Software Engineering for Modern Web Applications:
Methodologies and Technologies

Daniel M. Brandon
Christian Brothers University, USA

Information Science REFERENCE

INFORMATION SCIENCE REFERENCE

Hershey · New York

Acquisitions Editor: Kristin Klinger
Development Editor: Kristin Roth
Senior Managing Editor: Jennifer Neidig
Managing Editor: Jamie Snavely
Assistant Managing Editor: Carole Coulson
Copy Editor: Brenda Leach
Typesetter: Carole Coulson
Cover Design: Lisa Tosheff
Printed at: Yurchak Printing Inc.

Published in the United States of America by
Information Science Reference (an imprint of IGI Global)
701 E. Chocolate Avenue, Suite 200
Hershey PA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@igi-global.com
Web site: http://www.igi-global.com

and in the United Kingdom by
Information Science Reference (an imprint of IGI Global)
3 Henrietta Street
Covent Garden
London WC2E 8LU
Tel: 44 20 7240 0856
Fax: 44 20 7379 0609
Web site: http://www.eurospanbookstore.com

Library of Congress Cataloging-in-Publication Data

Software engineering for modern Web applications : methodologies and technologies / Daniel M. Brandon, editor.

p. cm.

Summary: "This book presents current, effective software engineering methods for the design and development of modern Web-based applications"--Provided by publisher.

Includes bibliographical references and index.

ISBN 978-1-59904-492-7 (hardcover) -- ISBN 978-1-59904-494-1 (ebook)

1. Application software--Development. 2. Internet programming. 3. Web site development. 4. Software engineering. I. Brandon, Dan, 1946-

QA76.76.A65.S6588 2008

005.1--dc22

2008008470

British Cataloguing in Publication Data
A Cataloguing in Publication record for this book is available from the British Library.

Chapter VIII
Evolving Web Application Architectures:
From Model 2 to Web 2

David Parsons
Massey University, New Zealand

ABSTRACT

This chapter explores how Web application software architecture has evolved from the simple beginnings of static content, through dynamic content, to adaptive content and the integrated client-server technologies of the Web 2.0. It reviews how various technologies and standards have developed in a repeating cycle of innovation, which tends to fragment the Web environment, followed by standardisation, which enables the wider reach of new technologies. It examines the impact of the Web 2.0, XML, Ajax and mobile Web clients on Web application architectures, and how server side processes can support increasingly rich, diverse and interactive clients. It provides an overview of a server-side Java-based architecture for contemporary Web applications that demonstrates some of the key concepts under discussion. By outlining the various forces that influence architectural decisions, this chapter should help developers to take advantage of the potential of innovative technologies without sacrificing the broad reach of standards based development.

INTRODUCTION

Web applications grew out of the World Wide Web in the 1990s, driven by the need, particularly within e-commerce applications, for dynamic content. Early Web content, limited to static hypertext markup language (HTML) pages, could not support on-line sales, transactions, personalisation, or any of the other features of the Web that we now take for granted. Subsequently, technologies that could build Web pages on the fly, such as common gateway interface (CGI) scripts, active server pages (ASPs) and Java Enterprise Edition components like servlets and JavaServer Pages (JSPs), transformed the landscape of the Web by introducing the Web *application*, rather than just the Web *site*.

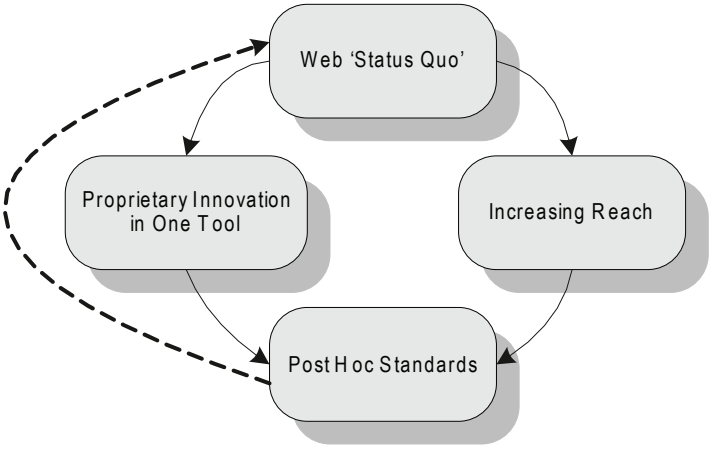
The cycle of innovation that saw the change from static to dynamic content in the mid 1990s has continued unabated. The post 'dot com' era has seen Web application design move into a number of new areas. On the one hand we see a rapid evolution of the Web client into the mobile, cross-platform space, with mobile browser-hosted technologies such as XHTML-MP (extensible hypertext markup language—mobile profile, the standard mark-up promoted by the Open Mobile Alliance, a consortium of commercial interests in the mobile communications industry), and micro browsers like Opera Mobile that run on top of a Java Micro Edition environment. On the other hand we see a change on the server side from the 'walled garden' Web application, providing only content from a single source, to open Service-Oriented Architectures that enable disparate services from many sources to be integrated by exchanging data using the extensible markup language (XML). On both client and server we see service driven 'mashups' (re-used, intermingled services) and the programmable Web, aspects sometimes associated with the umbrella term *the Web 2.0*. These changes raise questions about the future of the Web application, for example, can we realistically consider both a rich client and a mobile application that will work across many different types of device? How do extensible markup language (XML) based technologies like Web services and XSLT (extensible stylesheet language transformations) fit in to Web application architecture? How will the Web client evolve with the increasing use of browser hosted applications and Ajax (Asynchronous JavaScript and XML)? Much of the discussion around the Web 2.0 focuses on the social networking aspects of Web based applications, but there is an equally important discussion to be had about the underlying architectures and technologies of applications in the Web 2.0 era. In this chapter we try to address some questions about how Web application architecture continues to evolve, what forces come into play, and how apparently conflicting paths of development may converge into new approaches. Illustrative examples are provided using the Java Enterprise Edition and supporting open source tools.

THE WEB FRAGMENTATION CYCLE

One of the characteristics of the evolution of the Web is what we might term the *Web fragmentation cycle*, which is the effect of technology driven change in two conflicting directions. On the one hand, we have seen that a particular Web technology, for example a particular type of browser or server side application, can drive Web evolution using features specific to that technology. An early example of this was the introduction into Netscape Navigator of HTML tags for presentation (Lie & Bos, 1999). Netscape's development of LiveScript (later JavaScript), and Microsoft's introduction of the XMLHttpRequest into Outlook Web Access (Van Eaton, 2005) can also be seen in this light. On the other hand we see proliferation of Web access tools that increase the reach of the Web across many different types of client. One example of this was the consortium of mobile phone network operators that joined in the WAP (wireless access protocol) Forum in the late 1990s to enable Web access via a wide range of mobile devices by introducing the WAP browser and the wireless markup language (WML).

The effect of technology specific innovation is to narrow the accessibility of Web content to those who have the appropriate technology. While these innovations may increase qualities such as usability and functionality for some, they will exclude those who do not have the right technology. In contrast, the effect of wider reach is to encourage generic technologies for Web access that enable more types of client to access Web-based content, while potentially decreasing usability and functionality due to the need to run on a lowest common denominator platform. The resolution to this dichotomy has traditionally been the introduction of standards on a post hoc

Figure 1. The Web fragmentation cycle



basis, which set an industry-wide benchmark that most products will ultimately comply with. These standards enable the fragmentation caused by innovation to be (at least partially and temporarily) resolved by integrating propriety innovation into a common Web technology platform (Figure 1). Once proprietary innovations have been integrated into a common standard, those technologies enjoy broader implementation and wider reach, and the Web 'status quo' moves forward.

