Server-Centric Web Frameworks: An Overview

IWAN VOSLOO

Reahl Software Services (Pty) Ltd

and

DERRICK G. KOURIE

University of Pretoria

Most contemporary Web frameworks may be classified as server-centric. An overview of such Web frameworks is presented. It is based on information gleaned from surveying 80 server-centric Web frameworks, as well as from popular related specifications. Requirements typically expected of a server-centric Web framework are discussed. Two Web framework taxonomies are proposed, reflecting two orthogonal ways of characterizing a framework: the way in which the markup language content of a browser-destined document is specified in the framework (presentation concerns); and the framework's facilities for the user to control the flow of events between browser and server (control concerns).

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Software libraries*; *user interfaces*

General Terms: Design, Languages

Additional Key Words and Phrases: Architecture, specification techniques, user interfaces, web frameworks

ACM Reference Format:

Vosloo, I. and Kourie, D. G. 2008. Server-centric web frameworks: An overview. ACM Comput. Surv. 40, 2, Article 4 (April 2008), 33 pages DOI = 10.1145/1348246.1348247 http://doi.acm.org/10.1145/1348246.1348247

1. INTRODUCTION

A Web framework is a collection of software components that support the development and execution of Web-based user interface (UIs): UI that are presented to users at remote locations via their Web browsers. Software developed by the UI programmer resides on the server and is executed either on the server or the client browser. It determines the sequence of UI pages to be displayed on the browser, the content of these pages, actions available to the user, the messages to be exchanged between the

The authors would like to thank Reahl Software Services (Pty) Ltd. for funding this research.

Authors' addresses: I. Vosloo (corresponding author), Reahl Software Services (Pty) Ltd., PO Box 14240, Hatfield, Pretoria, 0028, South Africa; email: iwan@reahl.org; D. G. Kourie, Espresso Software Engineering Research Group, Department of Computer Science, University of Pretoria, Pretoria, 0002, South Africa.

back-end application and the server, etc. The syntax and semantics of this software is dictated by the particular framework that is used.

Web frameworks may be classified broadly as being either server-centric or browser-centric. In the former case, the locus of control, that relates the current state of computation to the details of the UI to be displayed, is embedded in software executed at the server. In the latter case, this locus of control shifts mostly to the browser. Until recently, almost all Web frameworks were server-centric.

This article is limited to an overview of server-centric Web frameworks. It is based on insights gleaned from a survey of 80 frameworks and a number of related specifications in the Java 2, Enterprise Edition^{TM} (J2EE) family.

Frameworks used in the survey are mostly open-source products for which the design and other technical information are readily available. Many of these are in wide use in industry and many proprietary frameworks employ the same techiques.

To provide a better understanding of Web framework functionality, a list of requirements typically expected of Web frameworks is given in Section 4.

Two taxonomies have been inferred from the survey, reflecting two orthogonal ways of characterizing server-centric frameworks.[1] Section 5 classifies the different ways in which the markup language content of a browser-destined document is specified in server-centric frameworks. Section 6 classifies server-centric frameworks according to the facilities they provide to the programmer for controlling the flow of events (and responses to them) between the UI and application. These two orthogonal characterizations correspond, respectively, to the view and control components of the well-known model-view-controller (MVC) architectural design pattern [Krasner and Pope 1988; Gamma et al. 1995].

The relevance of client-side execution is discussed in Section 7. Before concluding and suggesting directions for future research in Section 9, some problems involved in executing such specifications are briefly mentioned in Section 8. A list of the projects surveyed is supplied in Appendix A.

2. SCOPE

The taxonomies presented here are intended to enhance understanding of the field and to serve as a reference when analyzing a new Web framework. Although many of the surveyed frameworks are cited as examples in the taxonomies,[2] a full elaboration of the various features of each these frameworks, in tabular form or otherwise, is beyond the scope of this study.

Furthermore, in principle, a third taxonomy implied by the MVC architectural design pattern could be devised, namely a taxonomy relating to model concerns. Indeed, model concerns are addressed to a greater or lesser extent in many frameworks. Such support is generally in terms of facilities to transparently persist data, or to provide a mapping between programming language objects and the relational databases in which these objects were stored. (One example is the Enterprise Java Beans^{TM} (EJB) specification [DeMichiel 2003].) This is a large field in its own right and very much on the periphery of the focus here. Hence, a taxonomy based on model concerns is not addressed in this study. View and controller concerns, in contrast, are cornerstones in the design of a UI.

[1]It should be noted that neither taxonomy classifies frameworks directly. Individual frameworks employ a number of strategies for dealing with certain problems. The taxonomies classify those strategies. Frameworks often combine strategies.

