syntax supported by BUS in object operations.

5.3.2 Messages to Application Components

Applications are sent events by the BUS, when user events are sent to the BUS tagged with the name of this application. GGA events are directed based on the application name which makes up part of the graphic object's name, or a default application assigned to the BUS session. It is possible for many graphic objects in the same GGA display to be managed by applications using no cooperation (eg: a map serving application and the application managing the aircraft track symbols having no knowledge of each other).

```
<cgi|event|message|invocation|error
  session__ = NAME
  connect__ = XXX (managed by BUS for APP use)
  client__ = html(def)|xml (SUPERCEDED)
  userid__ = user name (authenticated from UI?)
  format__ = html|txt|xml (from BUS_EXT)
  urlbase__ = web site URL base (from SCRIPT_NAME)
  NAME = VALUE (xN) (derived from <event|cgi>)
  />
<result NAME=VALUE (from query)>
  [<xml object>,...]
```

Figure 5.8: BUS messages sent to Application Components.

The syntax of the user events is specified in Figure 5.8. The message will be a user event message or a result from an earlier query to the BUS. The event message will contain values to the connection ID, session ID, type of media the client is expecting, and the

web server gateway path (for URL events only). An authenticated user name may also be present if the web server has authenticated a URL requester, or the GGA user has been authenticated via a custom method. Each of these BUS-supplied tags have a trailing double underscore in the name to separate then from user data (user data fields that have a trailing double underscore are truncated). The remaining attributes refer to user data. The URL messages (with the `cgi` tag) have GET and POST data from the hypertext URL or HTML form. The GGA messages (with the other element tags) have object ID, x, y, action type, and other optional fields.

When an application sends a `query` (see Figure 5.6), the BUS will fetch the indicated objects and return them in a `result` message, so the application can determine the current state of its own and other's objects in the BUS object stores. The query attributes are repeated in the result attributes so the application will know which query this result is an answer to.

5.3.3 Dynamic Presentation Objects

All BUS presentation objects may use special BUS *dynamic objects* in their object structures (see Figure 5.9). These objects are interpreted by the BUS at render-time and implement looping, selection, data referencing, presentation referencing, recursion, and variable updates. The presentation reference instructs the renderer to render presentation objects managed by other applications using the current data context.

All presentation objects including dynamic presentation objects can use *Active Expressions* in content and attribute values. This expression evaluation is triggered by surrounding the value with dollar signs. A number of examples illustrate how active content is used:

```
<div> $ 'Take off at ' + data.LaunchTime + ' Tuesday'$ </div>
<td class="$ var.cellstyle $"> $ data.organisation $ </td>
<span> $ 1024 * int(data.kilobytes) $ </span>
```

These active expressions can use most of the capabilities of the Python language, and can use properties of the current data object and variables within scope (see `SetVar` in Figure 5.9). For the definition of the syntax of active expressions, see Appendix D.

```
<Reference busobject=LABEL>
<UseData [dpath=XPATH | ref=OBJREF]> BLOCK
  OBJREF=data (“.”+ATTRIB | “[” INDEX “]”)*
<UsePres [prespath=XPATH | ref=OBJREF]> BLOCK
<SetVar name=IDENT value=STRING> BLOCK
<Choose>
  BLOCK
  <test expr=EXPR> BLOCK then exits...
<If expr=EXPR>
  BLOCK
  <Then> BLOCK
  <Elif expr=EXPR> BLOCK then exits...
  <Else> BLOCK
<ForEach [dpath=XPATH | ref=OBJREF] [seq=data(def)|pres|match]>
  BLOCK (with p_total, d_total, p_index, d_index vars)
```

Figure 5.9: Dynamic presentation objects.

These dynamic elements can be used in the HTML, XML, GGA, and other data stores as required by the application designer. Variables can be set with the `setvar` element, and these values exist within the element scope. Inside the `foreach` loop, four additional variables are available which contain the total and current index of child presentation and data elements used in this loop.

Any element can have a `busname=LABEL` attribute. This is used by the `reference` element, which will search back through its parent objects for the given label, then render *that* object and its sub-objects using the current data context. This recursive rendering is used for trees of data that use the same presentation. Note that this recursion is self-terminating as the renderer will, at some point, reach the bottom of the data tree.

This combination of dynamic rendering capabilities encourages the reuse of presentation components and the integration of external presentation components from different applications. These features allow the creation of user interfaces that contain combinations of user-oriented displays and links from different applications.

5.3.4 Graphics Presentation Objects

The major work in adapting the GGA to the BUS design was the creation of a library of objects which can be expressed in XML, support the dynamic elements mentioned previously, and *serialise* themselves into valid GGA commands.

Mandatory:
  Id = IDENT ("Id"+INT automatic default Id format)
  idgen=IDENT (alternative ID generator - optional)
  x1=NUM
  y1=NUM
For line and rect:
  x2=NUM
  y2=NUM
General options:
  geo = r|t (r=georeferenced,t=geotransformed optional)
  busname=LABEL
Object options:
  draggable=YES
  fixed=YES
Properties options:
  linewidth=0-16
  linecolour=COLOUR (COLOUR=WORD|HHHHHH hex value)
  fillcolour=COLOUR
  filled=BOOL
  xmin=NUM|null
  ymin=NUM|null
  xmax=NUM|null
  ymax=NUM|null
Font options:
  style= (plain | ([bold] [italic]))
  fontsize=INT
  fontname= serif | sansserif | monospaced | dialog | dialoginput
  justify= left | right | center
Notify options:
  all=YES
  click=YES
  doubleclick=YES
  grab=YES
  drag=YES
  drop=YES
  annotation=YES
Link options:
  link=URL
  target=self(def)|blank|parent|top|NAME

Figure 5.10: Attributes of GGA presentation objects.

The common attributes of the graphics elements are listed in Figure 5.10. Attributes are serialised into command options when the GGA presentation object is requested to render itself. Attributes which are not specified can take the value present in the current *default* environment inside the GGA.

```
<set> BLOCK
<defaults [FONT] [PROP] [NOTIF] [outline=MODE]>
  MODE=freehand|rectangle|circle|ellipse|text
<image img=URL X1Y1 [OPT] [alpha=0-1.0]>
<line X1Y1 X2Y2 [OPT] [PROP]>
<rectangle X1Y1 X2Y2 [OPT] [PROP]>
<circle radius=NUM X1Y1 [OPT] [PROP]>
<ellipse [major=NUM] [minor=NUM] X1Y1 [OPT] [PROP]>
<polygon [closed=YES] plist=POINTS X1Y1 [OPT] [PROP]>
  POINTS = [(x,y),...] | STRINGREP
<symbol [xscale=RATIO] [yscale=RATIO] [heading=DEGEAST]
  [symbol=plane(def)|plus|triangle|diamond|hexagon|arrow|square|
  airport|radar|towedgun|tank|bigplane|ground|ship|sub|missile]
  [closed=YES] X1Y1 [OPT] [PROP]>
<ovalarc [majrad=NUM] [minrad=NUM] [startangle=DEGEAST]
  [endangle=DEGEAST] [segment=YES] [closed=YES] X1Y1 [OPT] [PROP]>
<text text=STRING [FONT] X1Y1 [OPT] [PROP]>
```

Figure 5.11: Graphics creation elements in GGA presentation objects.

In Figure 5.11, the syntax of graphics elements that can be used in the GGA display are described. The `defaults` object is used to set default attributes for objects that follow, reducing the need for repetitive attributes, and improving consistency. The image, line, rectangle, circle, ellipse, polygon, and text elements mirror their counterparts in the GGA command set (see Section 4.3). The symbol and ovalarc objects on the other hand, are *virtual* objects. They are used as symbols and arcs in the BUS and applications, but are implemented by many-sided polygons in the GGA applet.

The full set of GGA actions have been implemented in GGA objects (see Figure 5.12). These objects will send an instruction to the GGA to change a graphic element or user view when rendered. Like all BUS objects, these actions can be assembled into sequences that can be used to implement sophisticated graphics effects.

```
<hide obj=IDENT>
<show obj=IDENT [alpha=0-1.0]>
<move obj=IDENT [abs=BOOL]>
<remove obj=IDENT>
  Note: Magic IDENT "all_objects" deletes it all.
<layer obj=IDENT where=PLACE [target=IDENT]>
  PLACE=up|down|top|bottom|above|below
  (above and below require target)
<pan [amount=PERC"%"|INT] direction=n|ne|e|se|s|sw|w|nw>
<centre obj=IDENT>
<load url=URL [target=self(def)|blank|parent|top|NAME]>
<group [drag=BOOL] group=IDENT members=IDLIST>
  IDLIST=IDENTS separated by single space
<disband group=IDENT>
<expel member=IDENT group=IDENT>
<joinup member=IDENT group=IDENT>
```

Figure 5.12: Action elements in GGA presentation objects.

The reusable properties of the graphics elements, when combined with dynamic tags, active content, and shared data resources, make this pairing of BUS and GGA technology a potent combination for flexible and effective user interfaces in web application designs.

5.4 Building BUS Applications

The BUS uses a number of advertised TCP/IP ports to communicate with other web application components in a similar way to SQL database engines and web servers (eg: port 80).

A number of special URLs are available to the web application developer to help in testing systems of components and diagnosing faults. The BUS can provide information on the BUS gateway configuration, application component connections, internal data store contents, and connected user interface components.

On startup, the BUS is configured to load a local XML file called `bus.xml` and process these object operations. This foundation configuration file can load other files and be annotated with comments.

The BUS XML file loader recognises the special line syntax:

include files : [SPACE] "#include" SPACE FILENAME
comments : [SPACE] "#" COMMENT "\n"

The include directive reads and processes the next XML file before continuing with the next line. The comment lines are discarded before processing by the XML parser. These two additions add simple improvements to XML configuration files to assist with reuse, modularity, and maintainability.

5.4.1 Using the GGA UI Component

The GGA is embedded in a web page using the familiar `object` element (see Figure 5.13). To attach to the BUS port for servicing GGA connections, the PORT value must be set to the 11907 value. When using the BUS services, GGA instances should not include graphics commands in the `param` elements, as the BUS and serving applications will have no knowledge of what has been placed on the canvas.

```
<object name=GGA
  classid=URL
  codebase=PATH
  archive=gga.jar
  code=dsto.himalaya.HimalayaApplet.class
  class=STYLE
  width=INT
  height=INT
  >
  <param name=PORT Value=11907/>
  <param name=HORIZONTALSCROLLBAR value=true/>
  <param name=VERTICALSCROLLBAR value=true/>
  <param name=CMD_ID value=CMD_OPERATION/> ( zero or more commands )
</object>
```

Figure 5.13: Using the GGA UI component in a web page.

The GGA applets in a dynamic page can be sent a message or a command using the standard DHTML GGA interface:

```
document.gga.cmd(STRING)
document.gga.sendmsg(STRING)
```

Commands sent to the GGA will be reported to the attached application as an `invocation` event, so the application can track external changes to the GGA internal state. Messages sent using the `sendmsg(STRING)` method can carry structured language commands for the BUS or applications to interpret.

The message capabilities of the GGA can be used to add new capabilities to the GGA and BUS relationship. Dynamic (or static) pages with GGA applets can embed the `sendmsg(STRING)` command in applet `param` elements to send information to the attached server when the applet is loaded. The `sendmsg(STRING)` method may also be used from DHTML for the same purpose. The *GGA Manager* in the BUS currently recognises two forms of message syntax sent via this method:

```
message "_set NAME = VALUE"
   where NAME = session|ggaver|appname
message "_auth USERNAME = MD5_HASH"
```

The `set` message stores the name-value pair in the connection environment, which is passed on to applications. These values can also be used internally by the BUS to form sessions, and modify output depending on the software version. The `auth` message returns a user name and security hash supposedly sent by this BUS instance. The user name will be hashed with the secret BUS server number and compared to the user supplied hash before associating this user name with this connection. This is a simple form of authentication transference, which is not robust, but useful for experimentation purposes.

5.4.2 Using the Control Port

The BUS control port is accessed by connecting to BUS port 11910. Passing an XML formatted query on this socket will request the BUS to look up an internal data structure (see Figure 5.14). The query can request the value of an expression in the *Control Manager* context, or the value of a *BUS Manager* attribute.

```
<query expr=OBJEXPR/>
  returns:
<result error=MSG [expr=OBJEXPR]> OBJECT_TREE </result>

<query param=ATTRIB/>
  returns:
<result error=MSG [param=ATTRIB]> VALUE_STRING </result>
```

Figure 5.14: Syntax of messages using the BUS Control Port.

The requested value is serialised (possibly into XML for an object from the object stores), and returned in a `result` message. This feature allows management software to track transaction rates, memory usage, current sessions, and names of attached application components for example. In future this feature could be extended to allow authenticated users to change parameters in the BUS to optimise performance. This feature is designed to enhance the manageability and instrumentation of BUS web application designs.

5.4.3 Large Scale BUS Systems

A highly distributed system can be developed using the BUS architecture. In Figure 5.15, an example system is illustrated that shows how multiple clients of varying types can interact with applications using more than one BUS service. Browsers of different types interact with BUS-enabled applications via standard URL requests and MIME data display (such as XHTML, XML data, PNG image, or WAV sound). The GGA instances connect directly to the designated BUS service and interact asynchronously via user events and graphics update commands.

External applications may also fetch datasets in XML from the BUS directly using the BUS API, or via a web server and BUS gateway using the standard URL protocol. Web-enabled applications can use the gateway feature without modification – the gateway on the server behaves like any other web server resource.

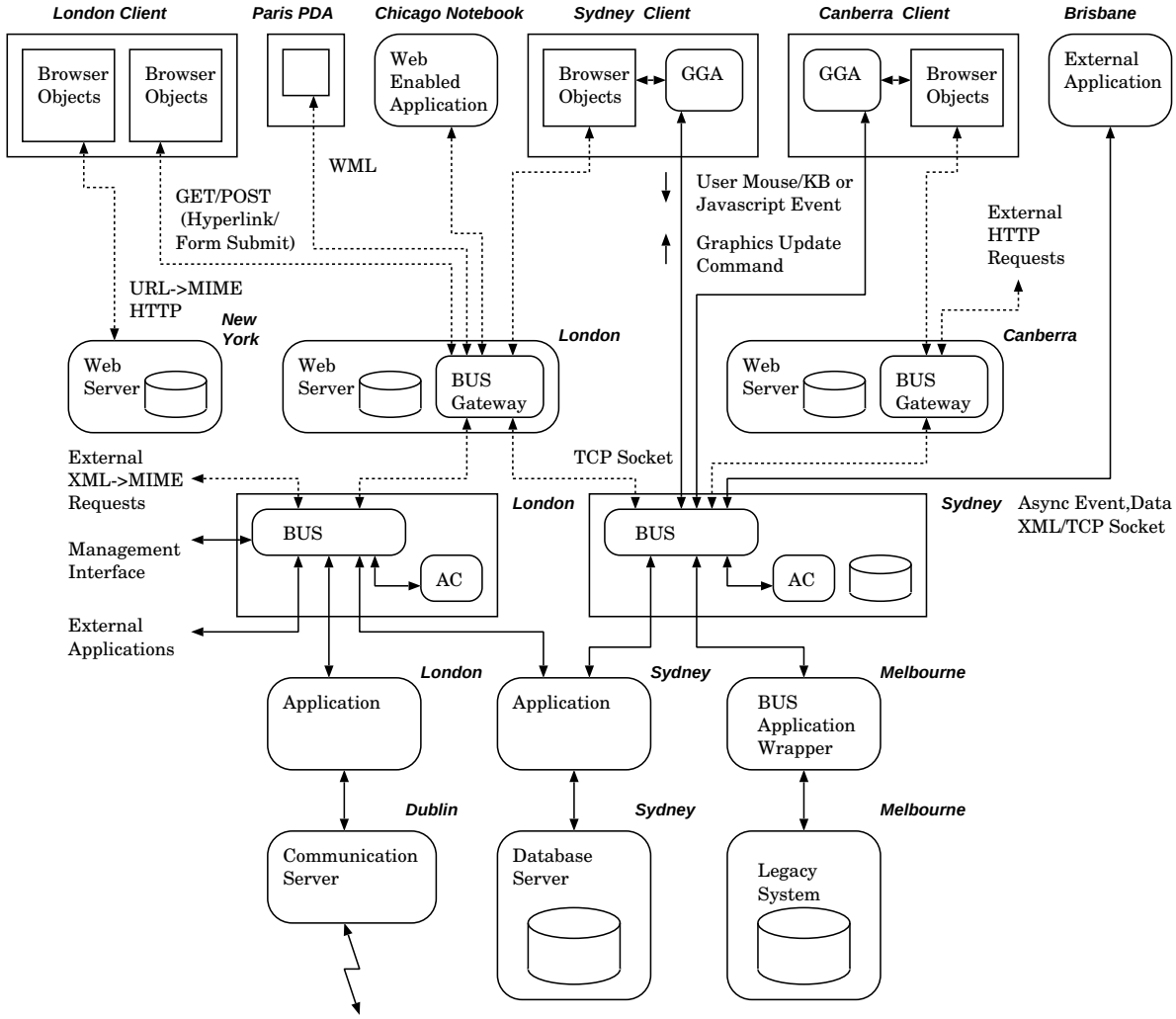

Figure 5.15: A distributed web application system example using BUS architecture.