Identifying the Web Fragmentation Cycle in Practice

Of course the fragmentation cycle shown in Figure 1 is an over-simplification, since there are many overlapping cycles related to different Web application technologies. Therefore the concept of the Web status quo is somewhat misleading, since there is never a stable point from which we embark on a new cycle. However it is possible to identify specific instances of this cycle at various points in time and identify the two aspects of innovation and reach. One such example was the level 1 document object model (DOM), which was designed to introduce a standard, cross platform, language independent view of how a document should be accessed and manipulated, but was partly based on the concept of the 'level 0' DOM, which was the de facto standard already implemented in the browsers of the time. Similarly, at around the same time, the HTML specification, including its many contradictions and lack of well-formedness, was engineered from innovations already established by browsers in common use. While the fragmentary, proprietary views of the DOM and HTML were emerging, others were attempting to maintain the original concepts of separation of structure, content and presentation that were originally intended for Web-based documents, for example, by promoting the standardisation and use of style sheets (Lie & Bos, 1999). Over a series of standardisation cycles, that concept has gradually been reasserted, with the development of the extensible hypertext markup language (XHTML) and cascading style sheets (CSS). Unlike HTML, XHTML does not allow the arbitrary mixing of content, structure and presentation, delegating presentation and, to some extent, structural management of Web page content (written in XHTML) to external style sheets (written in CSS).

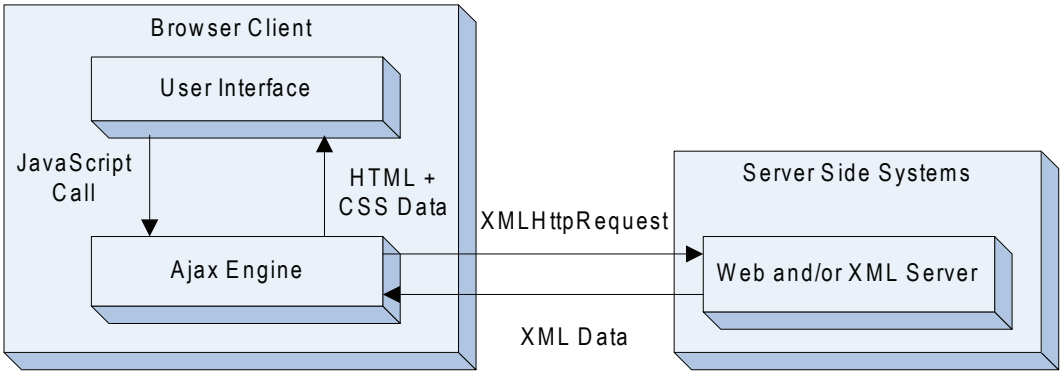
The Emergence of Ajax

A recent example of the Web fragmentation cycle has been the emphasis in browser hosted tools for providing a richer client experience, including Web service clients for mashup services and, in particular, Ajax (asynchronous JavaScript and XML). Ajax is not a particularly new concept, following on as it does from a longer tradition of client side processing. This began in the mid 1990s with the introduction of JavaScript into Netscape Navigator and Java applets into the HotJava browser, and was followed by dynamic HTML (DHTML) which combined HTML, style sheets and scripts in a way that enabled documents to be 'animated' (dynamically manipulated) in the browser, with a view of the document that has since been formalised by the W3Cs document object model (Le Hégaret, Whitmer, & Wood, 2006). However the significant difference between Ajax and previous approaches is the concept of the "one page Web application," whereby page content is updated asynchronously from the server without the whole page being rebuilt. The two main advantages of this approach are that it enables a more interactive user experience, and that it can reduce the amount of Web traffic required to update a page, though both of these are dependent on careful design. An early example of this approach was Google Suggest, which was able to dynamically populate a search text box with suggestions for search terms as characters were typed into it, providing, of course, that the browser was able to support it. Ajax itself is not a technology but a label, applied by Garrett (2005), to a way of building Web applications that uses the XMLHttpRequest object within client side scripts to seamlessly update Web pages. Garret summarised Ajax as a combination of:

- Standards based presentation using XHTML and CSS
- Dynamic display and interaction using the document object model
- Data interchange and manipulation using XML and extensible stylesheet language transformations (XSLT)
- Asynchronous data retrieval using the XMLHttpRequest
- JavaScript binding everything together

Figure 2 shows the general architecture of Ajax based systems. The key to this architecture is that the Ajax engine mediates between the user interface and the server, processing on the client where possible (using DHTML) and, where necessary, sending asynchronous HTTP requests and receiving XML data (or indeed data in any other suitable format) that it renders in the browser via the DOM. Of course the technologies listed

Figure 2. Ajax architecture



by Garrett are not the only way to provide one-page applications on the Web, since alternative technologies like Flash can be used to similar effect.

As a simple example of how Ajax might be applied, we might consider how it could be used to manage the submission of login data from a Web page. In the usual client server interaction, an HTML form will submit data to the server using a standard action that posts the data to a URI. JavaScript may perform some surface validation on the client (e.g., checking for empty fields or password length), but true validation (checking the user's ID and password against the security domain) is done after the page is submitted to the server. Here for example an HTML 'form' tag submits to a server side URI after invoking a local validation function

```
<form action="/processlogin" method="post"
onsubmit="return validateLoginData(this);">
```

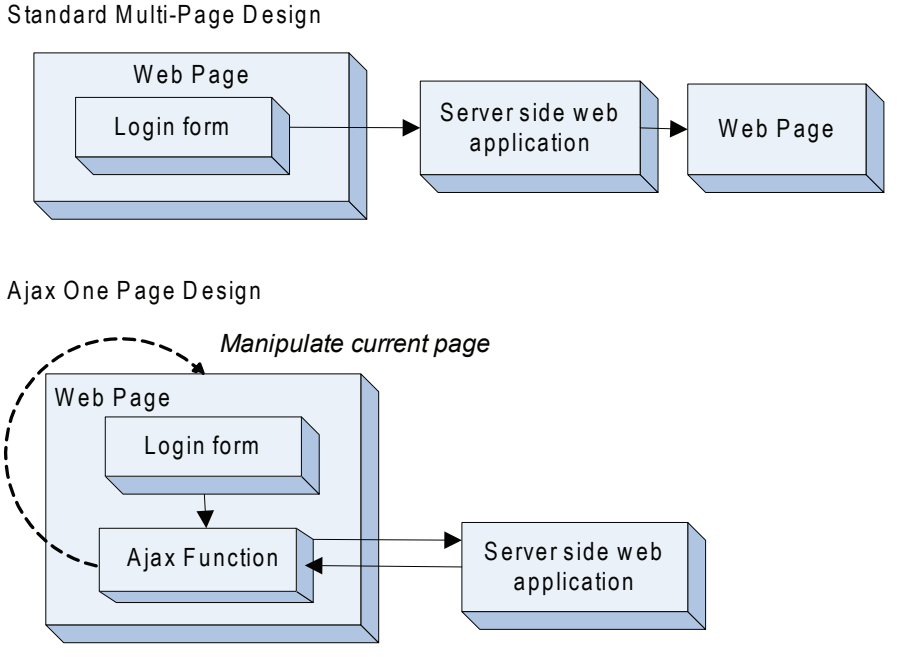
The server response will then be to generate a different client page, either a regenerated login page (if there are errors) or the next logical page in the business process. In contrast an Ajax approach might be to trigger a local Ajax function by an event such as a button being pressed (or tabbing fields, or keys being pressed) rather than submitting a form to a server side URI.

```
<input name="submit» type="button"
onclick="ajaxLogin();" value=»Submit" />
```

The local Ajax function will use an XMLHttpRequest to communicate asynchronously with the server, and trigger a function that processes the return value and interacts with the current page to update it appropriately.

```
...
xhrequest = getXMLHttpRequest();
…
xhrequest.onreadystatechange = processLogin;
xhrequest.open("post", url, true);
xhrequest.send(null);
...
```

Figure 3. Standard multi-page processing compared with the Ajax "one page" approach



```
function processLogin()
…
// process the server response data and manipulate the current page
```

Figure 3 shows how the two approaches differ in terms of their page flow. The Ajax version is the "one page Web."