[2]Surveyed project names are indicated in the text with a special font, for example, *Struts*. The relevant projects are also cited as usual and included in the Reference section.

Browser-centric frameworks need a vehicle at a client browser for the execution of their code. Many current browser-centric frameworks provide their own proprietary browser plugin for this purpose.

The scope of the survey is limited to frameworks that employ standardized[3] technologies to accomplish their effect. As a result, since so many interesting examples of browser-centric frameworks[4] rely on proprietary technology, this class as a whole is excluded.

The only *standardized* mechanism available for client-side execution of code is ECMAScript: the standard to which JavaScript implementations can adhere [ECMA General Assembly 1999]. Because of discrepancies between the ECMAScript standard and varying JavaScript implementations provided by individual browsers (and versions of browsers), JavaScript-based frameworks were not really viable in practice for quite some time.[5] As a result, examples of such frameworks are still very immature compared to server-centric frameworks which do not *depend* on JavaScript.

The focus on standardized technologies, coupled with the relative immaturity of standards-compliant browser-centric frameworks, naturally leads to the exclusion of browser-centric frameworks as a whole (more about this in Section 7).

In the rest of this article, the term "Web framework" is used in the sense of *server-centric* Web frameworks unless otherwise stated.

3. RELATED WORK

A few authors mention functionalities that are related to requirements of a Web framework. These are mostly mentioned in different contexts. For example, in Helman and Fertalj [2003] an overview is given of "[W]eb application generators": programs which generate a Web application based on a higher-level specification. Partly, the target of such generation is a specification technique dictated by a Web framework. As part of this work, a list is given of key features of Web applications. In Copeland et al. [2000] four Web development products are compared. Some of these products are in fact Web frameworks. The authors partly base their comparison on several proposed "categories of [W]eb development functions." Rode et al. [2004] presents an empirical study of end-user's mental models concerning several "concerns in [W]eb application development"; these concerns are enumerated in the article. In Westkämper [2004] an overview is given of the details of four prominent Java-based Web frameworks, discussed in terms of which concerns they address.

Not all items on the lists of features or concerns given by these authors are applicable to the presentation of a Web-based UI. Some of them address the modeling of an application as a whole (its database design, for instance). Several items are relevant, though, and contribute towards the list of requirements of Web frameworks proposed in this article (presented in Section 4).

A small number of articles have been published that compare Web development tools or techniques related to Web frameworks. As mentioned before, Westkämper [2004] gives an overview of the workings of four Java-based Web frameworks. In Draheim and Weber [2002], the authors present their ideas regarding "[W]eb interface programming based on the Model 2 architecture." According to the title of the report, it is an overview of architectures; on closer inspection, though, this technical report merely briefly

[3]Standardized here means standardized by an organization such as the World Wide Web Consortium (W3C).
[4]Recall that in browser-centric frameworks, most of the software that relates the current state of computation to the details of the UI to be displayed resides in the browser.
[5]With the latest browsers, the supplied JavaScript implementations have grown sufficiently similar to spur the development of frameworks that rely on JavaScript. These frameworks are, however, very immature compared to their server-centric counterparts that have been used widely for some years.

explains (in passing) the architecture colloquially known in Java circles as the “Model 2 architecture.” (Such “architectures” relate to the design of Web frameworks.) Four Web development tools are compared and discussed in Copeland et al. [2000]. Most of these tools are either Web frameworks or embed Web frameworks. The tools are discussed mainly in terms of functionality offered. None of these titles provides one with a structured overview of Web frameworks (or of Web development tools in general). In Fraternali [1999], a more complete survey is presented of Web development tools. This study categorizes Web development tools into six distinct categories, such as “visual editors and site managers,” “hypermedia authoring tools,” or “model-driven application generators.” Most of the tools discussed still need a low-level Web application framework for presenting the final UI, but Web frameworks are not explicitly mentioned as such. A few Web frameworks are, however, discussed under the heading of “Web-Database Programming Language integrators.” Web frameworks and topics regarding them have proliferated in the past few years since publication of Fraternali [1999]; perhaps had the article been written today, this category would have been generalized and named “Web frameworks.”

Here and there, articles are to be found introducing Web frameworks with novel ideas (e.g., Crespo et al. [1997]) or for specific types of applications (e.g., Qin et al. [2003]).