Application components (AC in Figure 5.15) can be located on the BUS server host or another host available over a network. The applications may also connect to more than one BUS service (for reliability or load management) and may provide another API for client connections or its own user interfaces.

The architecture also enables highly reliable and scalable services. If a web server should fail or become overloaded, the browser based application can be routed through another server equipped with BUS without losing the session. Should the application component fail, other instances of the application component can be used by BUS with the same protocol; however the current state information will be lost if not designed into the application program.

Systems based on BUS technology have been tested on a small scale where the BUS, web server, BUS gateway, application components, and the database run on the same

server host. The technology also scales to a very large size with globally distributed web servers with BUS gateways, BUS servers, applications running on other servers connected to standard database engines. Such large systems allow specialised server installation, gradual degradation on failures, and dynamic load sharing using established designs from large scale web server systems.

5.4.4 Collaborative Application Support in BUS Architecture

Collaborative applications enable geographically distributed teams to work effectively in a common information environment. This is achieved through common views, shared data, change notification, workflow, group work status display, and communication tools. Web applications are often the best choice for collaborative applications because they offer a central system to maintain state, store data, and deploy software. The problems of peer-to-peer designs (such as synchronisation and compatibility between platforms and software versions) are avoided.

The BUS and GGA technology support the design of collaborative web applications with shared view components, shared data stores, asynchronous content updates, real-time interactive graphics, and multiple-user graphics sessions. The shared but composable view components allow flexibility but also encourage reuse so the team members have a consistent view of the work space. The data stores in the BUS are able to be referenced by other presentation components, ensuring data need only be updated in one place so that team members have current and consistent information within the UI. Asynchronous content updates are achieved through communication through the GGA, triggering DHTML content update when the server state changes. This asynchronous update provides real-time UI changes that reflect each team members' contribution – ensuring that the team is always working in an internally consistent and valid information environment.

The most innovative aspect of the GGA though is the provision of real-time interactive graphics where changes in the graphics space can be shared with other team members as updates occur. This *shared graphics session* can support shared whiteboard, a distributed slide show, simultaneous annotation of an image, continuous situation assessment and dynamic distributed planning using a map, and group creation of large mind maps for example. The integration between the GGA and BUS technologies also allows form users

to create shared graphics objects, and graphics events to display BUS generated pages. Other uses for the GGA include a shared *ticker tape* for group news, a shared navigation display for drilling down in complex deep content, and a central clearing house for spatial and image related information in an emergency situation (such as placing events on a map, or assigning data to people or vehicles in a security camera image).

5.5 Technology Evaluation

The BUS language and transaction pattern is designed to be compliant with the specifications given in chapter 3, so the benefits of this technology will be aligned with the benefits (and limitations) of the more abstract presentation service in section 3.6. The benefits of the BUS technology are summarised in Table 5.1.

Table 5.1: Benefits of developing applications with the BUS.

Capability	*Benefits*
Hides user interface details	Eases developer work by using abstracted web and graphics structures.
Adds reusable objects to DHTML	Eases re-use of tables, pages, lists, and more complex compound components.
Manages collaborative sessions	The BUS can transparently transmit UI rendering to multiple users and simulate a single user to applications.
Integrated graphics	The GGA can display interactive maps, diagrams, and charts.
Promotes consistent user interfaces	Reuse of objects and graphics increase application consistency and developer productivity.
Webify legacy applications	Adding adaptor code will allow legacy applications to work over the web through the BUS.
Application integration	Multiple applications can contribute to a single user interface and a single user interface can launch events to multiple applications.
Interoperability	Operating with external web systems is eased with multiple interfaces to integrate with.
Speeds development	By letting the BUS handle the user interface, developers can work faster and concentrate on application logic, flow, and transforms.
Allows client profiles	The BUS can change its behaviour to suit user profiles. It can scale from simple text on PDA screens over GSM up to rich graphical content on multiple powerful LAN workstations.

When evaluating the BUS technology with the web application properties as criteria (see

Table 5.2), we observe that many of the properties are positively influenced by the use of the BUS.

Table 5.2: Evaluation of BUS Web Application Properties.

Property	**Enhance**	**Features**
Sep. of Concerns	√√	Separate UI, web services, BUS, and application layers
Simplicity	√	Simple language, events, application API
Modularity	√√	Tiered design, modular internal design
Familiarity	√	Standard HTML + graphics element
Learnability	∼	Basic XML used, simple applications work
Useability	√	Integrated UI
Consistency	√	Reuse of presentation
Orthogonality	√	Data, pres, app independence
Interactivity	√√	Async GGA protocol + DHTML integration
Multi-media	√√	Enriched graphic display
Continuity	∼	Seamless application restart
Flexibility	√√	Combinations of UI and graphics, APIs
Customisability		
Productivity	√	Built UI from other UIs, async web update
Configurability	√	Base presentation + data in BUS XML config
Collaborative	√√	Multi-GGA + web user sessions
Adaptability	√	Adapt to new UI via new manager
Clarity	√	Defined manager roles, objects, protocols
Reuse	√√	Reusable presentation, data, referencing
Inheritance	√	Objects copy and overload
Encapsulation	√√	UI handling service with graphics
Lang. Independence	√√	Any application language
Interoperability	√√	Connects multiple APIs, URLs, XML
Multiple UI Devices	∼	Possible mobile extension, email adaptor
Open Standards	√	DHTML, sockets, XML, XPath, CGI, HTTP
Multiple APIs	√√	URL, HTML, XML, GGA, Control, App APIs
Thin Client	√	Only standard browser needed
Applicability	√	General purpose web application tech
Scalability		
Plat. Independence	√√	Any modern browser without change
Efficiency	∼	Shared CPU tasks, low BUS-app traffic
Maintainability	√	Split applications, reusable modules
Instrumentation	√	Control port
Manageability	√	One BUS process, web deployed applet
Confidentiality		
Integrity		
Availability	∼	BUS can serve without app, BUS is critical node

Legend: ∼ = partial enhancement, √ = enhancement, √√ = major enhancement.

It implements the separation of concerns at several levels. The BUS-based design has five tiers: The browser, DHTML, and GGA for the user tier; the web server with local resources and BUS gateway form the second tier; the BUS itself is the third tier; the applications and

data services form the fourth and fifth tiers. The BUS also maintains a separation between data and presentation logic up to the time the UI requires an update. Configuration files use the XML syntax with transparent extensions, allowing non-programmers to build UI templates with XML editors.

To enhance system modularity, the BUS is separated from the GGA, web server, applications, and management software by simple but effective interfaces. Internally the BUS is also very modular, allowing the support of new UI managers and presentation object types with minimal changes to existing code.

The BUS and GGA technology is very strong on integration, user support properties, and a structured architecture. The developer, system support personnel, integrators, and users benefit from this architecture's clear focus on a high quality user interface while minimising the work programmers and integrators need to perform.

The BUS, like most web technologies, has limitations too. The BUS design shows some intrinsic limitations due to design choices:

- The speed is limited by multiple layers and XML parsing.
- Careful design is needed in asynchronous servers or one complex transaction can block all others until a time-out triggers.
- Using the BUS imposes another protocol and language web developers need to learn.
- There will be delays in adopting new protocol extensions, as the BUS adaptor will need to be modified.
- The BUS and applications must be reliable. Failures, data integrity problems, or memory leaks will have serious consequences, as these processes have global effects and must run without restarts.
- The BUS is a single point of failure. A software failure would terminate ALL current transactions leaving applications and user interfaces unconnected. Errors must be caught and handled gracefully (with rollbacks and restore).
- Database data must also be duplicated in the BUS for presentation objects to render it. This dataset duplication in the DBMS, application, and the BUS adds unwanted complexity and inefficiency to application design.

It would be possible to attack these limitations with aggressive design changes, but the resulting design would be likely to exhibit other design flaws. These design trade-offs affect all complex designs, so the best approach is to support the primary design features and minimise the effects of negative side effects.

The BUS *prototype implementation* also has some limitations:

- The speed is limited due to using an interpreted language and an unoptimised prototype.
- Only simple error detection and recovery is currently implemented.
- There is no access control between application object trees, so any application can write over or delete any other object.
- The user preferences capability is not implemented.
- There is limited protection against cross-site scripting attacks.
- The BUS does not currently check on the impact of possible operations. Large data transactions could use all memory and cause a fatal error.

The application of more development effort would overcome these problems for use in production systems.

5.6 Summary

In this chapter I have described the implementation of the presentation service in a Browser-based User-interface Service, which employs a number of novel features: The BUS supports component re-use through a prototype inheritance mechanism that is easy to learn and use; a simple language-independent XML language is used for communication with applications; and multiple users using different user interfaces can collaborate in the same session.

The BUS offers a unique solution to some of the problems that affect web application design; it is designed as a service that web application developers can use to build better web applications faster. The service hides some of the complexities of the web user interface protocols and languages, while offering significant engineering benefits. The BUS

architecture is designed for flexibility and platform independence and provides interface methods to suit many web application requirements.

In the next chapter, I will employ the BUS and GGA technologies in a number of experiments. The analysis of these experiments will explain more of how the BUS works in practise.

Chapter 6

Prototypes and Results

6.1 Introduction

The BUS and the GGA were designed around software engineering principles applied to the web application development domain. The core of this thesis is the claim that the presentation service and browser component simplify the development process and enhance the qualities of the resulting web applications. In order to test the improvements to the development process and application quality, a number of experiments were conducted using these new technologies.

The benefits of improvements in software engineering only become obvious in large projects where increasing complexity can cause severe losses in productivity and escalating fault counts. A large project is not the place to test new engineering concepts, so another method must be found.

In chapter 1, I presented a set of qualitative properties that web software can be evaluated against. I will use a number of small experiments in this chapter to demonstrate how the BUS and graphics component are used, and explain how the use of the new approach provides improvements in many of these web application properties.

6.2 Experimental System Design

The architecture of the system used to evaluate the characteristics of the prototype BUS and UI component is shown in Figure 6.1. The application behaviour is implemented in the experimental application component and, like most current web application designs, an external SQL database is used for data services. The significant difference from other designs is the use of a presentation service and UI components to handle common user interface functions and add to the user interface capability.

The architecture diagram in Figure 6.1 is colour coded to identify the role of each component. The red components are under test and are the focus of the experiments. The grey components are open source software in common use in many web application designs. The software components and data that have been acquired, adapted, and created for these experiments are coloured blue. Lastly, the components that have been developed to evaluate the features and characteristics of the BUS and UI components is shown in green. The communications paths, user interfaces, and internal data stores are also shown in green, as these artefacts can also be examined to gain insight into the operation of the web application.

Note that inside the application component, process flow appears similar to other designs, with familiar validation, authentication, and data access functions. The two major design differences are that the BUS application component must be able to handle multiple concurrent users and sessions, and responsibility for user interface handling is delegated to the BUS.

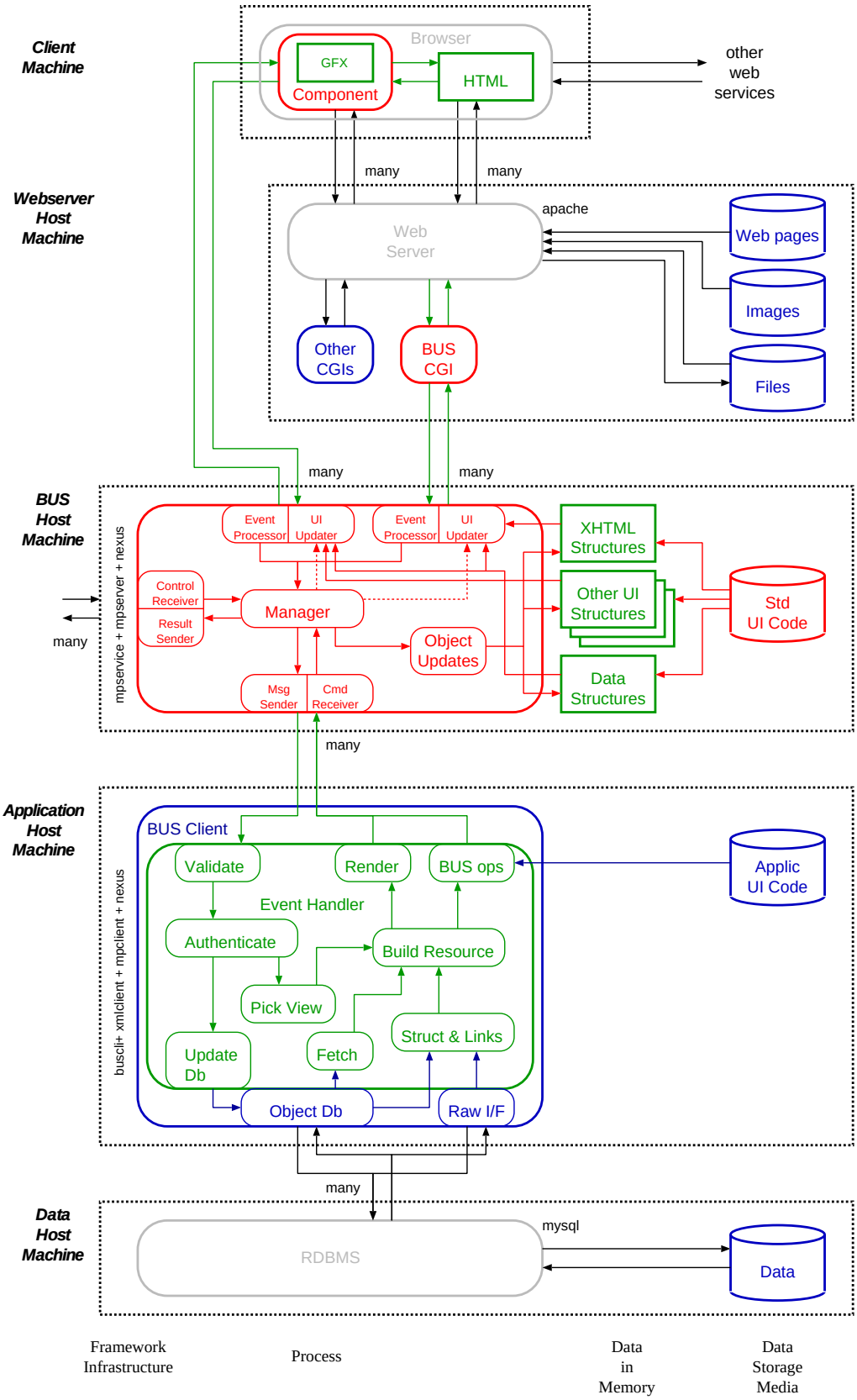

Figure 6.1: The architecture of the prototype experimentation framework.