Although they can bring significant benefits in the contexts in which they work, the aspect of fragmentation that Ajax and similar technologies bring is that they rely on the programming platform within the Web browser, excluding those Web clients that do not support these technologies. At the same time there has been a separate move towards broader Web access on mobile devices, which is in conflict with the reliance on client side JavaScript or similar tools that are needed to support Ajax. Indeed there is a potential conflict between two of the Web 2.0 patterns, as described by O'Reilly (2005), namely "software above the level of a single device" and "rich user experience." Of course there is no reason why these two patterns should be mutually exclusive, but the Ajax approach to a rich user experience, with the current level of mobile browser technology, is potentially fragmentary. Whilst some rudimentary Ajax implementations may be possible with JavaScript enabled mobile browsers like Opera Mobile, most of the highly interactive, desktop style Ajax application cannot currently be accessed from mobile devices.

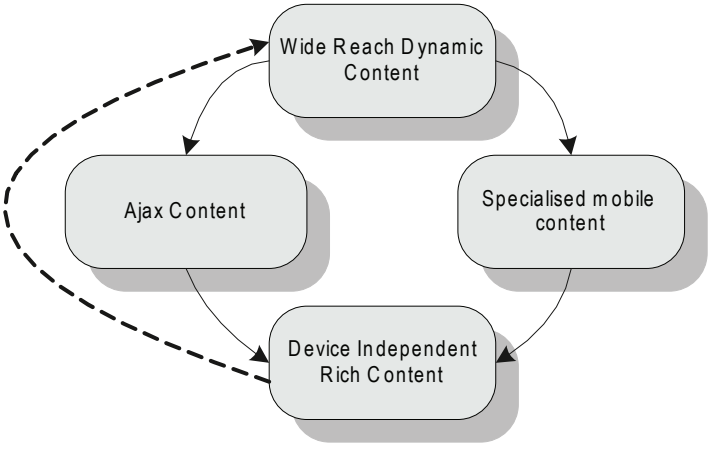
Mobile Standardisation: The .mobi Domain

At the same time as the desktop browser was being increasingly leveraged to develop a rich, interactive user experience based on asynchronous server requests, there was a move by the mobile phone network operators and handset manufacturers in the Open Mobile Alliance to increase the reach of the Web by introducing the .mobi top level domain in 2006. The intention of this new domain was to provide a standard way for users to access the mobile Internet by guaranteeing that a site with a .mobi extension was designed for mobile device access. The primary mechanism for this was to use XHTML-MP (extensible hypertext mark-up language—mobile profile), the successor to the WAP Forum's wireless mark-up language (WML) as the standard page mark-up for .mobi domains (the WAP Forum was the precursor to the Open Mobile Alliance). This approach, however, has come in for some severe criticism. Tim Berners Lee, the 'father' of the World Wide Web, commented that:

This domain will have a drastically detrimental effect on the Web. By partitioning the HTTP information space into parts designed for access from mobile access and parts designed (presumably) not for such access, an essential property of the Web is destroyed (Berners Lee, 2004).

However, over time we can expect that this separation between mobile and non-mobile Web sites will fade away as first technology, and then standards, will minimise the differences between desktop and mobile content delivery. Further development of mobile browsers will begin to provide Ajax or alternative rich client functionality on more types of mobile device. We can also expect the technologies that underlie the .mobi domain, such as XHTML-MP, to evolve over time and support such an increase in functionality. After a period dominated by fragmented tools and techniques, we can expect a common set of standards to emerge that will give consistency across the Web, regardless of browser, rendering specialised mobile content unnecessary. Figure 4 shows how Ajax and specialised mobile content can be seen to fit into the Web fragmentation cycle, and that the two trends should resolve into device independent rich content.

Figure 4. The fragmentation between specialised mobile content and Ajax



Listing 1. The innerHTML property and DOM methods compared

```
<?xml version="1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
  "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
 <head>
  <title>innerHTML and the DOM</title>
 </head>
 <body>
  <div id="section1"></div>
  <div id="section2"></div>
  <script type="text/javascript">

 // add an element containing text using innerHTML
   document.getElementById("section1").innerHTML =
    "<p id='para1'>Section 1 uses innerHTML</p>";

  // add an element containing text using the DOM
  var para2 = document.createElement("p");
  para2.setAttribute("id","para2");
  para2.appendChild(document.createTextNode
   ("Section 2 uses the DOM"));
  document.getElementById("section2").appendChild(para2);

  </script>
 </body>
</html>
```

Innovations That Do Not Become Standards

Although we have so far presented the fragmentation cycle as a repeating pattern of innovation and reach being resolved by standardisation, this is also an over-simplification in the sense that not all innovations are neatly resolved by the development of agreed standards. One example where this process does not appear to be satisfactorily resolved is the common use and implementation of the innerHTML property within many browser DOMs, despite the slim chance of it becoming a standard part of the DOM specification published by the World Wide Web Consortium (W3C). Use of this property is driven largely by its simplicity, since it makes it easy for Web developers to insert dynamic content into a text node of a Web based document. This makes it very useful for developing Ajax applications. Objections to the use of this property are primarily that it can be easily misused, since content, structure and presentation aspects of the document can become muddled together with client side scripts when content is manipulated in this way. Unfortunately the alternative approach, to use the standard APIs of the DOM, is often considerably more complex, compounded by variations in the way that the DOM APIs are actually implemented in different browsers. The simple example in Listing 1 shows an XHTML page that includes a script (written in JavaScript) that adds an element containing text using the innerHTML property, and adds a similar element using the methods of the DOM. Even for such a simple operation, the DOM version is much more complex. Not only is the code itself longer, but the performance of the innerHTML approach is much faster in most situations (Koch, 2006)

Given the conflicting viewpoints in this debate, it is difficult see to how the fragmentation can be easily resolved. Therefore we should acknowledge again that the simplistic view of the fragmentation cycle that sees a clean merging of innovations and standards will not always apply.

WEB 1.0 ARCHITECTURE

So far in this chapter we have focused largely on the evolving client side components of Web application architecture. However it is on the server that the major Web application processes actually take place. Therefore, we also need to consider how Web application architectures have evolved in terms of server side components and interactions. Before exploring how current innovations and standards may affect the future direction of Web application design, we will first review some of the main architectural aspects of established practice. In the spirit of the level 0 DOM, we might perhaps categorise pre-Web 2.0 architectures as "the Web 1.0." In this type of architecture, there are some standard patterns that are commonly used and integrated into popular Java Web application frameworks such as Struts and JavaServer Faces. These patterns include the Model 2 architecture (Seshadri, 1999), which is loosely based on the Model View Controller pattern (Buschmann et al., 1996) and the Template View pattern (Fowler, 2003).