Several authors are actively proposing methods of modeling Web applications; especially authors from the model-driven development camp [Ginige and Murugesan 2001; Ceri et al. 2000; Knapp et al. 2003; Güell et al. 2000; Winckler and Palanque 2003; Schranz et al. 2000; Koch and Kraus 2002]. Model-driven development is centered around the concept of building abstract models of a system, and then generating the final implementation of the system from the models [Mellor et al. 2003; Thomas 2004]. Usually, models are constructed on a high level, and then transformed successively into lower-level models (with increasing detail) until a representation is reached which can be executed. Model-driven development is thus usually very tied to an overall development process which covers a spectrum from requirements analysis up to implementation. The notations proposed by these approaches for modeling UI-related aspects of a Web application are interesting from the viewpoint of the present article, since they include specification techniques of Web-based UI. All these approaches, however, eventually end in the generation of a specification which is partly in terms of an existing Web framework in order to be able to execute a UI. Despite this, Web frameworks are typically not mentioned in this literature.

Thus, while some literature touches on what can be construed as requirements for Web frameworks, the literature seems quiet when it comes to specification techniques for Web-based UI that are directly executable and how this execution is accomplished in the execution environment represented by a Web server and its Web browser clients.

In contrast, ideas and arguments regarding these topics abound in the form of thousands of commercial and open-source Web frameworks, informal blogs, news groups, and other unmoderated Web-based discussion forums.

This proliferation of discussions and solutions may be construed as an indication that the problem (of finding the right abstractions with which to implement Web-based UI) has not been solved satisfactorily in practice. However, it may also be that the problem merely has a great number of variable parts, and that it needs to be partitioned more usefully.

Whatever the case may be, there does not appear to be a clear overview in the literature of what is happening in this space (even though the authors of related topics that *are* discussed in the literature implicitly assume the availability of Web frameworks). It would seem that the current development of the field happens mostly as a process of evolution in the world of technical discussion forums and projects.

Existing open-source Web frameworks are tangible results of this process. These frameworks have undergone some peer review, since they are used widely and their designs evolve in correspondence with feedback from their users. Existing framework implementations thus form an important repository of knowledge in this field; hence the overview given of these in the present article.

4. REQUIREMENTS

In order to give a better indication of the role of Web frameworks, a list of functional requirements of Web frameworks is proposed in this section.

No comprehensive list of functional requirements of Web frameworks has been found in the literature. By excluding topics deemed outside the scope of Web-based UI and generalizing slightly, the following list of topics has been distilled from some concerns and functionalities mentioned in the literature.

—*Presentation*. Presentation and the specification thereof are mentioned by Helman and Fertalj [2003] and Copeland et al. [2000]. Presentation is described in Section 4.1.

—*Form Handling*. Topics related to generating forms and dealing with input received from forms are mentioned in Helman and Fertalj [2003] and Copeland et al. [2000]. These are classified as "form handling" functionality, described in Section 4.2.

—*Validation*. Validation-related concerns are mentioned by Rode et al. [2004] and Westkämper [2004]. These are incorporated in Section 4.3.

—*Navigation*. Navigation-related concerns are mentioned by Helman and Fertalj [2003], Copeland et al. [2000], and Westkämper [2004]. In the list of requirements presented here, the closely related low-level concept colloquially termed "page flow" is included in Section 4.4.

—*Session Management*. Session management is mentioned in Rode et al. [2004]. It is incorporated in the presented list under the heading of "session state" in Section 4.5.

—*Security*. Security-related topics such as authentication and authorization are mentioned in Helman and Fertalj [2003], Rode et al. [2004], and Westkämper [2004]. These are incorporated in the list presented here under the heading of authentication in Section 4.6.

—*Back-End Integration*. The issue of integrating with back-end systems is only mentioned by Copeland et al. [2000]. In this article, back-end integration is included and explained in Section 4.7.

The list of requirements proposed in the present article is further extended by insights gleaned from a survey conducted on 80 Web frameworks. Specifically, the foregoing list is extended by functionality relating to event handling (Section 4.8), concurrency (Section 4.9), resource usage (Section 4.10), and other miscellaneous requirements (Section 4.11).

4.1. Presentation

Presentation is perhaps the simplest and best-understood functional requirement that a Web framework should provide. It is the first requirement recognized; early Web frameworks focused exclusively on presentation. (Note that presentation is very closely related to the view concerns of the MVC design pattern.)

"Presentation" entails everything a Web application needs to do in order to render its user interface in a client's browser, using hypertext markup language (HTML) or a similar markup language. Typically, in order to facilitate reuse, each page is not merely seen as a Web page; it is viewed rather as a window in a graphical user interface (GUI), composed of different UI components. The difference between the UI components on a Web page and those used in a GUI is that the latter usually are available as reusable programming language components with behavior linked to their presentation; the former are mere low-level representations. The low-level representations on a Web page not only lack behavior, but cannot be made into more complicated compositions for reuse.