6.3 A simple web forum application

A simple BUS application was implemented to investigate and demonstrate the design of a web application using a separate presentation engine. This simple web forum application allows users to maintain a list of topics, where each topic has a name and a set of associated posts. Posts may also contain other posts (or comments).

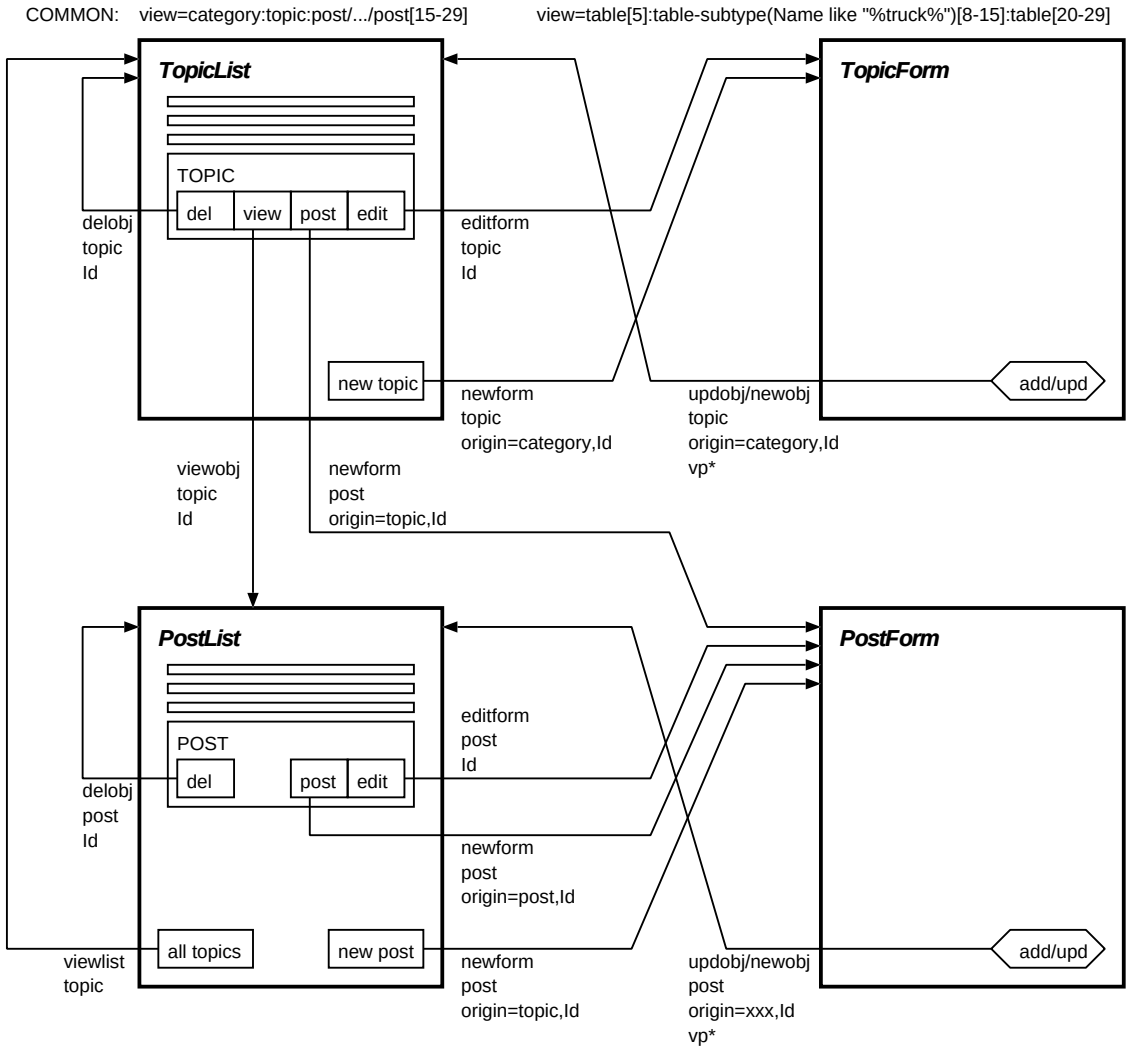

Figure 6.2: Web forum application design.

The application design is shown in Figure 6.2. The design consists of two list displays and two form displays. The small boxes are buttons which send the given parameter set to the BUS when clicked. The small polygon in the form displays are *submit* buttons that send form data with the parameters. The "del" buttons invoke a *POST* BUS event (as do the submit buttons), and other buttons invoke standard *GET* events using HTML hyperlinks. The arrows point to the display to be generated when this event is triggered.

The design of the application follows a standard pattern:

Application initialisation: Allocate internal structures and connect with external services (such as an SQL database).

BUS registration: Send the name of the application to the BUS for registration. Any event sent to the bus with `appname__="swforum"` will then be routed to this application.

BUS initialisation: Create a "branch" of the data object tree, and send a set of presentation components to the presentation tree structures that this application supports (just *HTML* in this case).

Respond to events: Block on the BUS socket waiting for a user event.

- **Validation:** Validate the user event fields to ensure mandatory fields are present and values are within expected ranges.
- **Authentication:** Check the authentication fields (if present), and set an authenticated user name for this connection if correct.
- **Database update:** If the operation is permitted, and the user event includes a request to create, update, or delete data, send the database request.
- **BUS data changes:** Identify the display the user is requesting, or the display that logically flows from the previous operation. If the data needed to support the presentation rendering is not present in the BUS, fetch data from the database and send new or the updated data objects to the BUS data store.
- **BUS render UI:** Send the command to *render* to the correct BUS presentation object.

Termination: Close BUS socket, close any service connections, save internal state, and close database connection before exiting.

6.3.1 Initialisation

When initialising the BUS objects, it is good design to delete existing objects which may exist from a previous relationship with this application component (remember that the BUS is a long running service). This cleanup is achieved with the BUS operation:

```
<bus:del ctype="html" dest="/swforum"/>
```

This operation will instruct BUS to find all elements of the root object "html" with the tag "swforum" and delete them and all of their children. Now we can create a fresh presentation object tree:

```
<bus:add ctype="html" dest="/">
  <swforum>
    <ErrorPage>
    <PostPage>
      <html>
        <head><title>Edit Post</title></head>
        <body class="std">
      ..page template here..
        </body>
      </html>
    </PostPage>
  </swforum>
</bus:add>
```

For this application, I have used XML configuration files to initialise the presentation object store. A preprocessor executes on the XML configuration file when it is read into the application component. The preprocessor simply interprets lines beginning with a hash (#) as a comment to be excluded, except for lines of the form:

```
#include template/postform.xml
```

where the engine inserts the included file and continues processing with those contents. These preprocessor extensions help manage and reuse XML resources, but still allow an XML editor to read and write the file.

A simple presentation component for displaying an error to the user should be available for every BUS application. Here is an example error display that can be included in an application presentation structure, or installed in the root "html" object:

```
<ErrorPage>
  <html>
    <head><title>Application Error</title></head>
    <body bgcolor="#ff88cc">
      <h1>Application Error</h1>
      <p> An application error has occurred.
```

```
      Full error data is in the log file. </p>
    </body>
  </html>
</ErrorPage>
```

If the application detects an error it cannot handle during a HTML resource request, the application can simple instruct the BUS with:

```
<bus:render ctype="html" dest="swforum/ErrorPage" dataset="swforum"/>
```

The application branch of the data object tree also needs to be initialised:

```
<bus:del ctype="html" dest="/swforum"/>
<bus:add ctype="data" dest="/">
  <swforum>
    <Posts/>
    <Comments/>
  </swforum>
</bus:add>
```

6.3.2 Application Code

Like many applications, the *swforum* application has parts to validate input, write updates and links to the database, and finally update the user interface. The significant difference with the BUS approach is in the user interface update method. To update the user interface, we must fetch new UI data and update the BUS, before sending the rendering request.

The high level parts of this web application is displayed in Figure 6.3, and we can see the structure of the application. The *buscli* class (imported in line 1) handles the common BUS client functions for Python BUS applications (such as reading in and sending an XML BUS operations file with the name "swforum.xml"). The "initac" method (line 15) is called at the start by buscli, and this method opens a data service link (line 16) , and sets the this class's path in the BUS object structures (line 17).

```
 1: from acfw import buscli, runsvc
 2: import sys, os, urllib, cgi, time, traceback
 3: from uniobj import *
 4: import simwebdb
 5: from dataobj import dataobj
 6: from xmlobj import xmlobj, xmltable

 7: xo = xmlobj( 'data' )
 8: err = sys.stderr.write
 9: uesc = urllib.quote_plus
10: dbesc = simwebdb.dbesc
11: hesc = cgi.escape

12: class swforum( buscli ):
13:     title='Simple forum application using the BUS'
14:     appname='swforum'

15:     def initac( self ):
16:         self.db = simwebdb.dataservice( 'InfoWeb', passwd, 'SimpleWeb' )
17:         self.branch = '/%s' % self.appname

18:     def bus_cgi( self, obj ):
19:         # assume init has built structures and components
20:         try:
21:             sysdata, userdata = self.processevent( obj )
22:             self.dbupdate( sysdata, userdata )
23:             self.updatebus( sysdata, userdata )
24:             self.render()
25:         except:
26:             traceback.print_exc()
27:             self.procerror( 'Error: %s' % self.appname, obj )
28:     def processevent( self, obj ): STUB
29:     def dbupdate( self, sysdata, userdata ): STUB
30:     def updatebus( self, sysdata, userdata ): STUB
31:     def render( self ): STUB

32: if __name__ == '__main__':
33:     runsvc( [ swforum ] )
```

Figure 6.3: Program listing of the prototype web forum application.

The code in "runsvc" will wait on a user event from the BUS. An event will call the buscli handler which in turn will call the bus_cgi method (line 18) in our application (if the event came from a web browser). The bus_cgi method interprets and separates the event data into system and user data (line 21), then updates the database if required (line 22), before sending new data to the BUS (line 23) to service a new display request, and finally commanding the BUS to render (line 24) the required presentation object with the correct

chain of data objects. The application will then go back to waiting for another user event inside the "runsvc" function.

```
 1: <view AppName="swforum" HostUrl="/bus.html" Table="Item"
 2:    urlfmt="/bus.html?appname__=swforum&objtype__=Item&Id=%s&act__=%s">
 3:    <heading>
 4:       <column fname="Id" />
 5:       <column fname="Name" />
 6:    </heading>
 7:    <dataset>
 8:       <item Id="4">
 9:          <data value="4" />
10:          <data value="Summary of Sydney progress meeting" />
11:       </item>
12:       <item Id="6">
13:          <data value="6" />
14:          <data value="New management agenda?" />
15:       </item>
16:    </dataset>
17: </view>
```

Figure 6.4: Example of the data update format for the *swforum* application.

The `updatebus()` method above is responsible for keeping the data object structures inside the BUS current. A change in the presentation object selection will require `updatebus()` to send `<bus:del>` and `<bus:add>` messages with new data objects. The data update object (see example in Figure 6.4) will match the data object structure that the presentation object is designed to accept.

6.3.3 Template code

The application appearance is defined in the presentation objects that the *swforum* application sends to the BUS. In Figure 6.5, the design of the component which displays a listing of posts or comments is shown. This is one of the components that was sent to the BUS when the application started, and could be modified in real time if required, but most applications modify the *data objects* and leave the presentation objects in their initial state.

```
 1: <SetVar name="urlfmt" value="$ data.urlfmt $">
 2:   <table class="list">
 3:     <FOREACH seq="match">
 4:       <tr class="heading">
 5:         <foreach>
 6:           <th class="list">$ data.fname $</th>
 7:         </foreach>
 8:       </tr>
 9:       <ForEach>
10:         <tr class="list">
11:           <foreach>
12:             <td class="list">$ data.value $</td>
13:           </foreach>
14:           <SetVar name="action"
15:             value="$ var.urlfmt % (data.get('Id','NULL'),'%s') $">
16:             <td class="actions">
17:               <a class="button" href="$ var.action % 'delete' $">Delete</a>
18:               <a class="button" href="$ var.action % 'copy' $">Copy</a>
19:               <a class="button" href="$ var.action % 'edit' $">Edit</a>
20:             </td>
21:           </SetVar>
22:         </tr>
23:       </ForEach>
24:     </FOREACH>
25:   </table>
26:   <div class="navblock">
27:     <a class="button" href="$ var.urlfmt % ('NULL','new') $">Create New</a>
28:     <a class="button" href="$ var.urlfmt % ('NULL','dblist') $">Category List</a>
29:   </div>
30: </SetVar>
```

Figure 6.5: Example of a general purpose entry list component.

Analysing the components structure, we see a hierarchy of XML elements with some content and attributes computed at display-time (expressions delimited by dollar signs). The outer element (in line 1) copies an attribute (the format string for URL requests to the host application) from the current data item (see Figure 6.4 for an example of the data structure) and assigns it to **urlfmt** which is visible anywhere in the scope of the **SetVar** element. The single child of the **SetVar** element is a table (line 2) — a standard HTML object.

On line 3, the active element **FOREACH**[1] with a sequence attribute defined to **match** allocates one child of the current *data* structure to each of the child *presentation* elements of this node for rendering.

Continuing to reference the data objects from Figure 6.4, the rendering now builds the table contents. The **tr**(table row) element on line 4 will then render in the context of the **heading** data object in this example (and the **ForEach** active element on line 9 will use the **dataset** data object). The **foreach** element (line 5) in the **tr** element (line 4) will then loop through each child data object and create a **th** (table heading) element (line 6) with the content supplied by the data object's **fname** attribute. In a similar way, the **ForEach** on line 9 loops through its data objects and creates a table row (using line 10) which renders the **td** elements (on line 12) with the values from the grandchildren objects.

The parts in lines 14 to 21 require more explanation. The **SetVar** element initialises the string URL template with the **Id** field from the data object, then creates the options for operations on this row (which in a more realistic example could be different for each row). Three hyperlinks are built and displayed with delete, copy, and edit actions. The final **href** attribute will be similar to:

/bus.html?appname__=swforum&objtype__=Item&Id=4&act__=edit

The `/bus.html` part identifies the web resource that acts as a gateway for this instance of the BUS. The `appname__=swforum` instructs the BUS on which application is to receive this user event message (this may not be the same application that created this web page). From the remaining fields, the *swforum* application can determine that the user wishes to edit a data item of object type **Item** and a identity number of *4*.

The **div** block on lines 26 to 29 provide two other options for the user: a request to create a new posting, and a request to list all posting categories. These hyperlinks are built with the same method as above.

[1] Upper or lower case letters do not matter in active tags or HTML tags. Different usage of case is used here to aid the reader matching opening and closing XML elements

6.3.4 Template script

The declarative XML language is human readable, but is not efficient or reliable for humans to write directly. Each computer language has a library to build XML streams from an object tree or calls to an API, and this is an effective way to build and maintain XML specifications; however humans are often required to manipulate the XML directly. Various XML based WYSIWYG editors are available which assist the human operator in building compliant XML and detecting errors.

Another option is using a more concise language with extensions to short-cut repetitive operations. The DMT (Dynamic Markup Template) syntax was produced to support the building and editing of a simpler template file and output the equivalent XML file, shielding the user from learning and using XML directly. An example usage is shown in Figure 6.6 which produces an XML file equivalent to the gplist.xml file shown in Figure 6.5.

```
1: $input data urlfmt style=list
2: table ~$style
3:     $match $data
4:         tr
5:             $loop
6:                 th $.fname
7:         $loop
8:             tr
9:                 $loop
10:                    td $.value
11:                $action = ${ $urlfmt % ($(.Id else 'NULL'),'%s') }
12:                $macro alink = a ~button href=${action % $1} $1
13:                td ~actions
14:                    $alink( Delete )
15:                    $alink( Copy )
16:                    $alink( Edit )
17: div ~navblock
18:     a ~button href=${ $urlfmt % ('NULL','new') } Create New
19:     a ~button href=${ $urlfmt % ('NULL','dblist') } Category List
```

Figure 6.6: Example of the Dynamic Markup Template language.