The Server Page Template Model

Established Web application technologies that support dynamic content, like JavaServer Pages (JSPs) and Microsoft's Active Server Pages, use a relatively simple model of a layered Web based architecture, whereby server side application execution can be integrated into presentational mark-up. Dynamic content is based on the transformation of database content into Web based pages, and form based input into database updates. Architectural patterns for this type of Web application are based on simplifications of the Model View Controller pattern, like the JSP

Listing 2. An example of the Template View pattern using a JavaServer Page and the JSTL

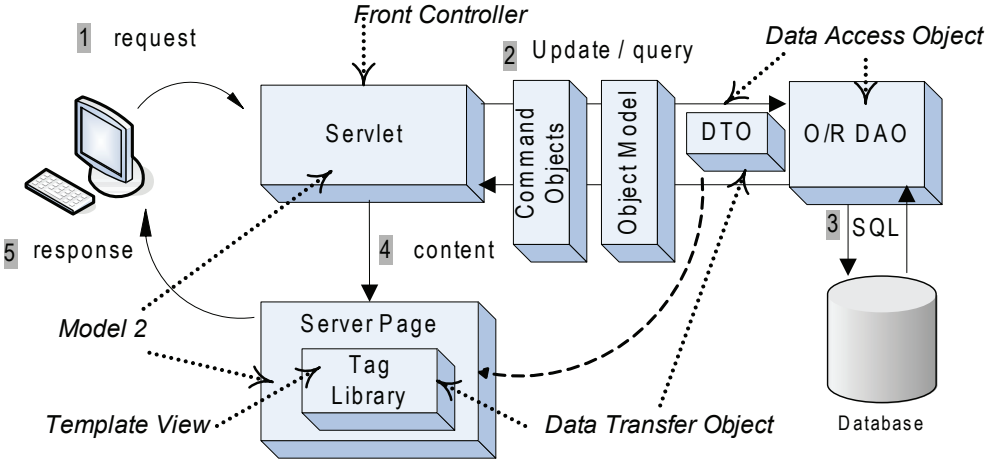
```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
 xmlns:c="http://java.sun.com/jsp/jstl/core"
 doctype-public="-//W3C//DTD XHTML 1.1//EN"
 doctype-system=" http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd "
 version="2.1">

 <jsp:useBean id="choice" class="com.webhomecover.beans.QuoteChoice"
scope="session" />
 <jsp:useBean id="contents" class="com.webhomecover.beans.ContentsDetails"
scope="session" />
 <jsp:useBean id="buildings" class="com.webhomecover.beans.BuildingsDetails"
scope="session" />

 <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
  <head>
   <link href="webhomecover.css" rel="stylesheet"
    type="text/css" />
   <title>WebHomeCover Insurance Quote</title>
  </head>
  <body>
   <h1>Here is your insurance quote from WebHomeCover</h1>
   <c:if test="${choice.buildings}">
    <h2>Your Buildings Quote:</h2>
    <jsp:getProperty name="buildings"
     property="formattedInsuranceQuote"/>
   </c:if>
   <c:if test="${choice.contents}">
    <h2>Your Contents Quote:</h2>
    <jsp:getProperty name="contents"
     property="formattedInsuranceQuote"/>
   </c:if>
   <form action="welcome.jsp" method="get">
    <input type="submit" value=
    "Thanks for the quote, now take me back to the home page" />
   </form>
  </body>
 </html>
</jsp:root>
```

Model 2 architecture (Seshadri, 1999). Here, a server side component at the controller layer, the *Front Controller* (Fowler, 2003), is responsible for interpreting client requests, delegating to other server side components in the model layer to provide the necessary data objects, and ensuring that the thread of control is ultimately forwarded to a component suitable for rendering the client response. This component is usually a server page, which is able to embed dynamic content into presentational markup using special tags. Fowler (2003) characterises this as the *Template View* approach to page generation, where the server page provides the presentation template and dynamic content is inserted into it at predefined locations, preferably using XHTML compliant tags. The code example in Listing 2 shows a JSP that uses the template view approach. It is a document in XML format that contains a combination of standard mark-up tags, using XHTML (the template, highlighted in bold text), and other tags, specific to the Java server side environment, that generate

Figure 5. The 'Web 1.0' application architecture, based on the Model 2 and Template View patterns



dynamic content. These tags are a combination of standard JSP tags ('jsp:root,' 'jsp:useBean,' 'jsp:getProperty') and the 'if' tag from the JSP Standard Tag Library (JSTL).

Behind the view/controller layer, the mapping between the data store and the model is the responsibility of data access objects (DAOs) that interact with the higher levels of the system via data transfer objects (DTOs) (Alur et al., 2003). In most real world cases the data store is a relational database, requiring some object relational (O/R) mapping, enabling the structured query language (SQL) to be used to read and write persistent data. The DTO components that encapsulate updates or query results can be embedded into server pages using appropriate tag libraries.

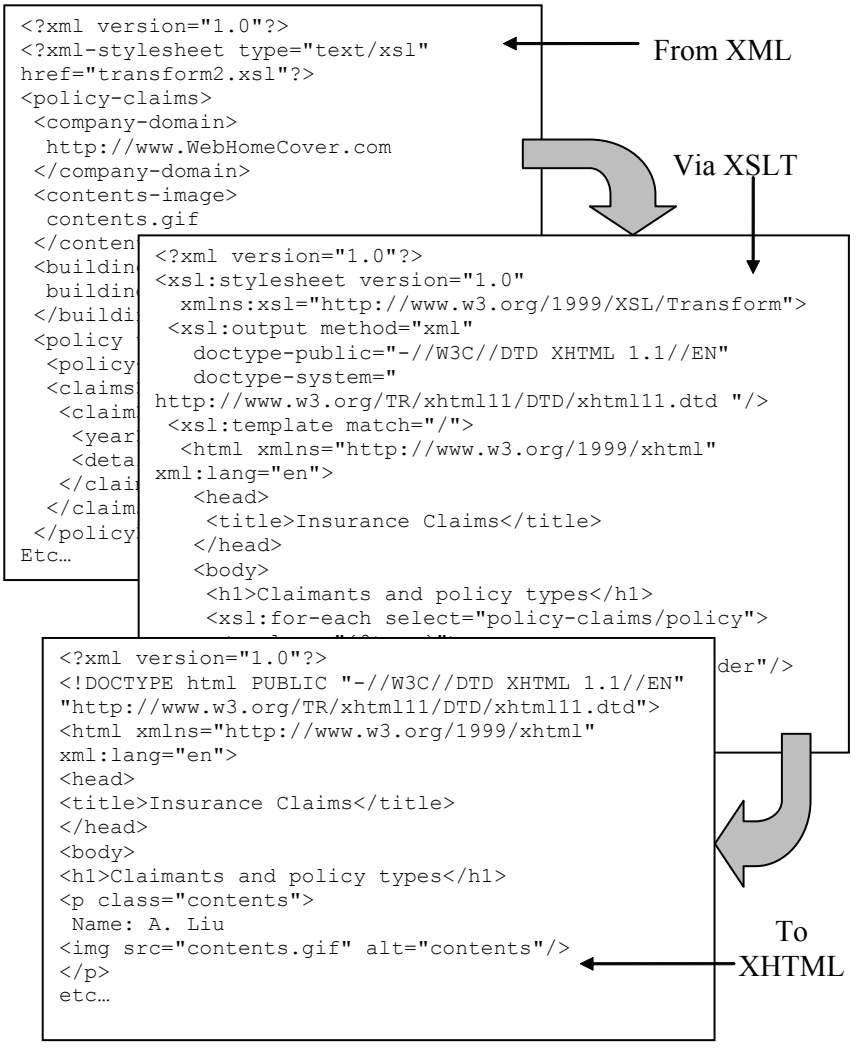
Figure 5 illustrates the basic 'Web 1.0' architecture. Initial routing of all hypertext transfer protocol (HTTP) requests from Web clients is handled by the front controller component that receives all the requests, parses their parameters and delegates to the appropriate object model layer components via command objects (1). There are a number of frameworks that encapsulate this design pattern, including Struts and JavaServer Faces. In both cases, the front controller component is a Java servlet. Any interaction between the model and persistent data is done via the DAO/DTO layer (2), which interacts with the underlying data store (3). Content is returned in DTOs that can be processed by the object model layer and/or embedded into server page components using tag libraries (4). These server page components can then generate a dynamic response (5). In "traditional" Web applications, that response has typically been in the form of HTML.