4.2. Form Handling

User input on the Web takes place through HTML forms that are submitted to the Web server by a browser. Forms are submitted to a particular uniform resource locator (URL) on a Web server, as a hypertext transfer protocol (HTTP) request. "Form handling" is a term often used to describe various tools which a programmer can use to deal with forms, and especially to deal with the user input that has been received as part of submitted forms.

For example: When user input is received via an HTTP request, it comes in the form of named text strings. These can be marshalled to more useful typed programming language objects. Errors can happen during this process, which need to be reported to the end-user.

Forms can also be submitted as a result of different buttons being clicked on the form. As part of form handling, a program needs to determine which button caused the submission and needs to take appropriate action, depending on that information.

4.3. Validation

Validation is also related to user input and is often seen as part of form handling. Validation usually refers to checking user input against some constraints. (For example, a number may be required to be within a certain range, or a string representing an email address can be required to match a regular expression.) However, validation might also be dependent on more sophisticated domain knowledge which might have to be checked in a back-end system (where domain-specific knowledge typically resides) on another tier in a multitier architecture.

Provision needs to be made in a Web framework for a programmer to easily specify what validations need to be carried out, how validation errors will be reported to the user (so the user can correct them), and what influence such errors will have on the dynamic behavior of the presented UI.

4.4. Page Flow

Page flow is a colloquial term used by Web framework designers to describe the relationship between the pages of a Web site. Page flow is a description of the possible ways in which a user can traverse the different logical locations (Web pages) in a Web-based UI. It is to locations (or pages) in a Web UI what control flow is to statements in a programming language.

Indeed, from the perspective of this study, the low-level page flow tools (buttons and links) can be likened to the goto statement in programming languages: The GUI of a page is really its address. An HTML link on another page to this address is to page flow what a goto statement is to control flow. In fact, there is a sense in which the well-known problems associated with goto statements [Dijkstra 1968] are exacerbated in the case of page flow, since the specification of dynamic behavior has to account for the fact that link specification may be dispersed between and embedded within several different Web pages.

A higher-level construct seems to be needed to specify the page flow of a Web application. Only very recently have a few attempts at such specifications been made by framework implementors.

Closely related to page flow is the concept of the navigational model of a Web application, which is widely discussed in the literature (see, e.g., Winckler and Palanque [2003]).

The implementation of page flow has implications for the client browser: Typically, a browser is based on the premise that a GUI denotes such a location on a Web site. The browser adds functionality, such as the ability to bookmark a location and to go backwards and forwards along the path traversed by the user, based on the identification of locations with Uniform Resource Locator (URL).

4.5. Session State

HTTP is a stateless protocol. It provides little support for relating a particular request to others in a lengthy conversation. Web applications, though, need some way of relating all the requests from a particular user during a user session in order to be able to store information between requests for a single user session. The information stored in the scope of a user session is referred to as "session state."

Session state is used, for example, when a user has provided input for several input items on a form, has submitted the form, and then expects that her input will subsequently be shown on the form, even in the event of a validation error requesting her to change some of the information. Such an interaction may span several requests to the Web server. The information supplied at the beginning in the submission of the form is needed to render, for example, a half-completed form later on (possibly with an error message added).

The following scenario illustrates the example.

—The browser issues a GET request to the server, which returns a 200 result, and an HTML file containing a form.

—The user completes the form fields and clicks on a submit button provided as part of the form.

—The browser issues a POST request to the server, with the information supplied by the user.

—The server validates the data supplied and detects that the data is invalid. In order to grant the user the opportunity to correct the input, the server responds with a 304 result, redirecting the browser to do a new GET request.[6]

—The browser issues the new GET request, upon reception which the server needs to respond with an HTML page containing a form very similar to the original one with the input as supplied by the client prepopulated.

4.6. Authentication

Authentication is related to session state.[7] Web applications often need users to authenticate themselves, for example, by means of a user name and password. Once

[6]Although it is possible at this point to return a suitable form, it is considered bad practice to return an HTML page in response to a POST request [Fielding et al. 1999; Jouravlev 2004]. Each request returned to a client may be bookmarked or revisited by clicking on the back button of a browser. However, POST requests are assumed by browsers to have side-effects. Moreover, this assumption causes most browsers to show a confusing warning message, should a POSTed GUI be revisited by clicking on the back button (for example). Hence the extra redirection to a GET request.

[7]Sometimes, authentication is implemented using the same mechanisms as session state, for example, when using Cookies.

authenticated, the Web application can modify its UI based on the identity of the user.[8] Authentication, however, is something that most Web application designers prefer to happen only once during a user session, to be stored by the application for the duration of the session.