The language is still declarative and hierarchical, but uses indentation to signify the hierarchy (like Python) instead of closing tags, and the use of macros and syntactical shortcuts reduce the need for repetition and special punctuation letters. This language is designed

to be used in specifying configuration files in the application component server, where BUS operations and object definitions are maintained. More information about the DMT language can be found in appendix C.

Other web applications in this set of experiments have a similar design, but instead of explaining the detailed operations of the application component again, I will concentrate on the new features being demonstrated.

6.4 Situation Display Experiment

The Situation Display is an application that helps an information worker understand the history, current configuration, and plans and capabilities of deployed resources, opposition resources, and the environment. The application is typically map-based and is overlaid with layers of vector objects and labels. Mouse actions on visible objects allow the user to *drill down* into details.

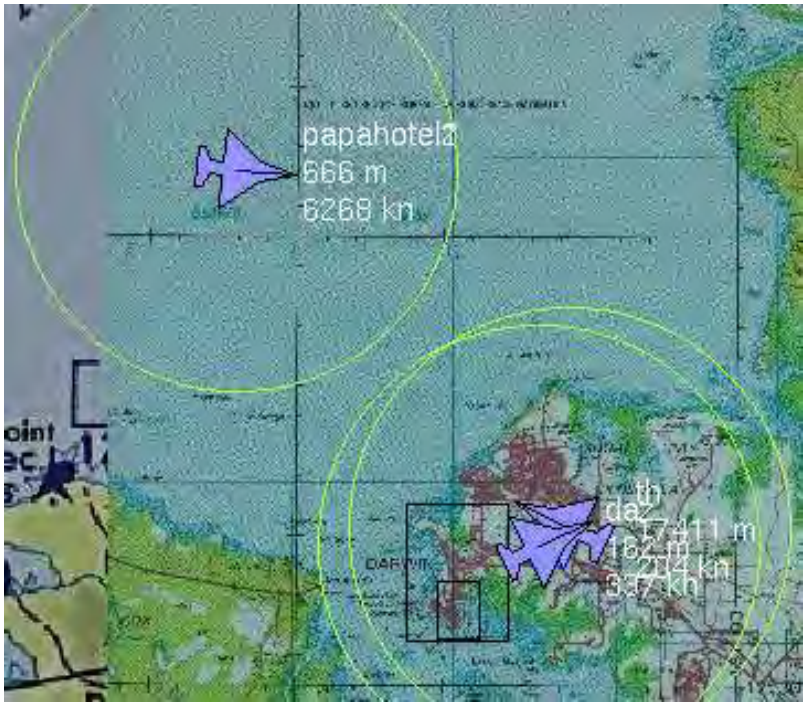

Figure 6.7: Simple Situation Display.

I designed an experiment that explored how the GGA component and the BUS could be used to implement a simple Situation Display. The application used an aircraft track display on a geospatially registered map image. A screen shot of part of the display is shown in Figure 6.7.

The map background was built from dynamically resized map images, overlay order being determined from scale detail. The position and breadth of each map was stored in meta-data embedded in its filename, then the SA application selected and transformed map images to suit the viewport the user had selected. If a higher resolution map segment was available, but too small to effectively display, the application rendered a black rectangle instead of the transformed image.

The active elements in the display were aircraft positions. The symbols were created with filled polygons, the track labels with text commands, and the radar range limits with simple circles. Each set of aircraft objects was assigned to a group so move commands would act on the set of objects, but still leave the application free to update aircraft heading (rotating the polygon object) and track label (replacing the text object). The application was configured so that a mouse double-click would show detailed aircraft track information in an associated named browser window.

The prototype application component linked dynamically with the BUS, simultaneously maintained other connections to data services, and updated the user interface in real time as new data became available. All user interfaces in the session were updated with changes to aircraft tracks using the same session presentation objects and data object tree. Other user controls were configured on a connection basis, which allowed users to pan to different areas of the map for instance. Several sites around Australia used the web application as air track updates were generated and enhancements were made to the live system.

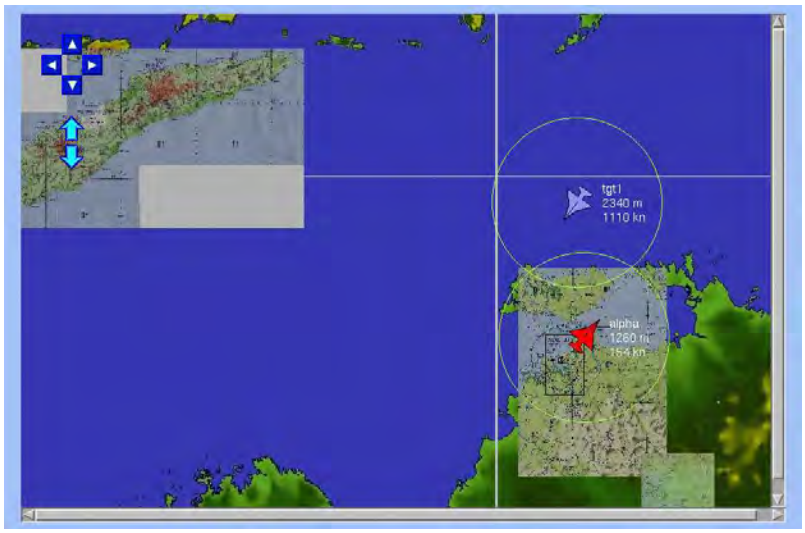

Figure 6.8: A Situation Display with mapping and controls

Enhancements (see Figure 6.8) were easy to implement, as the BUS functions and GGA component remained unchanged. A complete tiled image map of the world was added using meta-data embedded in filenames with no change to the application. The addition of panning controls involved some simple changes. The four direction buttons were added to the surface of the GGA display with new presentation objects:

```
<bus:add ctype="gga" dest="sitapp/SESSION" offset="after"
    connect__="CONNECT" session__="SESSION">
  <image fixed="YES" img="/icon/right.png" x1="50" y1="40" ID="PanEast"
    click="YES"/>
  ..others..
</bus:add>
```

The objects are specified in an application UI specification file, so a UI designer can change the appearance of the UI without possible changes in application behaviour. Note that the buttons are *fixed* so they will not pan when the display is panned. If the user clicked on the button, the application was sent a message of the user event in the form:

```
<event name="PanEast" action="click" x="4" y="7"/>
```

Initially the application saw no significance in this user event and ignored clicks on this new button. I then modified the event handler in the application to send a pan presentation object to the *connection* structure (not the *session*), and render the change to the User Interface. A pan command to the *session* would cause ALL attached components viewing the situation display to pan. The BUS operation to add the presentation object to the structure is:

```
<bus:add ctype="gga" dest="sitapp/CONNECT" offset="replace"
    connect__="CONNECT" session__="SESSION">
  <pan direction="e" amount="50%"/>
</bus:add>
<bus:render ctype="gga" dest="sitapp/CONNECT" dataset="sitapp/global"/>
```

These changes were made while users were *using* the application. The application state save, restart, and rebinding of the application component to the BUS were transparent to the user. Users across Australia just noticed richer maps appear and extra buttons that allowed them to pan the display. Many simple changes were made this way, in minutes instead of days.

The combination of features in the BUS and the GGA allowed this web application to be rapidly built, to demonstrate basic collaborative session features, to provide an easy-to-use display, and allow quick and effective enhancements.

6.5 Lightweight Collaborative Experiment

Another prototype investigated the interactive multi-user aspects of the architecture with a simple geospatial media application. A screen snapshot of this prototype is shown in Figure 6.9. The application maintains a set of icons located on a map with associated URL data, providing an awareness of web resources related to world locations. The application tracked multiple users all sharing the same session, and updated displays in real time as users added or moved information.

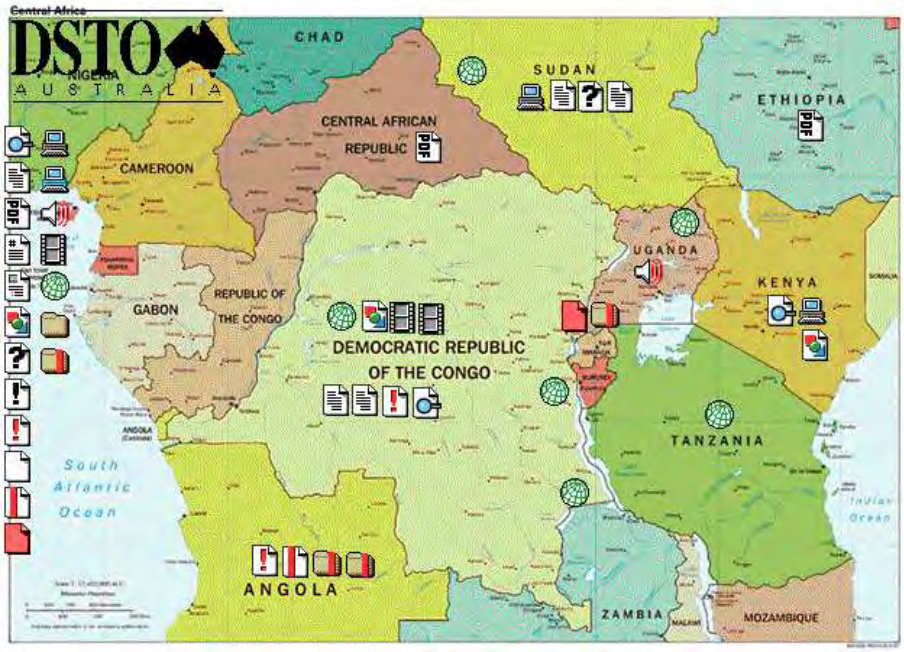

Figure 6.9: Simple File Map Tool.

A palette of web media types was available on the left of screen to build displays. To create a new resource, the user dragged an icon into a position on the map. The icon was immediately replaced on the palette, and the user was free to update the new resource. The user could click on the icon to view or edit the linked data in a browser form, or double-click on the icon to open the content in an attached browser window.

The database structure was a single table that stored one record for each resource in the application:

```
1:  map VARCHAR –eg: africa.gif
2:  name VARCHAR –eg: Economic Summary
3:  url VARCHAR –eg: http://www.worldtrade.gov/report/WE4456.html
4:  x INT –eg: (x,y) is location on map from top-left
5:  y INT
6:  mtype VARCHAR –Media type eg: gif,html,avi,pdf,svg
```

The application logic was equally simple:

```
1:  Handler( event ):
2:    if action==grab:
3:      if ID in palette:
4:        Restore palette icon
5:    elif action==drop:
6:      Update db entry
7:    elif action==click:
8:      Open form for ID
9:    elif submit:
10:     Store form to db with ID (without x,y)
11:     Update URL in object ID
```

Note that the doubleclick event is handled internally to GGA. Once the GGA objects have a link assigned, a doubleclick on that object will fetch that URL resource. The algorithm is simple and logical. The complexity of collaborative applications is handled inside the architecture. Conflicts are automatically resolved to align with the last event.

The web application was easy to build in this new environment and gave users a collaborative information tool that would require a major development effort using traditional development tools and components. The built-in drag-and-drop, click, and double click support made the application algorithm straightforward and low risk. The transparent collaborative support required no coding support. The whole application was built in a few hours, which represents high productivity for a multi-user graphical web application.

6.6 Geospatial Integration Experiment

The OpenMap application is an open source Geospatial Information System (GIS) application written in Java. Application developers can write a custom *layer* using a supplied API, and integrators can configure the application to run a set of applications (layers) together in one OpenMap application. The user can switch available layers in and out, and mouse interactions are cascaded down through layers until one layer detects an object that it manages at that location and captures the event. This approach to integrating applications is novel, but extensive learning, development, and debugging is required to get each application layer to work as required.

A version of the GGA applet was designed as a general purpose UI layer for the OpenMap geospatial Java application. It is called the GOL (Generic OpenMap Layer) [2] and it interfaces with the OpenMap application via the layer API provided by BBN. The GOL is not an applet, but a Java class that extends an OpenMap base layer class, and it implements a number of interfaces so it can communicate with the host application and a BUS server.

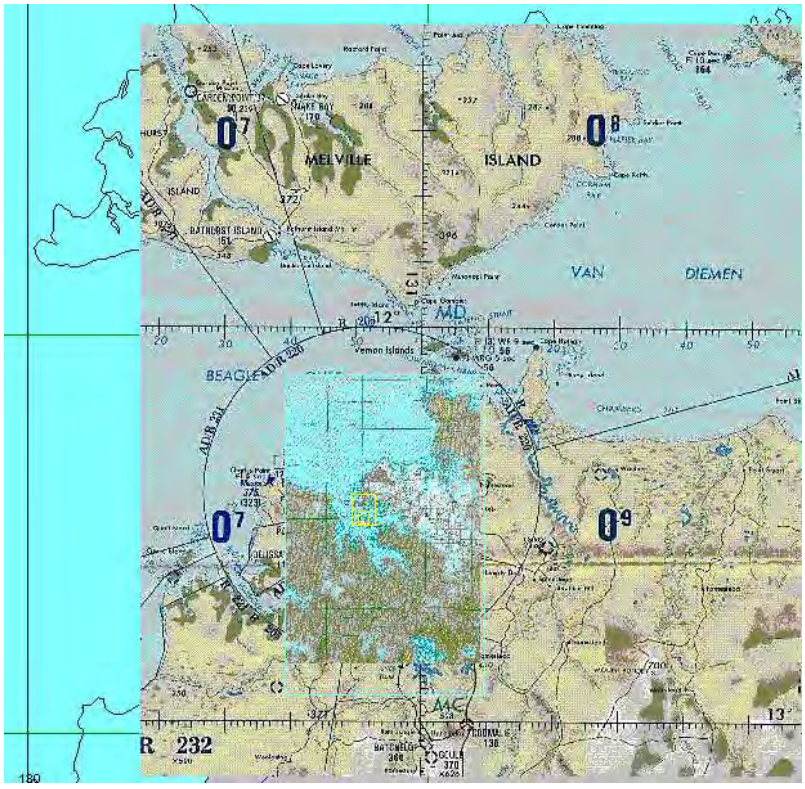

Figure 6.10: Maps displayed with GOL in OpenMap application.

[2]The GOL was implemented in Java by Peter Hoek from DSTO. I designed and tested the GOL, and provided the server applications.

Using the GOL, multiple layers dynamically connect with the BUS, and application components have been able to show raster maps at various resolutions and show moving radar tracks on client displays. In Figure 6.10, a cascading set of geospatially linked raster maps are overlaid on a Digital Chart of the World (DCW) dataset. User events on layer objects are sent to the correct application component through the BUS, and the BUS generates complex geospatial components as required.

An example of the GOL command set is shown in Figure 6.11. The commands create an aircraft track over a map background. The command in line 1 loads a processed map image to location -15.43 degrees latitude and 132.10 degrees longitude. Lines 5–9 set the defaults for commands that follow. A blue circle representing the range of a radar is drawn by the command in line 10, and the track path is shown by a light blue multi-segment line is drawn at line 12. A circle for the tracked aircraft and a multi-line label are created by commands in line 17 and 19. The final command ensures that the map image is moved underneath the vector graphics objects.

```
 1: load image "http://adder/hiat/cgi/imgproc.cgi?
 2:     f:zoom=13&q:file=../map/www.lib.utexas.edu/
 3:     map11.22s14.01s132.28e130.28e1274x1812.jpg"
 4:     at -15.43,132.10 deg using topo_layer notify click false
 5: set font fontname sanserif fontsize 14
 6: set notify all false
 7: set notify click true
 8: set attribute linecolour "FF0000" linewidth 1 fillcolour "FF00FF" filled false
 9: set attribute offset 0 0
10: draw linecolour "000099" linewidth 1 circle -13.8,132.4 to -14.2,132.8
11:     using trackradar2
12: draw linecolour "3333DD" polygon
13:     -14.5,132.35
14:     -14.3,132.2
15:     -14.0,132.6
16:     using trackpath2
17: draw offset -8 -8 linecolour "0000FF" linewidth 2 circle -14.0,132.6 size 16,16
18:     using trackhead2
19: write draggable offset 12 0 anchor w linecolour "FF0000" text
20:     "TX2-y\nFA-18\n4120m\nseeking"
21:     with fontname 10 at -14.0,132.6 using tracktext2
22: change topo_layer layer to bottom
```

Figure 6.11: Example commands for OpenMap GOL-based application layer.