The XML Transformation Model

More recently, XML (extensible markup language) based technologies have become popular in Web application architectures, in no small part due to the limitations of HTML, which is purely a page mark-up syntax and cannot be used to represent data at any level of abstraction, whereas XML can be used to represent semi-structured data that is not restricted to page mark-up (Abiteboul, Buneman, & Suciu, 2000). Certain principles underlie the use of XML in a Web application. Its role is essentially to provide services that cannot be supported using more traditional approaches to Web applications based on representing content directly in HTML. There are four broad categories where XML provides such services (Bosak, 1997):

- Applications that require the Web client to mediate between two or more heterogeneous databases;

Figure 6. The Transform View pattern, where a source document is transformed into a client page using an XSL Transformation



- Applications that attempt to distribute a significant proportion of the processing load from the Web server to the Web client;
- Applications that require the Web client to present different views of the same data to different users;
- Applications in which intelligent Web agents attempt to tailor information discovery to the needs of individual users.

Meeting these requirements means that a Web application that uses XML needs a more complex server side architecture than the simple template model, and data has to be represented in, and translated between, multiple formats, which may include XML, object model and database level representation. Whereas the template view pattern assumes a component based plugging-in of object model components into a server page

template, the XML based model requires some kind of transformation to take place from data, taken from some source and represented by XML, into mark-up that can be interpreted by the client. Fowler (2003) characterises this approach as the *Transform View* model where, instead of using a server page to embed object model components into a page template, an extensible stylesheet language (XSL) processor is needed to transform an XML document into a generated client page. In order to make this transformation, the XSL processor applies an XSL transformation (XSLT) stylesheet that defines how the input document is to be transformed.

Figure 6 shows how an XML document is transformed by an XSL Transformation into an XHTML client page. This transformation can take place either on the client or the server, depending on whether the client is able to process the transformation (most desk top browsers are able to do this).

The main problem with executing transformations on the client, even assuming the client is able to perform those transformations, is that they will provide static content, since the assignment of the transforming style sheet will be hard coded into the XML document. Alternatively, we would have to rely on the client providing the necessary scripting and DOM support to perform the dynamic transformation. In contrast, a transformation executed on the server can be applied dynamically and reliably, independent of the client. Listing 3 shows an example of how such transformations can be executed on the server using tags from the JSP Standard Tag Library. Specifically, the 'transform' tag from the JSTL XML library applies an XSL transformation style sheet to an XML document.

A simplified visualisation of the distinction between the template and transform approaches in shown in Figure 7. In the template view, the server page integrates components from the object model into a mark-up based template. The page template is embedded in the server page itself. In the transform approach the XSL processor integrates elements from XML documents into a mark-up based transformation, with the page template embedded in the transformation style sheet. If the transformation is performed on the server, the server page simply triggers the transformation process.

XML in Web Applications

Bosak's (1997) four roles of XML can be supported in various ways by the transform model. Mediation between heterogeneous databases can be seen from more than one perspective. In some

Listing 3. The Transform View pattern implemented on the server using the JSP Standard Tag Library

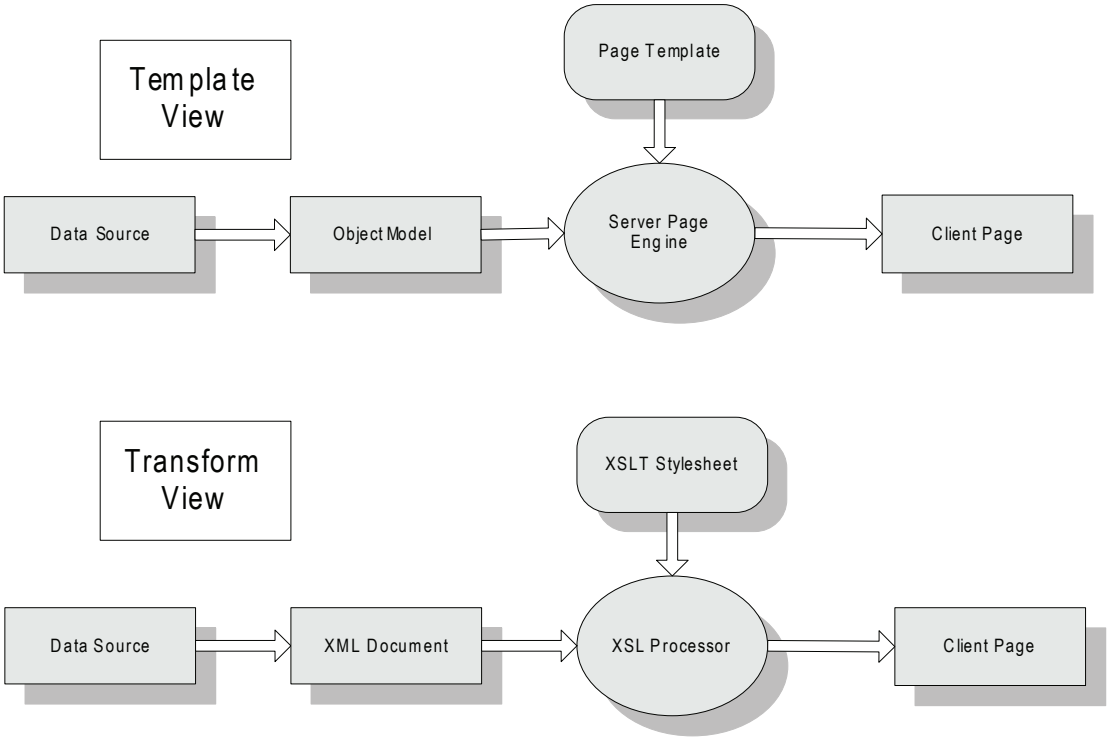
```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
   xmlns:c="http://java.sun.com/jsp/jstl/core"
   xmlns:x="http://java.sun.com/jsp/jstl/xml"
   version="2.0">

 <jsp:directive.page contentType="text/html"/>
 <jsp:output omit-xml-declaration="true" />

  <c:import url="policies.xsl" var="stylesheet"  />
  <c:import url="policies.xml" var="xmldocument"  />
  <x:transform doc="${xmldocument}" xslt="${stylesheet}" />

</jsp:root>
```

Figure 7. The Template View and Transform View patterns compared



cases, XML Web services can be used to integrate the various data sources of a single organisation or collaborating group of organisations. In this sense XML can be seen as the natural successor to more traditional forms of electronic data interchange (EDI). In other cases, such as those included in the patterns of the Web 2.0, there is the concept of specialised databases being the core item of value in a system, with the opportunity to reuse these databases in new combinations of services. In this type of scenario the databases in question may be distributed over a wide range of providers, with no formal relationship or set of contracts between them. Here, the XML Web services are in the public domain. From any of these perspectives, raw XML data may be the way of communicating between systems, but transformations are necessary to apply queries or filters to the data and transform it to meet the specific requirements of a client system.

There are various ways that XML can be used to offload some of the processing from the server to the client, but again some aspect of transformation is essential. XML documents provide a presentation neutral way of representing the underlying content of an application, but the typical role of the Web client is to render that data in an appropriate way. Tools for this type of transformation, such as XSLT, are therefore an essential component of an XML-centric Web application architecture. This is what underlies the concept of the transform model, since XSLT is able to express how one XML document may be transformed into another document, which may be XML or may use a different syntax such as XHTML. The way that processing can be offloaded from the server to the client ranges in sophistication from the simple separation of XML data, XSL transformations and CSS (cascading style sheets, used to style the presentation of a document) to Ajax applications. In the former case,

we rely on the ability of the browser to download, cache, and process by transformation and styling, XML documents. Since the same transformations and style sheets can be used with multiple XML documents from similar data sources (i.e., related but different queries across the same data source) all that is required of the server is to send the XML documents to the client for processing. In the latter case, various Ajax tools can be used to develop rich client processes based on JavaScript and asynchronous XML messaging. This is a more complex approach, since instead of relying on the browser's native ability to process XSLT, we are relying on a JavaScript application to support client side processing and parsing of XML documents. While it is possible to offload some presentational style sheet processing to small devices using this kind of markup, not all mobile clients are able to manipulate XML documents using XSL Transformations or by using client side scripting languages like JavaScript. However, many small devices can support the use of Java Micro Edition to manipulate XML documents on the client, using small footprint parsers such as kXML. Therefore an Ajax-style approach is possible even in the absence of JavaScript.