Note that a CGI image processing program is called to zoom the image to 13% of its size. This is an example of the high flexibility that an architecture can have when based on the HTTP protocol and use URL requests returning MIME content. Any hyperlink can fetch dynamic content or signal another web application.

The `notify click true` default setting requests OpenMap to send any mouse clicks on these objects to the layer server as a user event message. The server associates mouse clicks to application behaviour. As a result, the server may (for example) send a command to the OpenMap layer to change *trackpath2* to red:

```
draw linecolour "FF3333" polygon
  -14.5,132.35
  -14.3,132.2
  -14.0,132.6
  using trackpath2
```

and internally store state that the user has *selected* this object. Future mouse gestures or button presses may cause other internal and UI actions with *trackpath2* as the subject.

The GGA command set needed to be extended to suit the geospatial component. The location and size parameters could now be specified by a degrees value in *nnn.mmmm* format or with standard integer pixels. Objects with a Latitude-Longitude location are said to be *georeferenced* and move with map projection changes and panning. These projection changes are sent from the applet to the server as another part of the spatial extensions:

```
  projection change BN,BS,BW,BE,WIDTH,HEIGHT,PROJ\n
```

Objects with pixel-based locations are not linked to the map. If an object also has a size based on degrees, it is *geotransformed* and changes both location and size with changes to the linked map view. Objects may also be allocated an *offset* which allows the display designer to move objects a fixed distance from a point to aid in clarity (eg: a city labels can be offset from the cities they are related to).

The GOL brought the benefits of a general purpose UI component into the OpenMap application world. Once the GOL was developed, it was loaded on the machines used for the experiment and small applications were developed on the server side. If changes

were needed, the small server application component scripts were changed and tested, then the component was restarted. The running machines noticed an improvement in the application with little development effort, deployment hassles, interruptions to the user experience, or risk of hidden faults crashing the application.

We can contrast this with the experience of the other application developers. They employed a variety a Java development environments and tools. Each complex application layer needed to be changed and tested. Then client applications must be exited and the new class libraries installed and configured, before clients are able to restart the applications. The frequency of errors was also higher in these layers, due to the mixing of UI and application logic. To create collaborative features, the traditional developers needed an elaborate multi-threaded RMI architecture, where the GOL application component handled collaboration internally. There was also a benefit in encapsulation, where the GOL programmer was an expert in Java and the OpenMap API but needed to know nothing about the Python application on the server. The application programmer need only know the GOL language, and know nothing of Java or the complexities of the OpenMap API and platform dependencies.

6.7 Geospatial Applet Component Experiment

Another experiment built[3] the OpenMap Java application into a Java Applet with communication and integration APIs similar to the GGA. This development resulted in a complete GIS tool deployable on the web, having integration hooks into the surrounding web page, and binding with a remote BUS server for connection with application components. The OpenMap Applet (OMA) operated in a similar way to the OpenMap application described above, however each layer operated as a GOL except that layers shared the communication socket with the server. The resulting applet was much larger than the GGA, as it contained the OpenMap application, the GOL code, and included GUI functionality from the Java Swing library.

[3]The OMA was built in Java from my specifications by a DSTO contractor.

The applet is embedded in a web page using the standard HTML markup:

```
<applet name="om" code="dsto.om.OMApplet" codebase="/hiat/applet/openmap"
  archive="omapp.jar" width=640 height=480>
</applet>
```

After loading and initialising, the applet may connect to a server if configured, or take commands directly from the user interface or web page use of the applet methods. Required GIS datasets are loaded from URLs using standard HTTP transactions.

```
 1: om_load graphics.class as gfx
 2: om_select gfx


 3: # A few fixed objects:
 4:    draw circle at 20,20 radius 50 using c1
 5:    draw linecolour 00ffc0 rectangle 20,20 to 250,350 using r1


 6: # A two degree circle around darwin:
 7:    draw linecolour ff4040 circle at 130.9:-12.4 radius 1 deg using DarCirc


 8: # A yellow line from darwin to brisbane:
 9:    draw linecolour ffff00 line 130.9:-12.4 to 153:-27.5 using DarBrisLine \
10:       add link http://adder/hiat/doc/ioa_api.html into DataWindow


11: # City nodes:
12:    set attribute linewidth 1
13:    set attribute linecolour ffffff
14:    set attribute fillcolour 202080 filled true
15:    set font style italic fontsize 10 fontname sansserif justify center
16:    set notify click true drop true drag false
17:    set attribute anchor c offset 0,0
18:    draw circle at 115.8:-31.9 radius 5 pix using c1a
19:    write anchor e offset -10,0 text "Perth\n117-88" at 115.8:-31.9 using c1b
20:    draw circle at 151.2:-33.9 radius 5 pix using c2a
21:    write anchor w offset 10,0 text "Sydney\n154-02" at 151.2:-33.9 using c2b
22:    draw circle at 147.3:-42.9 radius 5 pix using c3a
23:    write anchor n offset 0,10 text "Hobart\n201-23" at 147.3:-42.9 using c3b
24:    draw fillcolour ffa0a0 circle at 149.15:-35.3 radius 5 pix using c4a
25:    write anchor se offset -10,-10 text "Canberra\n195-91" at 149.15:-35.3 \
26:       using c4b
```

Figure 6.12: Example initialisation data file for an OpenMap Applet application.

The applet contains all of the native OpenMap functionality, so may also be used without integration with the web page environment or an application component on the server,

but the strong architectural features are realised with UI integration to browser resources and connection to one or more application components.

The OMA applet used a similar command language (see Figure 6.12) except that the application needed to identify which layer it was addressing before sending commands. Events coming into the server were also annotated with the name of the layer that the event came from.

```
setProjection(PROJ)
   where PROJ = Mercator | CADRG | Gnomonic | Orthographic
   eg: onClick="om.setProjection( 'CADRG' )"

zoom( MODE, AMOUNT )
   eg: onClick="om.zoom( om.RELATIVE, 0.5 )"

pan( DIRECTION, AMOUNT )
   eg: onClick="om.pan( om.SOUTH_WEST, 10.0 )"

layers( LAYERS )
   where LAYERS = DayNight | Graticule | Countries | Cities | Lakes
   eg: onClick="om.layers( 'Graticule' )"

setCenter( LAT, LONG )
   eg: onClick="om.setCenter( -9.65, 109.04 )"
```

Figure 6.13: Examples of OpenMap Applet Javascript commands.

New web page integration methods were needed to support the geospatial functionality of the OMA (see Figure 6.13). These new methods allowed HTML buttons and other Javascript sources to change the configuration of the running applet.

The conversion of the OpenMap application to a UI component compatible with the presentation service concept opened up new possibilities for geospatial web applications and added to the adaptability of the BUS and general-purpose browser component architecture. The mapping support of GGA has been limited to images of raster maps with objects placed at computed locations based on a non-realistic flat earth projection. The OMA offers a complete GIS component that performs geospatial projections and transformations, with the added benefits of web-based deployment, browser integration API, and UI-application separation with BUS technology.

6.8 Virtual Reality Integration Experiment

To extend the BUS and UI component architecture into a three dimensional UI, a number of protocols and tools were considered. The VRML markup language was a standard for 3D work on the web, but not all browsers were equipped with the required plug-in, and web page integration and server interaction had limited and non-standard functionality.

To build a 3D UI capability that would work with all browsers and the BUS technology, I designed an applet that would include a 3D library and a communication language that supported the building and transforming of 3D worlds, and the capture and reporting of user events. The Spatial 3D Applet (S3DA) prototype applet was built[4] around the *Anfy* 3D library and an XML language designed for server interaction.

```
<message> CMD* </message>
   CMD is one of:
   <Style {ATTRIBNAME=VALUE} name=ID />
   <Image bn=LL bs=LL be=LL bw=LL url=URL name=ID alpha=INT />
   <Line n1=LL e1=LL a1=LL n2=LL e2=LL a2=LL alpha=INT
        style=STYLE_ID colour=COLOUR size=INT name=ID />
   <Symbol n1=LL e1=LL a1=LL pitch=DEG yaw=DEG roll=DEG alpha=INT
        ref=SYMNAME style=STYLE_ID colour=COLOUR size=INT name=ID />
   <Camera n1=LL e1=LL a1=LL pitch=DEG yaw=DEG roll=DEG name=ID />
   <Change target=ID {ATTRIBNAME=VALUE} />
   <Delete target=ID {ATTRIBNAME=VALUE} />


where:
   LL = Float value of displacement
   ID = Identifier in form [A-Za-z][A-Za-z0-9_]*
   DEG = Decimal degrees
   COLOUR = Integer in signed decimal format:
        Bits 0-7 Blue, bits 8-15 Green, bits 16-23 Red
   SYMNAME = Plane | Ship | OTHER
   ATTRIBNAME = Object attribute identifier name
   VALUE = String attribute value
```

Figure 6.14: Message definition for S3DA input.

The S3DA message input options are shown in Figure 6.14. For this experiment, I attempted the use of an XML API instead of the structured English API used in the other applet languages. This had the advantage that the language mapped directly onto the

[4]The S3DA applet was skillfully coded in Java by Peter Hoek at DSTO from my specifications. Considerable testing and refinement was needed before we were able to reliably run the component with the server interface. Peter also assisted in the conduct of this experiment.

BUS presentation object design, but it was difficult to read easily. The language supports a minimal set of 3D objects, but sufficient for the exploration of the 3D UI component concept.

The applet supports the loading of image tiles to form base maps, and a flexible generation of symbol objects and lines to create information displays on the maps. The applet will determine objects indicated by mouse gestures and return these events to the server for application logic actions. A camera object allows applications to move the user viewport through the 3D space, controlling position, height, zoom, pitch, and roll.

```
<message>
  <Change target="Camera" n1="3.55" e1="14.11" a1="2.45"
      pitch="-71.6" yaw="118.2" roll="0.0" />
</message>

<message>
  <Change target="Camera" n1="3.51" e1="14.12" a1="2.45"
      pitch="-71.4" yaw="117.9" roll="0.0" />
</message>

<message>
  <Change target="Camera" n1="3.47" e1="14.13" a1="2.45"
      pitch="-70.9" yaw="117.5" roll="0.0" />
</message>

<message>
  <Change target="Camera" n1="3.42" e1="14.13" a1="2.45"
      pitch="-70.5" yaw="117.2" roll="0.0" />
</message>
```

Figure 6.15: Flying POV example using the S3DA component.

A fly-through model was evaluated. A series of camera coordinates was produced by a server (see Figure 6.15), and sent to a number of S3DA applets in the session at regular time steps. This produced a distributed fly-through of a simulated battlespace[5], providing decision makers with an appreciation of the conflict area from different vantage points.

This UI component further extends the applicability and flexibility of the BUS and component UI architecture. The S3DA can co-exist with OMA and GGA instances in the same web page or across the country, sharing sessions and interacting with shared data.

[5]For security reasons, screen images and code examples cannot be included in this thesis

6.9 Evaluation of Experiments

This set of experiments demonstrated simple development designs, user interface encapsulation, and high useability displays. I will now evaluate what properties of the web application domain were enhanced by this new approach, and which features of the technology was responsible for the enhancement. I will also dissect problems that occurred during development, isolate the causes, and suggest solutions.

If the resources were available, it would have been useful to have a programmer experienced in another web application technology develop applications to an identical specification. It would also be a useful experiment to have a web application programmer use the BUS and GGA technology without an understanding of its internal mechanisms. These experiments would permit a detailed comparison of the utility of this new web technology. Of course, it would be difficult to draw reliable conclusions from single samples, especially using only toy web applications.

As resources and time were not available, my approach is to analyse the experimental prototypes in the framework of the desirable web properties stated in Chapter introduction. This analysis is somewhat subjective, but each claimed property enhancement can be traced back to a feature designed into the BUS and GGA technologies. Many of these features have also been demonstrated in the above experiments.

Separation of Concerns: The above experiments demonstrate the separation of the web application into: the browser user interface device augmented with the GGA; the web server with BUS Gateway, applet deployment and passive content storage; the BUS for the presentation service engine; the application components to implement business logic; and a database for application data persistence.

A separation of concerns is also used inside the BUS. The BUS internal design uses independent modules such as *UI Managers* to control and convert UI protocols, *Application Managers* to communicate with application components, and a *Control Manager* to service management software. The BUS also maintains a clear division between presentation logic and data objects, up to the point where the UI is rendered.

Simplicity: Basic XML syntax is used with XPath expressions to communicate with the BUS. The presentation object semantics are based on familiar HTML elements, basic

graphic objects, and dynamic elements based on simple program-control concepts.

Modularity: The BUS is designed as loosely coupled objects, cooperating to produce sophisticated functionality. This modularity allows new UI Managers to be installed, additional functionality added to an API, or a new presentation object type to be added without unintended side-effects.

Familiarity: From the user's point of view, the only change from traditional web applications that require new skills will be the inclusion of interactive graphics in the user interface. However, application designers that use familiar and intuitive graphics UI dialogue will assist in providing a familiar and comfortable environment for the user.

The developer using the presentation service and GGA component will be in unfamiliar territory though, and need to adapt their development style and application architecture to make use of this new technology.

Learnability: The interfaces of the BUS and GGA are straight-forward, orthogonal, and readable, which eases the learning task for new developers.

Useability: The useability of these new web applications is enhanced by the graphical representation of data to aid understanding, the interactivity supplied by the GGA component, and the composite UI web displays possible through referencing object structures in the BUS.

Consistency: The abilities to reuse presentation and data material in the BUS encourages consistent use of existing and externally supplied objects. A single graphics component will provide consistent behaviour in all graphical display roles (such as drag and drop, scroll bars, and a double-click to open content).

Orthogonality: The dimensions of presentation, data, and application logic are independent in the new design. One aspect can change without influencing others. In fact, creators of data, presentation, and applications do not need to know of each others work. The integration of these three aspects are performed inside the BUS.

Interactivity: The BUS makes no direct contribution in this area, however the GGA adds many enhancements for interactive web applications. The asynchronous nature of the protocol allows the UI to respond directly to user actions, web page events, and state changes in the BUS and the application.

Multi-media: The capabilities of the GGA adds basic graphics components to enable the creation of charts, maps, and diagrams. These displays can include pictures and text, and be overlaid with other graphics and annotations. This graphics capability fills the gap in the HTML set of presentation components.

Continuity: The BUS adds some service continuity support, as it will continue to display existing user interface content from internal presentation and data stores, even if the responsible application is temporarily unavailable. The BUS is, however, a critical component for an architecture using this technology, and a failure of the BUS will paralyse all user interfaces.

Flexibility: The BUS and GGA are independently flexible components, and together they provide extensive flexibility. The full range of dynamic web interfaces augmented with interactive graphics is available.

Customisability: The BUS and GGA do not enhance this attribute of web applications directly. The ability for users to customise the user interface is left to application components.

Productivity: The GGA encapsulates the functionality for implementing interactive graphics. The BUS encapsulates dynamic web resource management and event reporting. This removes some the complexity from web application development, enhancing the productivity of the programmer.

The user productivity is also assisted. Using the BUS, developers can more easily build user web views that are assembled from various presentation components from different applications.

Configurability: Base presentation and data objects are supplied in the BUS configuration file. Changing this file can change the appearance and behaviour of web applications without code changes to the BUS or the application components.

Collaborative: The support for collaborative applications is one of the strongest features of BUS and GGA technology. The examples above illustrate how easy it is to build a small web application that supports many cooperating users. The BUS can also support collaborative sessions between different UI technology clients which offers new possibilities of sophisticated applications with minimum programming work.