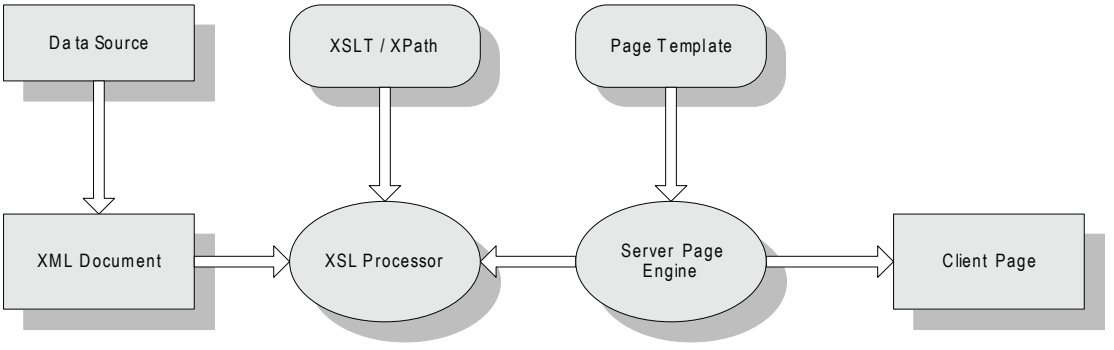
Where it is not possible to offload the processing necessary for XML transformations onto the client, the server can take responsibility for enabling the Web client to present different views of the same data to different users. One of the useful ways that we can apply XSL transformations of XML data in a Web application is to provide different transformations for different types of client device. For example, the same content, represented in XML, could be transformed into XHTML for desktop browsers, wireless markup language (WML) for 'legacy' mobile phones, XHTML-MP for more recent mobile phone browsers, and so on. The same techniques can be use for personalisation or customisation of the content itself, targeting the user as well as the device

Finally, agents can utilise XML because it provides metadata about Web-based content. This aspect of XML is what supports the concept of the semantic Web, enabling a greater degree of reflection about Web-based information sources on the part of Web applications. Although discussion of the Semantic Web is beyond the scope of this chapter, XML and XML Schema are important foundation technologies that support the higher level aspects of the Semantic Web technology stack such as the resource description framework (Hendler, 2001).

The Limitations of the Transform Model

Whilst transformations, used directly, can have certain advantages such as the ability to leverage the common abilities of browsers to perform XSL transformations on the client, on their own they have a number of drawbacks. First, the syntax of the XSL transformation is complex, and can be difficult to maintain, particularly where the "apply-templates" approach is used to transform the XML documents using pattern matching. Second, using a transformation directly cannot easily be integrated with other server side processes, even if the transformations are actually executed on the server rather than the client. The advantage of the XSLT approach is that a relatively small set of transformations can be used to manage a large number of XML documents in a Web application. However they are not particularly flexible. In contrast, using a tag-based template approach gives us a highly flexible way of integrating both mark-up and programmatic logic. Passani & Transatti (2002) argue that XSL Transformations are a very poor approach to providing adaptive mark-up for different devices, because they mean writing separate transforms for each type of client. In contrast, template based tools such as the wireless abstraction library (WALL) enable us to encapsulate different page generation behind processing tags (Passani & Transatti, 2002). It is possible, however to use parts of XSL combined with tags to gain the benefits of both approaches.

Figure 8. Combining transform and template views



Listing 4. Combining the Template View and Transform View approaches in a JSP

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:c="http://java.sun.com/jsp/jstl/core"
  xmlns:x="http://java.sun.com/jsp/jstl/xml"
  version="2.1">
 <jsp:directive.page contentType="text/html"/>
 <jsp:output omit-xml-declaration="false"
 doctype-root-element="html"
 doctype-public="-//W3C//DTD XHTML 1.1//EN"
 doctype-system="http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd" />

 <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
  <head>
   <link href="webhomecover.css" rel="stylesheet"
    type="text/css" />
   <title>WebHomeCover Insurance Policies</title>
  </head>
  <body>
   <div>
    <!— main page presentational content here,
        including XPath expressions, e.g.
    —>
    <c:import url="./policies.xml" var="xmldocument"/>
    <x:parse doc="${xmldocument}" var="xml"/>
    <x:out select="$xml/policies/policy/policy-holder"/>