Adaptability: The BUS is designed with adaption in mind. New UI Managers can be integrated with a minimum of coding and a minimum of internal complications. Each Manager module is a separate object using simple interactions with other objects, so behaviour can be easily extended (to implement authentication for example).

Clarity: The BUS and the GGA use many interaction methods and APIs, but each protocol and language is self-contained and logical. It can be seen from the experiments above that the simple BUS operations, presentation objects, and data structures are logical, uncluttered, and readable.

Reuse: The BUS is designed to extract maximum reuse from existing and externally managed presentation and data objects. These objects can be copied and modified by BUS operations, or referenced during rendering. The BUS and GGA themselves are general-purpose reusable components, able to be employed in many web application designs.

Inheritance: The BUS objects implement a type of *prototype inheritance* where objects are copied then attributes can be overloaded and sub-objects changed. The GGA graphics components uses a weak form of inheritance with defaults providing base object attributes.

Encapsulation: Both the BUS and the GGA are designed to encapsulate user interface functionality. The prototypes above demonstrate the use of the BUS and GGA APIs to generate user interfaces using minimal effort.

Language Independence: The GGA can communicate with applications developed in any language using simple strings. The BUS has XML APIs which are also language-agnostic. This technology can merge presentation and data from many types of application environments in many languages simultaneously. The UIs from a Perl application, a PHP application, and a Java application can be seamlessly joined in a BUS managed user interface (and user events will also be sent back to the correct application).

Interoperability: This technology can interoperate with a variety of other systems and technologies via DHTML, URLs, and XML (including XHTML and RSS). The BUS and GGA can both send and receive requests using APIs.

Multiple UI Devices: The BUS is designed for any web interface, so it can display and interact with users on workstations, fixed web appliances, and PDAs. Several of the experiments above involved the development of a new UI manager for the BUS to support the new UI component. This demonstrates the ability of the BUS to be configured for new UI devices.

Open Standards: The BUS is designed around the open web standards: DHTML, TCP sockets, XML, XPath, CGI, and HTTP. The GGA is a standard Java applet compliant with DHTML standards. No proprietary protocols have been used in the design of the BUS and GGA.

Multiple APIs: The BUS has multiple APIs: URL, HTML, XML, GGA, Control, and Application. The GGA uses the embedded parameter commands, DHTML interaction, and the socket connection to the BUS or directly to an application.

Thin Client: The BUS supports standard DHTML and other MIME types without requiring any change in the configuration or software installation in the client hosts. The GGA is designed using older features of Java so this UI component will run on almost all browsers being used today.

Applicability: Neither the BUS nor the GGA are designed for a particular domain or segment of the web environment. Both technologies are able to be used over the full spectrum of web application designs.

Scalability: There is always a trade-off between scalability and flexibility. The BUS and GGA are designed to be very flexible, so the designs will require revision if scalability becomes the dominant factor in the usage of the technology.

Platform Independence: The GGA is designed to run in almost any browser in use today (without downloads). The BUS is developed in the Python language, which is supported on all major platforms. The majority of development and testing of this technology has been with the *Linux* operating system, the *Apache* web server and the *Firefox* browser, but no part of the technology is limited to these products.

Efficiency: This property is another victim of the trade-offs to gain flexibility and the separation of concerns. This property could be enhanced with further optimisation and development.

Maintainability: The separation of concerns, modularity, and the ability to split application functionality over many separate processes assists with the identification of faults, fault isolation, and repair without unintended side effects.

Instrumentation: The BUS *Control Port* offers an API to gather information on the running BUS process, including applications connected, transaction rates, memory statistics, and response times. This is a unique feature that offers system managers insight into the performance of a running service.

Manageability: The BUS is a single running process which is easy to deploy to any platform, and uncomplicated to run. The GGA is deployed by a web server, is cached on the client, and can be updated in a single place. This simple architecture requires only simple management procedures.

Confidentiality: The BUS and GGA offer no enhancements to confidentiality. Practical web designers would need to consider adding a web server that supports HTTPS, strong user authentication, user access control in the applications, and database encryption.

Integrity: The BUS and GGA offer no enhancements to this web application property.

Availability: There are little benefits in availability by using the BUS and GGA technology. There is in fact a weakness due to the BUS becoming a critical node that will disable all connections should it fail. This is a common problem for essential software such as framework cores, web servers, and authentication agents.

6.9.1 Discussion

The speed of developing applications is governed by the ease in mastering the development environment, the efficiency of translating business requirements into software, the availability of stable well-supported libraries of code for common functions, and the detection of errors in design and implementation. The BUS and GGA technology has been demonstrated in the above experiments to assist the developer in each of these developer support areas. These developer oriented features reduce developer effort, reduce mistakes due to complexity, and allow the creation of more maintainable web applications.

Efficiency is not a key decision factor in the choice of web application technology, but extremely poor efficiency would impact on the user experience and extra hardware costs. Runtime efficiency is to be considered for user interface response, scalability in user count and application size, and minimising server infrastructure investment. The three key factors in the BUS runtime speed are XML transformations, signal routing, and presentation rendering. The XML parsing and generation take significant time (compared to binary or flat structured protocols), but performance can be increased with the use of new high-efficiency XML libraries. The signal routing is governed by fast lookup tables and if-statements so this function is generally not a significant factor in performance. The exception here is if sending a signal causes a socket to block, possible delaying other services or clients until the request times out. This is a problem for all multi-connection servers but can be mitigated with careful refinement of the design. The presentation rendering is a complex operation, but has been observed providing satisfactory response times in a lightly loaded server. For a serious application with hundreds or thousands of concurrent users, a local presentation content cache would need to be implemented, or an external cache service (eg: the Squid cache server or Memcached) would need to be configured.

A BUS based solution offers many levels and types of integration. It's URL and XML interfaces allow it to connect to many other software systems without modification, the GGA component can be embedded in other applications, and many different types of software can be adapted to use the application component interface.

The BUS is written in the Python language, allowing it to run on almost all current hardware and operating systems. The XML based API is designed to allow connections from any programming language over the network. The user interface side allows connections from any software that can make a request via a URL, and will serve XHTML, XML, text, and images as required. This makes the BUS and GGA install-anywhere and plug-into-anything technology.

Each UI component was developed using a different architecture. The concept of a reusable UI component that could execute commands from a server and report user interface events was a common theme, but the each experiment explored different methods of achieving the goal. In Table 6.1 we see the large differences between the applet implementations.

The GOL was a very large component, due to the necessary inclusion of two large class

Table 6.1: Class statistics for the UI component applets

Applet	Size (bytes)	Class Files	Includes
GGA	157255	81	–
GOL	1600805	644	Openmap layer and java swing classes
OMA	490838	192	–
S3DA	153152	24	Anfy 3D core and Microstar XML library

libraries: the openmap layer bean and the java swing library. The OMA in contrast dynamically loaded classes under server control. The S3DA was different again. It is a component that was built using an XML based API and wrapped a simple space-optimised 3D library.

Each component serves a different purpose and can be integrated into a BUS web application design. This flexibility in architecture and deployment eases the burden of web developers and promotes highly functional, interactive, and distributed applications.

6.10 Summary

With the above experiments, I have explored several different styles of web applications using the new architecture. In the development of each prototype, I found significant benefits. Collaborative applications became easy to build. Web applications were easy to integrate with each other and with other systems. With only a small amount of application code and XML configuration files, user interfaces became highly functional and easy to change (particularly by non-programmers). It all operated on basic browser functionality, so all users could use these web applications without downloads or configuration changes. Applications could be changed transparently to the users, at low risk, without software rollout, reboots, or application server reconfiguration. The separation of web applications into five layers (browser and UI components, web server, BUS server, application components, and database and communications services) made it easy to isolate faults and easy to change an aspect of the application without side-effects. The code is event driven but without excessive layers of object factories, schema designs, complex transaction patterns, or needing a full IDE to build.

There are some costs in this new architecture that should be considered too. The use of

structured English and XML as a protocol in a multi-tier architecture introduces latency as these protocols must be generated and parsed in multiple processes during the servicing of a user event. The ever rising performance of computing power and network capacity will mitigate this limitation over time; however this performance limitation will influence the choice of this architecture for high transaction web applications.

Another problem observed in these experiments is the need to duplicate data in the BUS that is managed in the database. Each application component must track changes to the database and refresh the BUS data store so that rendering presentation components will use state information that mirrors that information in the application data store. For small items of information at low transaction rates, deleting the BUS objects and re-inserting them is an option. For the majority of applications though, this requirement to keep the BUS data objects current is the price we pay to have the advantages of *separation of concerns* and encapsulation of the user interface into a reusable service.

Of course, this architecture introduces new languages and protocols for the developer to learn, but the intention is to encapsulate some of the user interface complexity *inside* the presentation service and UI components. As a result, the developer will be able to build sophisticated web applications using a small set of simple protocols (English-like and simple XML) instead of the complex array of protocols and languages currently required.

Even taking into account the limitations above, I am confident that these experiments have proved that the web application built on the concepts of a presentation service and general-purpose UI components can offer significant improvements in development productivity, user interface quality, adaptability, interactivity, manageability, and support for systems integration.

Chapter 7

Conclusion

7.1 Summary

I set out in this thesis to investigate the problems in building and maintaining high quality web applications, to develop innovative software that would address these problems, and describe and test this new software technique.

I have proposed a unique presentation service, and described how this approach manages much of the user interface complexity. I then proposed and investigated a client-side graphics component that would provide benefits for collaboration, visualisation, and user responsiveness. The design of the BUS then explored the operation, interfaces, data structures, and protocols for a browser based presentation service integrated with the GGA component. Through several experiments with prototype software, I demonstrated the enhancements to the development processes, enhancements to applications, and benefits to the user through improved user interfaces.

7.2 Benefits of Technology

The use of a presentation service will reduce the development effort required to build new web applications. Many of the functions that are needed in each new web application are provided as part of the presentation service and new applications can reuse and adapt previously created presentation objects. This reuse not only reduces design, build, and test time, but also promotes the consistent look and feel of the user interface.

Highly interactive web applications are supported by a number of innovative features. A general purpose graphics component enables asynchronous messages to travel between browser and application component, as well as providing graphics functions within the user interface. User interface components and sessions can be shared, aiding collaborative work by immediate visualisation of changes in application state. Actions within the browser page can be configured to affect graphical objects as well as other HTML components. Javascript and other AJAX techniques can also be integrated to further enhance the interactive experience.

The presentation service provides several interfaces for integration and application synthesis. The client interface is designed for user interface connection but is also able to return HTML, XML, and other MIME types to software through a standard URL fetch mechanism. The application component interface enables any program written in any language to nominate itself as a service for presentation clients. Application components may also take a role in maintaining data or presentation structures inside the presentation service without any client transactions. The control interface is designed to allow external management tools to fetch usage parameters and statistics, and selectively modify control variables.

User interface may be constructed from the outputs of several application components *without the application component requiring change*. The data structures updated by application components are dynamically folded into rendering streams controlled by the presentation object definitions. Application components can also embed hyperlinks into data and presentation components that will allow users to open links to presentation provided by the application component owning the data. Integration into external applications can also be performed by the embedding of a descriptive URL hyperlink.

7.3 Significance

The field of web applications is experiencing strong growth, yet many aspects of web application development remain difficult, and problems remain unsolved. This work offers a software technology that has the potential to save developer programming hours and increase application quality in a software technology that is used by over 70% of the developed world.

The presentation service is a unique and useful contribution to the field of computer science. It introduces a new architecture for the development of user interface software that is simple, modular, and implements a separation of concerns.

The GGA component and the other related UI components are also unique developments that implement an orthogonal set of features to support highly interactive graphical user interfaces in web application designs. The extreme flexibility and functionality of these components are due to the product of the innovative use of layers, groups, addressable objects, and the variety of integration methods.

The combination of the presentation service and generic UI components has been shown to enhance the web application development task, and improve the functionality and useability of the resulting user interfaces. These two design technologies offer significant improvements to the run-time environment for all web applications.

7.4 Future work

The presentation service and UI component technology enhances the properties of web applications built using it; however this has only been demonstrated in small scale developments. A large scale web application development that used this technology would give researchers the opportunity to measure the positive and negative effects the technology has on the development process and the end product.

The current GGA command language requires the replacement of an object even when only one attribute is to be changed. The language could be expanded to allow attribute level operations, which would free the application from the need to keep a copy of these objects for updating purposes.

The GGA language supports general purpose graphics operations, but needed extending to support GIS functionality in the GOL and OMA components. Further research is needed to develop a language that is simple to use, but can easily extend to specialised UI component functions.

The SVG graphics language is designed for the display of vector graphics in the browser environment and, as it matures, may be a better language for graphics descriptions and

user event handling than the GGA protocol. Currently, the SVG is only partially supported by browsers, and advanced abilities such as event reporting and drag-and-drop are not standardised.

The drive for higher user interface interactivity and more attractive appearance has driven the demand for AJAX based tools and libraries. This is a cooperative rather than competitive technology to the BUS, and the highly interactive GGA design is able to work well with the Javascript behind most AJAX libraries. There is an opportunity for further research in understanding the boundaries and interfaces between frameworks, AJAX toolkits, code toolkits, SOAP web services, and the new BUS presentation service.

The BUS currently supports HTML, XHTML, XML, and text interfaces via URL, and GGA interfaces as long-running clients. Some experimental work has been done on serving dynamically generated images and GGA commands from complex presentation objects such as maps, business charts, and diagrams. The development of these new *compound components* in the BUS would enable applications to display interactive maps, charts, and diagrams by simply supplying a set of data objects to the BUS and sending a render command.

The BUS has potential for presenting applications with WML on mobile devices. This BUS extension would allow mobile user application to benefit from BUS technology, and also enable mobile UI connections to join BUS sessions to implement mobile collaborative applications. Research is required on how mobile UI technologies can be implemented in a presentation service.

The current BUS implementation uses multiplexed synchronous sockets that are efficient and simplify design; however, calls to application components that require lengthy processing times may block access for users that do not need access to that component. To counter this, the BUS could move toward a mixed design employing multiplexed synchronous sockets and application access threads that allow access to application components that block. Research is required to understand the complex transaction patterns of a service that exposes multiple styles of socket interfaces that may block.

Appendix A

Glossary

This glossary provides a short explanation of some of the technical terms used in this thesis. Where the definition comes from an external source, that source is cited. Terms in italics are also defined in this glossary.

Adaptability: The capability to be adapted to a new purpose. This may require the development of an *adaptor* to interface to the new protocol or interface. An *adaptable application* is design with loose coupling between modules, general purpose interfaces, and an architecture that allows new software to be inserted at various places in the execution chain.

Application: A computer program that is designed to accept user input, perform business logic, and return information to the user. An application is different software to a *server*, a *framework*, a *utility*, or a *toolkit* for instance.

Brittle System: A system that performs as required, but when work is applied to change it slightly, it fails catastrophically. A *flexible* system changes shape easily and still retains its core properties.

BUS Application: A long-running program that tracks multiple simultaneous sessions and is connected and registered to one or more BUS APIs. Receives user events, updates data and presentation objects within the BUS, and updates UI BUS sessions. Provides business logic, workflow, and transformation functions. May provide or connect to communication and persistence capabilities.

Class Inheritance: A reference to an abstract object skeleton that provides the object with default attributes, data, and behaviour. Relies on a shared and agreed understanding of class taxonomy.

Flexibility: For this thesis, a *flexible application* will mean an application that can be extended by users to perform an unanticipated role without extra code development or database changes.

Integratability: The ability of software to be connected to the user operating environment with a minimum of development and system administration effort. This may include the facility for this software to make requests and updates on external systems using a variety of standard protocols and transaction patterns, and the facility to accept similar requests and updates.

Maintainability: A *maintainable application* is designed to support rapid problem diagnosis, resolution, enhancements, and testing. Maintainability is enhanced by an architecture that uses modular design, loose coupling, a separation of concerns, and human-readable data structures, protocols, and configuration.

Orthogonality Properties that can vary freely without affecting or requiring changes to other properties are said to be orthogonal. This term is borrowed from mathematics and CPU instruction set design (where an instruction set is *orthogonal* if instructions can be used with most registers, contrasting with architectures that define *special registers* or *special instructions*).

Prototype Inheritance: A copy of an existing instantiated object so that it may be specialised by changes to location, contained data, attributes, and/or behaviour. Does not use abstract classes. Contrast with *Class Inheritance*.

Robustness: In this thesis, robustness is the property of software to keep functioning despite programming, system, network faults, and unexpected data input. It also covers the ability of a system to recover from faults, supporting rapid fault identification, rectification, and controlled restart using non-development personnel.

Usability: The properties of the system, perceived at the user interface, that enable users to quickly learn and use the system to meet user functional requirements. This includes a rapid response time, familiar terminology and processing sequences, clear and consistent display and controls, and reversible actions.

Web Application: An application that provides functionality to many users through a web server to web browser client software. A web application must support multiple simultaneous users and sessions, cope with high peak transaction loads and a variety of possible network errors, and be adaptable to rapidly changing requirements, protocols, and software infrastructure.

Web Application Framework: A working software structure that allows developers to "plug in" code at a number of places in the user response chain. A web application framework will typically bind to a web server (or embed a web server), interpret incoming requests, route requests to an appropriate internal handler or user code, generate content, and manage the response to the client. Frameworks will also typically detect and handle errors, keep transaction logs, and provide security protection.

Web Application Toolkit: A code library, set of modules, or programs that perform a set of well-defined functions of typical web applications. The toolkit functions may decode or encode web protocols, dynamically generate MIME content, manage multithreading, authenticate users, track sessions, or recover from errors for example.

Web Service: Software functions that are available via web request-response transactions using a published web address (URI) is considered a web service. This functionality may simply return content to an anonymous client for each request, or maintain state and use authentication and session. Typically, applications that are accessed via the SOAP protocol using WSDL and UDDI are known as web services, but most web applications can be considered a web service as the same protocols that a user browser uses to fetch a web resource can be used by other software acting as a *client* to this *server*.

Appendix B

Abbreviations

AJAX Asynchronous Javascript And Xml.

API Application Programming Interface.

ASP Active Server Pages (Microsoft dynamic web pages).

BUS Browser User-interface Service.

CGI Common Gateway Interface.

CMS Content Management System.

CORBA Common Object Request Broker Architecture.

CRUD Create, Read, Update, Delete transaction types.

CSS Cascading Style Sheets (CSS2 and CSS3 are more recent versions).

CSV Comma Separated Variable data file format.

DCE Distributed Computing Environment.

DCOM Distributed Component Object Model.

DHTML Dynamic HTML (includes Javascript and stylesheets).

DMT Dynamic Markup Template language.

DOM Document Object Model.

DRY Don't Repeat Yourself (A design principle).

DTD Document Type Definition.

EBNF Extended Backus-Naur Form (Language syntax format).

ECMA European association for standardising information and communication systems.

EJB Enterprise Java Beans (Large scale Java framework).

FCGI FastCGI (web application to web server interface).

GD A image building library.

GGA Generic Graphics Applet.

GIS Geospatial Information System.

GOL Generic Openmap Layer (A plug-in layer for OpenMap applications).

GUI Graphical User Interface.

HTML HyperText Markup Language.

HTTP HyperText Transfer Protocol.

IDL Interface Description Language.

IE Internet Explorer (Microsoft web browser software).

IIS Internet Information Server (Microsoft web server software).

IP Internet Protocol.

IRC Internet Relay Chat.

ISAPI Internet Server Application Programming Interface.

IT Information Technology.

J2EE Java 2, Enterprise Edition.

JAR Java Archive format.

JDBC Java DataBase Connectivity.

JS JavaScript.

JSF Java Server Faces.

JSON JavaScript Object Notation.

JSP Java Server Pages.

JVM Java Virtual Machine.

LAMP Linux, Apache, MySQL, and PHP/Perl/Python web application environment.

LDAP Lightweight Directory Access Protocol.

MIME Multipurpose Internet Mail Extension.

MVC Model View Controller software pattern.

NSAPI Netscape Server Application Programming Interface.

ODBC Open Database Connectivity.

ORM Object Role Modelling or Object-Relational Mapping (a software layer that transparently converts database content into host language objects).

OS Operating System.

PDA Personal Digital Assistant.

PDF Portable Document Format.

PHP PHP: Hypertext Preprocessor (recursive acronym).

RAD Rapid Application Development.

RDBMS Relational DataBase Management System.

RDF Resource Description Framework.

REST Representational State Transfer.

RGB Red Green Blue colour format.

RMI Remote Method Invocation (Java protocol).

RSS Really Simple Syndication (a web feed format).

SCGI Simple Common Gateway Interface.

SOA Service Oriented Architecture.

SOAP Simple Object Access Protocol.

SQL Structured Query Language.

SSI Server Side Includes.

SSL Secure Sockets Layer.

SVG Scalable Vector Graphics (XML graphics language).

SWF ShockWave Flash.

TCP Transmission Control Protocol.

UDDI Universal Description, Discovery and Integration service.

UI User Interface.

URI Uniform Resource Identifier (A global identifier in the context of the World Wide Web).

URL Uniform Resource Locator.

VRML Virtual Reality Markup Language.

WAP Wireless Application Protocol.

W3C WWW Consortium (Guides web standards and protocol development).

WML Wireless Markup Language.

WSDL Web Services Definition Language.

WSGI Web Server Gateway Interface.

WSUI Web Service User Interface.

WTP Wireless Transaction Protocol.

WWW World Wide Web.

WYSIWYG What You See Is What You Get (interactive GUI editors).

X11 X Windows UI Protocol.

XHTML Xml version of HTML.

XML eXtensible Markup Language.

XP eXtreme Programming (a software development methodology).

XPATH XML path definition syntax.

XSL eXtensible Stylesheet Language.

XSLT eXtensible Stylesheet Language Transformations.

Appendix C

The Dynamic Markup Template language

The aim of the DMT is to be concise, expressive, and easy to learn and read. Using the DMT should allow developers and site designers to be more productive, create more reliable BUS structures, and increase the flexibility of designs.

C.1 Structure definition

The structural elements of the language are line elements and must begin a line, but may wrap across two or more lines. Values are strings and can omit quotes if consisting of a single word or number. These elements are listed in Figure C.1.

```
 1: $inputvars ARG1 ARG2 ARG3=DEFAULT [assign input values to identifiers]
 2: $rawfile FILENAME( ARG1 ARG2 ) [insert raw file contents in output]
 3: $include FILENAME( ARG1 ARG2 ) [process file contents]
 4: $template FILENAME( ARG1 ARG2 ) [process file with this output as body]
 5: $match [renders matched presentation and data objects]
 6: $loop [renders all child presentation for each data object]
 7: $find XPATH [sets the data pointer to item(s) found with xpath expr]
 8: $choose
 9:     $test EXPR
10: $if EXPR
11:     $then
12:     $elif EXPR
13:     $else
14: $reference EXPR [select data object within current data object tree]
15: $VAR = TEXT
16: $macro MACRONAME = TEXT $1 $2 TEXT
17: TAG ~STYLE PROP1=VALUE TEXT
18: # COMMENT
19: "TEXT"
```

Figure C.1: Structural elements of the Dynamic Markup Template language.

The $loop iterates through the current dataset, rendering contents. The $if, $then, $elif, and $else allow decisions on conditional display to be implemented using current data values. The $choose and its embedded $test operate like a `case` statement in many languages. A test will select a single presentation object to render and then exit the $choose element.

The language has been designed to make flexible presentation design easy, but there is no support (deliberately) for coding application logic with the template language.

C.2 Expression definition

Expressions may use a combination of variables, operators, constants, and functions (see Figure C.2). Variables are sources from the local variable scope, special system-supplied variables, values available from a template body (at a depth of two or more if required), and the attributes of the current data object.

```
 1: $VAR
 2: $_VAR [system variable. eg: templatebody, (index, maxindex,
 3:                  item, itemlist) in loops, now, filedate, filename]
 4: $.VAR [replaced with data attribute variable]
 5: $/VAR [replaced with template body variable]
 6: $//VAR [replaced with body of body variable, etc]
 7: $MACRONAME( ARG1 ARG2 )
 8: $(VAR default VALUE) [use variable, defaulting it to new value if N/A]
 9: $(VAR else VALUE) [use variable, or return value if N/A]
10: ${ EXPR }
```

Figure C.2: Variables and expressions of the Dynamic Markup Template language.

C.3 Discussion

Indentation defines scope and therefore the content (resolution) hierarchy. Local variable definitions are searched before variables from outer or global scopes.

Filenames do not require an explicit extension. The DMT will search for a filename with the expected extension on the path.

A backslash before a character voids its special meaning, and a backslash at the end of the line joins this line to the next line, deleting the leading space of that line.

Values only need quotes if string contains space. Single words, numbers, color values, and pathnames do not need quotes.

Keyword names are searched before variable names, so it is impossible to assign a value to `for` with `$for = 43`.

A special attribute ‘busname‘ can be used for recursive reference. Assigning a value to this attribute in a node allows children nodes to reference this presentation node with their place in the data tree, implementing display object recursion. As the data tree is of finite depth and cannot have recursive elements, the renderer is protected against infinite recursion.

Appendix D

BUS Active Expression Syntax

Content and attributes in the BUS can use "$ EXPR $" syntax to compute dynamic values during object rendering. The resolution of the expression uses the Python interpreter engine, and most Python language features are available to the BUS application developer (even when the application is built in another language).

Expressions can have the following format:

```
EXPR = TERM | EXPR OP EXPR
TERM = CONST | VAR | TERM "." METHOD TUPLE | TERM "." ATTRIBUTE
     | "(" EXPR ")" | FUNC TUPLE | TERM "[" INDEX "]"
     | LISTCOMP
TUPLE = "(" [EXPR] ("," EXPR)* ")"
INDEX = INTEXPR | [INTEXPR] ":" [INTEXPR] | KEYEXPR
```

There are three major namespaces that can be referenced:

var: Variables in scope from `<SetVar>` element.

Reference with `var.NAME`

data: Current data object from data resource hierarchy.

Child nodes in `data[INDEX]`,

Attributes in `data.NAME` or `data.get("NAME"[,default])`

self: This presentation element.

Element tag is in `self.otype`,

Child nodes in self.content, accessed with `self[INDEX]`,

Attributes in `self.NAME` or `self.get("NAME"[,default])`

Note: As presentation and data objects are built from XML, numbers will be in string format. The expression writer needs to use `int(value)` or similar before using the number in a numeric context.

Constants:

INT | FLOAT | "STRING" | 'STRING' | TUPLE | LIST | DICT

List Comprehensions:

LISTCOMP = "[" EXPR (for VARLIST in SEQ)+ (if EXPR)* "]"

Operators:

VALUE ARITH_OP VALUE
TEMPLATE % TUPLE : Replace %s with values from tuple
TEMPLATE % DICT : Replace %(KEY)s with values from dictionary
SEQ + SEQ
STRING + STRING

Boolean Operators:

A COMP_OP B
A and B
A or B
not A
VALUE [not] in SEQ
STRING [not] in STRING

String Test Methods:

find(SUB)
index(SUB)
count(SUB)
startswith(SUB)
endswith(SUB)
isdigit()
isalpha()

String Transform Methods:

strip()
replace(OLD,NEW)

```
title()
upper()
lower()
split(SEP [,MAX])
join(SEQ)
```

Regular Expression Functions:

```
All RE library functions start with "re."
search( PAT, STRING [,FLAGS] ) -> MATCHOBJ
split( PAT, STRING, [,MAX] ) -> SEQ
findall( PAT, STRING, [,FLAGS] ) -> SEQ of TUPLES
sub( PAT, REPLACER, STRING [,COUNT] ) -> STRING
escape( STRING ) -> SAFE_STRING
  FLAGS = re.I(ingore case) | re.M(mutiline) | re.S(dotall)
  MATCHOBJ has .group(GROUP), .groups(), .groupdict()
```

General Functions:

```
int(EXPR[,BASE])
float(EXPR)
str(EXPR)
len(SEQ|STRING)
max(SEQ)
min(SEQ)
sum(SEQ)
reversed(SEQ)
sorted(SEQ)
zip(SEQ (,SEQ)*)
range( MIN [,MAX] [,STEP] )
enumerate(SEQ)
lambda VARLIST ":" EXPR
reduce( FUNC, SEQ [,INIT] )
```

This language provides for most requirements for data transformation and decision branching in the presentation layer. However, for specialist uses, the language can be expanded by simply including a module containing the required functions in the BUS rendering environment.

Bibliography

[Abdelzaher & Bhatti 99] T. F. Abdelzaher and N. Bhatti. "Web server QoS management by adaptive content delivery". In *Intl. Workshop on Quality of Service*, pp 216–225, London. June 1999.

[Barbacci *et al* 95] Mario Barbacci, Mark H. Klein, Thomas A. Longstaff, and Charles B. Weinstock. "Quality Attributes". Technical Report CMU/SEI-95-TR-021, Software Engineering Institute, Carnegie Mellon University. 1995.

[Barta & Schranz 98] Robert A. Barta and Markus W. Schranz. "JESSICA: An Object-Oriented Hypermedia Publishing Processor". *Computer Networks*, Vol. 30, No. 1-7, pp 281–290. April 1998.

[Berglund 06] Anders Berglund. "Extensible Stylesheet Language (XSL) Version 1.1". Technical report, World Wide Web Consortium. Available from `http://www.w3.org/TR/2006/REC-xsl11-20061205/`. 2006.

[Berners-Lee 89] Tim Berners-Lee. "Information Management: A Proposal". Technical report, CERN. March 1989.

[Berners-Lee *et al* 94] T. Berners-Lee, R. Cailliau, A. Luotonen, H. F. Nielsen, and A. Secret. "The World-Wide Web". *Communications of the ACM*, Vol. 37, No. 8. 1994.

[Bianco *et al* 07] Phil Bianco, Rick Kotermanski, and Paulo Merson. "Evaluating a Service-Oriented Architecture". Technical Report CMU/SEI-2007-TR-015, Software Engineering Institute, Carnegie Mellon University. September 2007.

[Black 06] David Black. "Ruby for Rails: Ruby Techniques for Rails Developers". Manning Publications. May 2006.

[Booch 91] G. Booch. "Object Oriented Design with Applications". The Benjamin-Cummings Publishing Company, Redwood City, CA. 1991.

[Brown & Najork 96] Marc H. Brown and Marc A. Najork. "Distributed active objects". *Computer Networks*, Vol. 28, No. 7-11, pp 1037–1052. May 1996.

[Brown 02] David Bruce Brown. "A Views-Based Design Framework for Web Applications". M.Sc. thesis, University of Waterloo, Canada. 2002.

[Buschmann *et al* 96] F. Buschmann, R. Meunier, H. Rohnert, P.Sommerlad, and M. Stal. "Pattern-Oriented Software Architecture: A System of Patterns". John Wiley and Sons Ltd, Chichester, UK. 1996.

[Cassell 94] James Cassell. "The Total Cost of Client/Server: A Comprehensive Model". In *A Gartner Group Conference on the Future of Information Technology Industry.* November 1994.

[Ciancarini *et al* 98a] P. Ciancarini, A. Rizzi, and F Vitali. "An extensible rendering engine for XML and HTML". In *Proceedings of WWW7, Computer Networks and ISDN Systems*, volume 7, pp 225–238. 1998.

[Ciancarini *et al* 98b] Paolo Ciancarini, Robert Tolksdorf, Fabio Vitali, Davide Rossi, and Andreas Knoche. "coordinating multiagent applications on the WWW: A reference architecture". *IEEE Transactions on Software Engineering*, Vol. 24, No. 5, pp 362–375. May 1998.