   </div>
  </body>
 </html>
</jsp:root>
```

Rather than transforming documents in a single process, we can combine partial transformations with template based mark-up. This can help us to gain the benefits of XML in Web applications without losing the programmatic flexibility of the template model. However the key is to combine both transformations and templates into a single server page to ensure adequate performance. It is technically possible to generate a server page, marked up using device adaptive tags, from

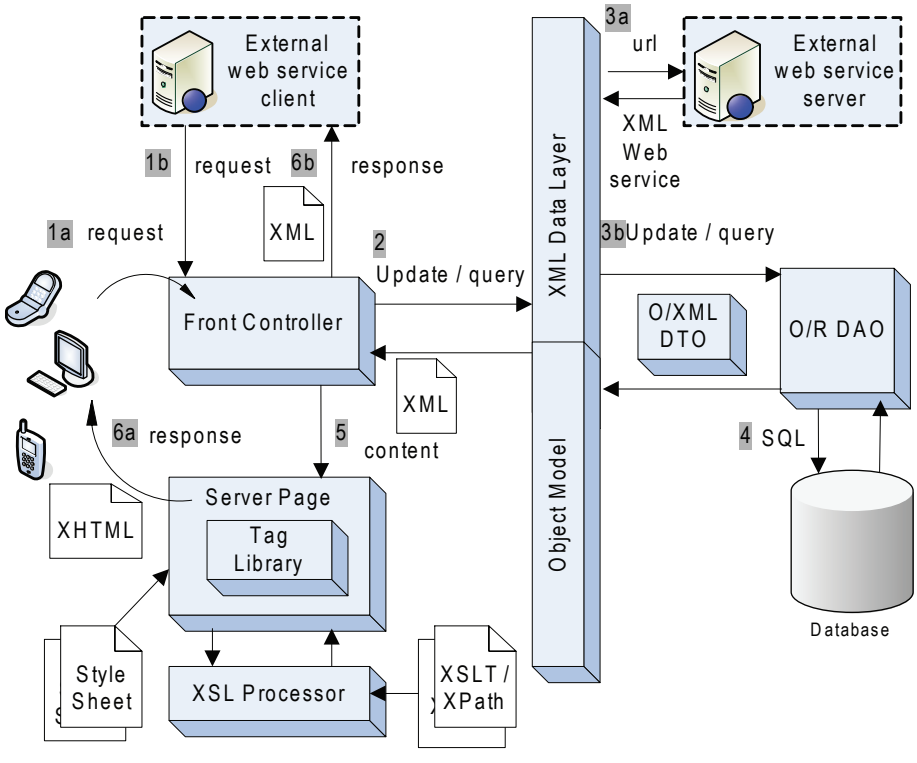
a transformation, and then run that generated page through the server page engine. However this two stage process can be very inefficient. The alterative approach is to integrate partial transforms, for example using XPath filters on the XML data, within the same server page as the adaptive mark-up. This means a single pass through the server page (Figure 8).

The code extract in Listing 4 shows the basic principles of this approach. The template view provides the main structure of the page and generates the XHTML tags that build the presentation. Within this page, however, XPath expressions are applied to an XML document to insert dynamic content. One of the main contrasts between the transform and mixed template and transform approaches is that the responsibility for the XHTML mark-up is in the XSLT document in the transform approach, but in the server page in the combined template/transform approach.

WEB 2.0 GENERATION APPLICATION ARCHITECTURE

In the previous sections we introduced some of the architectural features of Web applications that may use either template or transform approaches, or a combination of both. In this section we describe a Web 2.0 generation architecture for adaptive Web applications that leverages XML as a data representation format within an adaptive architecture, integrates service oriented aspects, and combines both template and transform elements. Perhaps the most important feature of this architecture is the introduction of an XML data layer that integrates

Figure 9. A Web application architecture integrating XML



disparate data sinks and sources into a cohesive XML management process at the same level as the object model. This enables a system to manage requests and responses in a consistent way, regardless of whether the client request is for a Web service or presentation mark-up, and whether our data sources are local databases or remote Web services. An outline of the architecture and its core processes is shown in Figure 9.

In this architecture, client requests to a front controller component may come either from presentation oriented devices (1a) or external Web service clients (1b). The request is delegated to a joint XML data and object model layer (2). Depending on the requirements of the request, queries delegated via the XML data layer may either retrieve XML content from one or more external services running on other Web servers (3a), generate XML content from a local database via suitable queries (3b) or perhaps combine multiple data sources. Where content is based on external Web services, content will already be in XML format. Where content is held locally in a database that is not natively XML, some kind of transformation will be required between relational and XML data. Rather than implementing such a direct transformation, which would provide only XML to the controller layer, it is probably wise to maintain the data transfer object (DTO) pattern, but add XML generation capabilities to the DTOs. This approach means that the DTOs can be used both as objects in their own right within the object model of the underlying entities in the system and as sources of XML documents. A layer of object/relational data access objects (DAOs) is required to perform the translation from relational data to object model and create the XML generating DTOs (4). Multiple XML documents may be generated from a model that is created from a single database query, since multiple object level queries can be used against any graph of DTOs. The XML content returned from the XML data layer (5) is transformed via an XSL process utilizing XPath queries to build parts of a document. These XML path queries should be performed via a server page using tag libraries that integrate both XML data retrieval and adaptive markup to ensure encapsulation and reuse. Once the content is transformed it can be transferred to the device in the generated mark-up format for processing by the client. This may be a device oriented mark-up (6a) or a Web service (6b). Of course many mobile devices can both process mark-up and act as end points for Web services. It should be noted that services may take different forms. For example, a server side component may provide XML messaging service for Ajax clients by using the mechanism described above, whereby program components generate XML documents and stream these to JavaScript components on the client. However Web services may take a more heavyweight form, using standards such as Web services description language (WSDL), simple object access protocol (SOAP) and universal description, discovery and integration (UDDI). In this case, generation of XML from programming components will typically be done via tool support to generate the necessary XML documents, stubs and skeletons to enable client/server interoperability. The current generation of Java Web service tools are based on the Java API for XML Web services (JAX-WS).

This type of XML centric architecture has a number of benefits. First, we can gain reuse of both external Web services and local components. Second, we can provide adaptivity that integrates both client device mark-up and Web service endpoints. From a business perspective it provides maximum leverage of external services and APIs while gaining maximum potential distribution across all possible client types. Encapsulation of data access and transformation should provide benefits in terms of maintainability and cost. For example, separating out concerns between XML, object models and the database, rather than directly generating XML from the database, can assist in the reuse of legacy data sources and provide added security through additional layer-

ing. In terms of the Web fragmentation cycle, the architecture is based on common XML, Java and SQL standards while integrating some aspects of innovative Web 2.0 service based and cross-device architectures.

Integrating Java and XML

So far we have introduced a number of features of Web application architecture relating to either XML or Java, but indicated that a Web application architecture need to be able to support data objects that can convert to and from Java and XML. In this section we discuss some aspects of how Java and XML can be integrated, including some standard tools. In a Java based architecture, the O/XML DTOs should be based on the relevant parts of the JavaBean specification that enable them to be used via tag libraries in JSPs, and be manipulated by the JSP Expression Language. These beans provide both an object model to organise the requested set of data and a way of generating XML documents from one or more beans. This is particularly useful where a client page will include data from multiple tables that have some kind of associative relationship, realised in the application as a related graph of objects. Generating XML from JavaBeans can be done using frameworks or custom code. Geary (2001), for example, provides some guidance on how to implement JavaBeans that can generate XML. In cases where the XML content encapsulates nested elements from multiple domain components, the beans will form a composite pattern (Gamma et al., 1995) with the composite objects organising sub elements and the leaf objects generating element content. Listing 5 shows a simple implementation of this pattern, where the class that implements the 'getXmlElement' method (representing an insurance policy) is associated with a collection

Listing 5. Generating XML content from composite JavaBeans

```
public String getXmlElement(boolean graph)
 {
  StringBuffer element = new StringBuffer();
element.append
("<policy policy-number=\"" + getPolicyNumber() + "\">");
element.append
("<start-date>" + getFormattedDate() + "</start-date>");
element.append
("<annual-premium>" + getAnnualPremium() +
"</annual-premium>");
element.append
("<number-of-claims>" + claims.size() +
"</number-of-claims>");
  element.append("<paid-up>" + getPaidUp() + "</paid-up>");
  if(graph)
  {
   element.append("<claims>");
   Iterator<Claim> claimIter = claims.iterator();
   while(claimIter.hasNext())
   {
    Claim claim = claimIter.next();
    element.append(claim.getXmlElement());
   }
   element.append("</claims>");
  }
  element.append("</policy>");
  return element.toString();
 }
```

of 'Claim' objects that generate their own XML content via a call to a polymorphic 'getXmlElement' method. Each of these methods generates a fragment of a larger XML document.

The link between the JavaBeans and the database will depend on the nature of the database and the transactional requirements of the persistence layer. In some cases a simple manual mapping using JDBC may be adequate, but it is likely that frameworks implementing standards such as the Java Persistence API will be required for industrial strength persistence. How the JavaBeans interact with the XML layer will also depend on the requirements of the application. As we have described, manual solutions are possible, but it may be more appropriate to use tools such as the open source XMLBeans (Apache Software Foundation, 2006) or Java XML binding (JAXB) which is integrated into the standard Java platform from version 6 onwards. The choice here will often depend on whether the application is mainly driven by an object model or by XML documents, since frameworks like XMLBeans and JAXB depend on the data model being derived from XML Schemas, and Java objects being generated subservient to that XML data model. Another important consideration is the generation of Java objects from XML documents. This can be done manually using the Java API for XML processing (JAXP) but is perhaps better performed by tools like JAXB, particularly since more recent versions of JAXB, unlike the original implementation, can convert both from XML to Java and from Java to XML. The partial code example in Listing 6 shows how JAXB can be used to generate XML documents from Java objects via a 'Marshaller,' in this case a collection of 'Policies' objects. The collection of Policies is derived from an ObjectFactory class generated by the JAXB framework.

Once XML documents are generated from the JavaBeans layer, XML processing should be performed via JSPs using tags from libraries such as the JSP standard tag library (JSTL), which includes a dedicated sub-library for XML processing. However there are a number of other XML processing tag libraries available in Java that may be used instead. Listing 7 shows the JSTL being used for a simple XSL transformation—similar to listing 3—but this time, not using a static XML document. A JavaBean is used to generate an XML document which is transformed by a style sheet using JSTL tags. In this code example we assume that the getXmlDocument method is responsible for marshalling the various components of a complete XML document, in contrast to the

Listing 6. Transforming objects into XML documents using JAXB

```
ObjectFactory policiesFactory = new ObjectFactory();
Policies policies = policiesFactory.createPolicies();
JAXBContext ctx = JAXBContext.newInstance(Policies.class);
…
Marshaller m = ctx.createMarshaller();
m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
m.marshal(policies, System.out);
 try
 {
  OutputStream os = new FileOutputStream (new File("newpolicies.xml"));
  m.marshal(policies, os);
 }
 catch(Exception e)
 {
  e.printStackTrace();
}
```

Listing 7. Using the JSTL tags for processing of JavaBeans that generate XML

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:c="http://java.sun.com/jsp/jstl/core"
  xmlns:x="http://java.sun.com/jsp/jstl/xml"
  version="2.0">

 <jsp:directive.page contentType="text/html"/>
 <jsp:output omit-xml-declaration="true" />

 <c:import url="claims.xsl" var="stylesheet"  />
 <x:transform doc="${claim.xmlDocument}" xslt="${stylesheet}"/>

</jsp:root>
```

getXmlElement method we saw in Listing 5, which generated a fragment of a larger document. Using the JSP Expression Language, 'claim.xmlDocument' refers to the getXmlDocument method of a JavaBean stored in the scope of the Web page under the lookup name 'claim.'