[Clement *et al* 05] Luc Clement, Andrew Hately, Claus von Riegen, and Tony Rogers. "Universal Description, Discovery and Integration v3.0.2 (UDDI)". Technical report, Organization for the Advancement of Structured Information Standards. Available from `http://uddi.org/pubs/uddi_v3.htm`. 2005.

[Clements *et al* 02] Paul Clements, Rick Kazman, and Mark Klein. "Evaluating Software Architectures: Methods and Case Studies". Addison-Wesley. 2002.

[Coar & Robinson 99] K. Coar and D. Robinson. "The WWW Common Gateway Interface Version 1.1". Internet Draft, Internet Engineering Task Force. Work in progress. April 1999.

[Coda *et al* 98] F. Coda, C. Ghezzi, G. Vigna, and F. Garzotto. "Towards a Software Engineering Approach to Web Site Development". In *Proceedings of the* 9^{th} *International Workshop on Software Specification and Design*, pp 8–17, Ise-Shima, Japan. April 1998. IEEE Press.

[Cowan *et al* 06] John Cowan, C. M. Sperberg-McQueen, Francois Yergeau, Eve Maler, Tim Bray, and Jean Paoli. "Extensible Markup Language (XML) 1.1 (Second Edition)". Technical report, World Wide Web Consortium. Available from `http://www.w3.org/TR/xml11`. 2006.

[Cox & Novobilski 86] Brad J. Cox and Andrew J. Novobilski. "Object Oriented Programming: An Evolutionary Approach". Addison-Wesley. 1986.

[Darcy & Kemerer 05] D.P. Darcy and C.F. Kemerer. "OO metrics in practice". *Software*, Vol. 22, No. 6, pp 17–19. Nov.-Dec 2005.

[DM 02] DM Solutions Group. "ROSA Java Applet". August 2002. Available from `http://www.maptools.org/rosa/`.

[ECM97] ECMA. "Standard ECMA-262 ECMAScript: A general purpose, cross-platform programming language". June 1997.

[Engels *et al* 07] Holger Engels, Christian Kochs, and Stephan Schuster. "wingS White Paper". Technical report, Wings Framework Development Team. Available from `http://wingsframework.org/doc/whitepaper/pdf/whitepaper.pdf`. 2007.

[Fayad & Schmidt 97] Mohamed Fayad and Douglas C. Schmidt. "Object-oriented application frameworks". *Commun. ACM*, Vol. 40, No. 10, pp 32–38. October 1997.

[Fernandez *et al* 00] Mary F. Fernandez, Daniela Florescu, Alon Y. Levy, and Dan Suciu. "declarative specification of web sites with strudel". *VLDB Journal*, Vol. 9, No. 1, pp 38–55. 2000.

[Ferraiolo *et al* 03] Jon Ferraiolo, Fujisawa Jun, and Dean Jackson. "Scalable Vector Graphics (SVG) 1.1 Specification". Technical report, World Wide Web Consortium. Available from `http://www.w3.org/TR/SVG/`. 2003.

[Fielding & Taylor 00] Roy T. Fielding and Richard N. Taylor. "Principled Design of the Modern Web Architecture". In *ICSE2000: Procedings of the International Conference on Software Engineering*. 2000.

[Fielding 00] Roy Thomas Fielding. "Architectural Styles and the Design of Network-based Software Architectures". PhD thesis, University of California, Irvine. 2000.

[Fielding *et al* 98] R. Fielding, J. Gettys, J. Mogul, H. Nielsen, L. Masinter, P. Leach, and T. Berners-Lee. "Hypertext Transfer Protocol – HTTP/1.1". Internet Draft, Internet Engineering Task Force. Work in progress. November 1998.

[Fraternali 98] Piero Fraternali. "Web development tools: a survey". *Computer Networks*, Vol. 30, No. 1-7, pp 631–633. April 1998.

[Gam] "Gamelan Java Software Development Web Site". `http://www.gamelan.com/`.

[Gebhardt & Henderson 99] J. C. Gebhardt and L. Henderson. "WebCGM: Industrial-strength vector graphics for the Web". Technical report, CGM Open Consortium, Inc. Available from `http://www.cgmopen.org/webcgmintro/paper.htm`. January 1999.

[Gellersen & Gaedke 99] Hans-Werner Gellersen and Martin Gaedke. "object-oriented web application development". *IEEE Internet Computing*, Vol. 3, No. 1, pp 60–68. 1999.

[Gellerson *et al* 97] Hans-Werner Gellerson, Robert Wicke, and Martin Gaedke. "WebComposition: An Object-Oriented Support System for the Web Engineering Lifecycle". In *Proceedings of the 6th International WWW Conference*. 1997.

[Goo06] Google. "How To Use the Google Template System". February 2006. Available from `http://google-ctemplate.googlecode.com/svn/trunk/doc/howto.html`.

[Goo08] Google, Inc. "Google Maps". 2008. `http://code.google.com/apis/maps/`.

[Goodman 98] D. Goodman. "Dynamic HTML: The Definitive Reference". O'Reilly and Associates Inc., Sebastopol, CA. 1998.

[Gosling & McGilton 96] J. Gosling and H. McGilton. "The Java Language Environment: A White Paper". Technical report, Sun Microsystems. Available from `http://java.sun.com/docs/white/langenv/`. 1996.

[Gregorio 06] Joe Gregorio. "Why so many Python web frameworks?". September 2006. Available at `http://bitworking.org/news/Why_so_many_Python_web_frameworks`.

[Hester *et al* 98] A. Hester, R. Borges, and R. Ierusalimschy. "Building Flexible and Extensible Web Applications with Lua". *Journal of Universal Computer Science*, Vol. 4, No. 9. 1998.

[Houben *et al* 05] Geert-Jan Houben, Peter Barna, and Flavius Frasincar. "Hera Presentation Generator". In *Proceedings of WWW conference: 2005*. 2005.

[Ingham *et al* 98] D. B. Ingham, S. J. Caughey, and M. C. Little. "Supporting Highly Manageable Web Services". In *Proceedings of the 7th International WWW Conference*. 1998.

[Ishikawa *et al* 07] Masayasu Ishikawa, Peter Stark, Mark Baker, Shin'ichi Matsui, Toshihiko Yamakami, and Ted Wugofski. "XHTML Basic 1.1 Specification". Technical report, World Wide Web Consortium. Available from `http://www.w3.org/TR/xhtml-basic/`. 2007.

[Jav] "Java Boutique Java Developer Web Site". `http://javaboutique.internet.com/`.

[Johnson & Foote 88] Ralph Johnson and Brian Foote. "Designing Reusable Classes". *Journal of Object-Oriented Programming*. June/July 1988.

[jsD] "Dojo Javascript Library". `http://dojotoolkit.org/`.

[jsMochi 06] Mochi Media Inc. "Mochikit: A lightweight Javascript library". 2006. `http://www.mochikit.com`.

[JSONspec 06] JSON.org. "Introducing JSON". 2006. `http://www.json.org/`.

[jsP] "The Prototype Javascript Library". `http://www.prototypejs.org/`.

[kar08] "Karrigell Website". `http://karrigell.sourceforge.net/en/front.htm`. 2008.

[Karmarkar *et al* 07] Anish Karmarkar, Yves Lafon, Noah Mendelsohn, Martin Gudgin, Jean-Jacques Moreau, Henrik Frystyk Nielsen, and Marc Hadley. "SOAP Version 1.2 part 1: messaging framework (second edition)". Technical report, World Wide Web Consortium. Available from `http://www.w3.org/TR/2007/REC-soap12-part1-20070427/`. 2007.

[Kay 07] Michael Kay. "XSL Transformations (XSLT) Version 2.0". Technical report, World Wide Web Consortium. Available from `http://www.w3.org/TR/2007/REC-xslt20-20070123/`. 2007.

[Krasner & Pope 88] Glenn E. Krasner and Stephen T. Pope. "A Cookbook for Using the Model View Controller User Interface Paradigm in Smalltalk 80". *Journal of Object Oriented Programming*, Vol. 1, No. 3. August 1988.

[Kristensen 98] Anders Kristensen. "Template resolution in XML/HTML". *Computer Networks*, Vol. 30, No. 1-7, pp 239–249. April 1998.

[Layman *et al* 06] Andrew Layman, Richard Tobin, Dave Hollander, and Tim Bray. "Namespaces in XML 1.1 (Second Edition)". Technical report, World Wide Web Consortium. Available from `http://www.w3.org/TR/xml-names11`. 2006.

[Lee & Shirani 04] Seung C. Lee and Ashraf I. Shirani. "a component based methodology for web application development". *J. Syst. Softw.*, Vol. 71, No. 1-2, pp 177–187. 2004.

[Leff & Rayfield 07] Avraham Leff and James T. Rayfield. "WebRB: A Language and Runtime for Multi–page Relational Web Applications". In *PLDI 2007*. 2007.

[Lutz 96] M. Lutz. "Programming Python". O'Reilly and Associates Inc. 1996.

[Maia 02] Joao Prado Maia. "Introducing Smarty: A PHP Template Engine". May 2002. Available from `http://www.onlamp.com/pub/a/php/2002/09/05/smarty.html`.

[Min08] Miniwatts Marketing Group. "Internet World Stats - December 2007". 2008. `http://www.internetworldstats.com/stats6.htm`.

[Musciano & Kennedy 02] Chuck Musciano and Bill Kennedy. "HTML & XHTML: The Definitive Guide". O'Reilly Media, Inc., 5th edition. August 2002.

[Net07] Netcraft Ltd. "December 2007 Web Server Survey". December 2007. `http://news.netcraft.com/archives/2007/12/29/december_2007_web_server_s%urvey.html`.

[Niederst 99] J. Niederst. "Web Design in a Nutshell". O'Reilly and Associates Inc., Sebastopol, CA. 1999.

[Obj95] Object Management Group Inc. "The Common Object Request Broker: Architecture and Specification", 2.0 edition. July 1995.

[O'Reilly 05] Tim O'Reilly. "What is Web 2.0? Design Patterns and Models for the Next Generation of Software". *O'Reilly Network*. September 2005. Available from `http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web%-20.html`.

[Parr 04] Terence John Parr. "enforcing strict model-view separation in template engines". In *Proceedings of the Thirteenth International World Wide Web Conference*, pg 224. ACM Press, New York, NY. May 2004.

[Plo06] "PlotKit - Javascript Chart Plotting". 2006. `http://www.liquidx.net/plotkit/`.

[Pyt06] Python Software Foundation. "Python Web Server Gateway Interface v1.0". 2006. Available from `http://www.python.org/dev/peps/pep-0333/`.

[Quarto-vonTivadar *et al* 05] John Quarto-vonTivadar, Brian Kotek, Brian LeRoux, Sandy Clark, and Perry Woodin. "Discovering Fusebox 4". Techspedition, 2nd edition. 2005.

[Ramm *et al* 06] Mark Ramm, Kevin Dangoor, and Gigi Sayfan. "Rapid Web Applications with TurboGears". Prentice Hall. 2006.

[Ramu & Gemuend 00] Chenna Ramu and Christina Gemuend. "cgimodel: CGI programming made easy with Python". *The Linux Journal*, Vol. 75. July 2000.

[Rees 97] M.J. Rees. "Exploiting the Full Web User Interface Spectrum". In *Proceedings of AusWeb97*. 1997.

[RMI97] "Remote Method Invocation Specification". 1997. Available from `http://www.javasoft.com/products/JDK/1.1/docs/guide/rmi/spec/rmiTOC.doc%.html`.

[Robinson & Jackson 99] B. Robinson and D. Jackson. "SVG Viewer". `http://sis.cmis.csiro.au/svg/`. 1999.

[Rub07] "Ruby on Rails". `http://api.rubyonrails.org/`. 2007.

[Rustad 05] Aaron Rustad. "Ruby on Rails and J2EE: Is there room for both?". July 2005. Avalaible from `http://www.ibm.com/developerworks/web/library/wa-rubyonrails/`.

[Ryman *et al* 07] Arthur Ryman, Roberto Chinnici, Sanjiva Weerawarana, and Jean-Jacques Moreau. "Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language". Technical report, World Wide Web Consortium. Available from `http://www.w3.org/TR/wsdl20/`. 2007.

[Schranz 98] Markus Schranz. "Engineering Flexible World Wide Web Services". In *Symposium on Applied Computing (SAC)*. February 1998.

[Schwabe *et al* 01] Daniel Schwabe, Luiselena Esmeraldo, Gustavo Rossi, and Fernando Lyardet. "Engineering Web Applications for Reuse". *IEEE Multimedia*, Vol. 8, No. 1. jan-mar 2001.

[Sec02] Security Space. "Apache Module Report". June 2002. `http://www.securityspace.com/s_survey/data/man.200205/apachemods.html`.

[Shneiderman 97] B. Shneiderman. "Direct Manipulation for Comprehensible, Predictable and Controllable User Interfaces". Technical report, Human Computer Interaction Lab, Dept. of Computer Science, University of Maryland. 1997.

[Sim] "SimpleJSON". `http://www.python.org/pypi/simplejson/1.3`.

[Ste05] "WSGI Server Utils". 2005. Available from `http://www.owlfish.com/software/wsgiutils/documentation/`.

[Sultana 08] Deanie Sultana. "Aussie Internet usage overtakes TV viewing for the first time". News release, The Nielsen Company. March 2008.

[Sun02] Sun Microsystems, Inc. "Designing Enterprise Applications with the J2EETM Platform", second edition. 2002. Available from `http://java.sun.com/blueprints/guidelines/designing_enterprise_applicat%ions_2e/DEA2eTOC.html`.

[Sweeney 00a] M. Sweeney. "BUS: a Browser Based User Interface Service for Web Based Applications". In *Australian Computer Science Communications: Proceedings of the First Australasian User Interface Conference*, volume 22, pp 103–109. IEEE Computer Society, Los Alamitos, California, USA. January 2000.

[Sweeney 00b] M. Sweeney. "Interactive Graphics for Web Based Applications". In *Proceedings of the 1st International Conference on Web Information System Engineering*, volume 1, pp 395–399. IEEE Computer Society, Los Alamitos, California, USA. June 2000.

[Taivalsaari 96] A. Taivalsaari. "On the Notion of Inheritance". *ACM Computing Surveys*, Vol. 28, No. 3. September 1996.

[Theng & Thimbleby 98] Yin Leng Theng and Harold Thimbleby. "Addressing Design and Usability Issues in Hypertext and on the World Wide Web by Re-Examining the "Lost in Hyperspace" Problem". *Journal of Universal Computer Science*, Vol. 4, No. 11. 1998.

[Tittel *et al* 95] E. Tittel, M. Gaither, S. Hassinger, and M. Erwin. "Foundations of World Wide Web Programming with HTML and CGI". IDG Books Wordwide, Foster City, CA. 1995.

[Ungar & Smith 87] D. Ungar and R. B. Smith. "Self: The Power of Simplicity". In *OOPSLA '87 Proceedings*. 1987.

[van derVlist 02] Eric van der Vlist. "XML Schema". O'Reilly and Associates, Inc. 2002.

[Wall & Schwartz 92] Larry Wall and Randal L. Schwartz. "Programming Perl". O'Reilly and Associates, Inc. 1992.

[Walton & Hibbs 06] Bill Walton and Curt Hibbs. "Rolling with Ruby on Rails Revisited". December 2006. Available from `http://www.onlamp.com/pub/a/onlamp/2005/01/20/rails.html`.

[Wampler 01] Dean Wampler. "Cat Fight in a Pet Store: J2EE vs .NET". *OnJava.com*. 2001. Available from `http://www.onjava.com/pub/a/onjava/2001/11/28/catfight.html`.

[Web05] "Introduction (version 1.39)". 2005. Avalable from `http://www.masonhq.com/docs/manual/Mason.html`.

[Zhao *et al* 02] Weiquan Zhao, David Kearney, and Gianpaolo Gioiosa. "architectures for web based applications". In *The Fourth Australian Workshop on Software and System Architectures*. 2002.

[zor06] "High Performance JavaScript Vector Graphics Library". 2006. `http://www.walterzorn.com/jsgraphics/jsgraphics_e.htm`.