Adapting to Client Devices

If an architecture is to be flexible enough to manage multiple types of client browsers, the server side page generation has to be able to adapt to different types of client device, which can be identified by the 'User-Agent' field in the HTTP request header. If an XML transform layer is in place, the underlying content can easily be adapted, but how it is actually adapted is another question. Fortunately, there are a number of Java-based technologies that can assist us in adapting server-side content to the client device without manually interrogating the user-agent field. One example is the wireless abstraction library (WALL), a JSP tag library that builds on the wireless universal resource file (WURFL) (Passani & Trasatti, 2002). WURFL is an XML database that is able to map user agent information to the capabilities of the originating device, while WALL generates device specific mark-up. JSP pages based on the template view model, using WALL syntax, can be integrated with XML elements using the transform view model by applying tags (such as those from the JSTL) that enable XPath queries to be applied to an XML document and the results included in a server page. During the page processing, the WALL tags will be turned into the appropriate mark-up for the client device while the XPath elements will provide the content from the source XML document. Listing 8 shows part of a server page that incorporates WALL tags to generate adaptive mark-up, along with XPath queries using the JSTL. The key difference between the XML processing in Listing 7 and that in Listing 8 is that the former uses the transform view pattern, whereas the latter uses the combined template and transform view pattern outlined in Figure 8.

By using the WALL device aware library we are able to provide a single server page for many different types of device. However we might in addition choose to provide our own customised transform for particular types of device. For example, a transform could be used that would take advantage of the rich client technologies available on the desktop. Many Web 2.0 applications leverage Ajax and Flash, though there are many other possibilities. At the least, we might apply a cascading style sheet (CSS) to manage

Listing 8. Combining the WALL library with the JSTL to generate adaptive mark-up

```
<%@ taglib uri="http://java.sun.com/jstl/xml" prefix="x" %>
<%@ taglib uri="/WEB-INF/tld/wall.tld" prefix="wall" %><wall:document>
<wall:xmlpidtd />
<wall:head>
 <wall:title enforce_title="true">Claim Display</wall:title>
</wall:head>
 <wall:body>
  <wall:block>
  <wall:br />
  <jsp:useBean id="claims" class="webapp.classes.ClaimsBean" />
   <x:parse doc="${claims.xmlDocument}" var="xml"/>
   <x:forEach select="$xml/claims/claim">
    Claim Description: <x:out select="description"/>
    <wall:br/>
    Claim amount: $<x:out select="amount"/>
    <wall:br/>
   </x:forEach>
  </wall:block>
 </wall:body>
</wall:document>
```

the presentation in the browser. Where the client is a service end point, and the system provides content as a Web service, direct XSLT transforms can be used, since they would not be generating presentation mark-up but XML documents suitable for Web service use.

CONCLUSION

In this chapter we have described how Web applications have evolved from static content, through dynamic content based on a server page template model, to contemporary architectures that rely heavily on the transformation of XML documents and increasingly complex client side applications. We have discussed this evolution within a model that shows how Web technologies tend to fragment as a result of innovation, and then find broader reach as a result of standardisation. To develop Web applications in this context we need to be aware both of new technological developments and also how generic standards support is evolving. For example we may wish to develop Ajax applications but also deliver cross platform applications to both desktop and mobile clients. To be innovative while broadening reach as much as possible it is necessary both to work within well supported standards (e.g., parts of the DOM) but also to leverage tools that provide flexibility across platforms, whether it be increasing support for certain technologies on multiple platforms (e.g., JavaScript support in mobile browsers) or tools that adapt themselves to different clients (e.g., tag libraries for adaptive mark-up.) To support the integration of different approaches and technologies, a highly flexible architecture that leverages patterns, layers and XML centric data management is necessary. In this chapter we have described a reference architecture that is based on Java, XML and server side tag libraries that supports the transformation of local or service based content into client-adaptive output formats, using a combination of the template view and transform view design patterns. This architecture takes into account both current thinking on the technologies of Web-based applications and the encapsulation and reuse of standard libraries.

Building Web applications based on this architecture should enable developers to gain maximum flexibility and reach without relying too heavily on proprietary tools.

REFERENCES

Abiteboul, S., Buneman, P., & Suciu, D. (2000). *Data on the Web - From relations to semistructured data and XML*. San Francisco: Morgan Kaufmann.

Alur, D. Crupi, J., & Malks, D. (2003). *Core J2EE patterns: Best practices and design strategies*, 2nd Edition. Upper Saddle River, NJ: Sun Microsystems Press / Prentice Hall.

Apache Software Foundation. (2006). *Apache XMLBeans*. Retrieved on July, 2007, from http://xmlbeans.apache.org/index.html

Berners Lee, T. (2004). *New top level domains .mobi and .xxx considered harmful*. Retrieved on January, 2007, from http://www.w3.org/DesignIssues/TLD

Bosak, J. (1997). *XML, Java, and the future of the Web*. Retrieved on July, 2007, from http://www.ibiblio.org/pub/sun-info/standards/xml/why/xmlapps.htm

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). *Pattern-oriented software architecture: A system of patterns*. Chichester: Wiley.

Fowler, M. (2003). *Patterns of enterprise application architecture*. Boston: Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. 1995. *Design patterns: Elements of reusable object-oriented software*. Reading, MA.: Addison-Wesley.

Garrett, J. (2005). *Ajax: A new approach to Web applications*. Retrieved on July, 2007, from http://www.adaptivepath.com/publications/essays/archives/000385.php

Geary, D. (2001). *Advanced JavaServer pages*. Upper Saddle River, NJ: Sun Microsystems Press / Prentice Hall.

Hendler, J. (2001). Agents and the Semantic Web. *IEEE Intelligent Systems, 16*(2), 30-37.

Koch, P. (2006). *ppk on JavaScript*. Berkeley, CA: New Riders.

Le Hégaret, P., Whitmer, R., & Wood, L. (2006). *Document object model (DOM)*. Retrieved on July, 2007, from http://www.w3.org/DOM/Overview

Lie, H. W., & Bos, B. (1999). *Cascading style sheets: Designing for the Web,* 2nd edition. Harlow, England: Addison Wesley Longman.

O'Reilly, T. (2005). *What is Web 2.0: Design patterns and business models for the next generation of software*. Retrieved on July, 2007, from http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html

Passani, L., & Trasatti, A. (2002). *WURFL*. Retrieved on July, 2007, from http://wurfl.sourceforge.net/

Seshadri, G. (1999). Understanding JavaServer pages model 2 architecture: Exploring the MVC design pattern. *JavaWorld*, *December*.

Van Eaton, J. (2005). *Outlook Web access - A catalyst for Web evolution*. Retrieved on July, 2007, from http://msexchangeteam.com/archive/2005/06/21/406646.aspx