Of course, authentication needs to be implemented in a secure way, and in this case neither the client nor the transport mechanism can necessarily be trusted.

The HTTP protocol provides more than one authentication mechanism, which can be further augmented with the use of HTTP over secure socket layer (SSL). However, the problem is exacerbated by the fact that many older Web browsers implement the standard incorrectly [Radke 1999b], leading to complexities for a Web server that has to cater to all browsers. For example, some browsers that do not support digest authentication will, when challenged using digest authentication, reply using the (insecure) basic authentication scheme instead of failing [Radke 1999a].

4.7. Back-End Integration

Usually, Web applications are built using a multitier architecture, with their presentation tier being Web-based. Web frameworks are concerned with such Web-based presentation tiers.

In a degenerate case, i.e., when the multitiers are collapsed into one) this Web tier would need some way of accessing a local database. More often, the Web tier has to access a remote database or an application server. Such “back-end systems” often need to be accessed within the boundaries of a single database transaction or security realm. For example, a user should not have to log in once for every back-end system accessed; once a user is authenticated, those credentials should be propagated to the security mechanisms of all the back-end systems involved.

A Web framework should provide abstractions for a programmer that automatically deal with these low-level technical concerns in a sensible way. Apart from the fact that Web programmers do not want to have to think about such low-level, complex, technical issues, many UI programmers may not be well versed in such matters.

4.8. Event Handling

An incoming request from a browser signifies that the user has triggered an event of some kind, either by submitting a form or by having clicked on a link. Such a request can be interpreted in many ways. For example, it can be seen as notification that a UI event has occurred, or a batch of UI events can be derived from it. The design of a Web framework determines what constitutes an event, how events are triggered, how events relate to HTTP requests, and how custom program code will be invoked in response to events.

In some cases, for example, a request is simply mapped to a method call. Other frameworks have more complicated programming models for dealing with events.

4.9. Concurrency

Web-based applications are, by their very nature, accessed concurrently by several users. Since servicing a single request may take a long time, a way is needed to service requests in parallel. This can be done by maintaining a pool of processes, a pool of threads in a single process, or even a pool of machines with pools of processes on them.

[8]Many Web applications also depend on the user’s authentication level to control the way in which particular resources are displayed to a user.

Some servers also have an asynchronous model: They kick off processing (typically on a back-end system) for each request as it is received, without blocking to wait for the processing (which is often input/output (IO)-intensive) to complete. The system can then alternatively poll for results from such processing (and quickly send a response) or start more processing for incoming requests (e.g., see Rushing [2005]).

The particular concurrency model chosen has quite an impact on how some other requirements can be implemented. A naïve implementation of session state, for example, could keep some information for a user in memory. However, generally, memory cannot be shared between processes or machines posing a problem in circumstances where subsequent requests from the same user could be routed to machine, or another process (as often done by load-balancing software).

When using multithreaded approaches, care needs to be taken that the programs and libraries involved are all thread-safe.

4.10. Resource Usage

A common thread running through many implementation discussions for Web frameworks has to do with the economical use of resources. Several scarce or expensive resources are used by Web applications or Web frameworks. Frameworks have to take care to use resources such as the following in ways that remain performance-efficient when scale-up occurs.

—*File Descriptors*. File descriptors are in limited supply on a machine, and Web serving software uses them for writing to log files, in writing to temporary files for persisting session state, and the like.

—*Sockets*. Sockets (or transmission control protocol (TCP) ports) are also limited on each machine, thus limiting the number of network connections that can be maintained at one time.

—*Connections*. Connections to databases or other back-end systems are often only allowed in limited number, and creating a connection typically involves a performance overhead too large to incur on every HTTP request. Thus, connections to back-end systems might not be created for each request. Instead, a pool of connections is usually maintained so that the total number of connections is centrally managed, and an already created connection can quickly be allocated to the servicing of one request.

—*Memory*. Memory cannot remain allocated to each particular user for the duration of her session, given the number of concurrent users that have to be serviced.

—*Processes*. Processes (and, to a lesser degree, threads) involve overhead to create and destroy. Thus, they also are typically created in advance in a pool from which existing processes or threads can be allocated to requests.

4.11. Miscellaneous

Miscellaneous requirements may be added to the important set explained thus far. For example, Web applications often need special support to enable them to be presented in different languages, to work with different locales, or to use different character encodings for Web pages [Spolsky 2003; Dürst 2005; Westkämper 2004].

5. STRATEGIES FOR VIEW CONCERNS

A taxonomy of strategies employed for handling the view concerns of Web frameworks is proposed in Figure 1. The discussion of the taxonomy is structured as an ordered

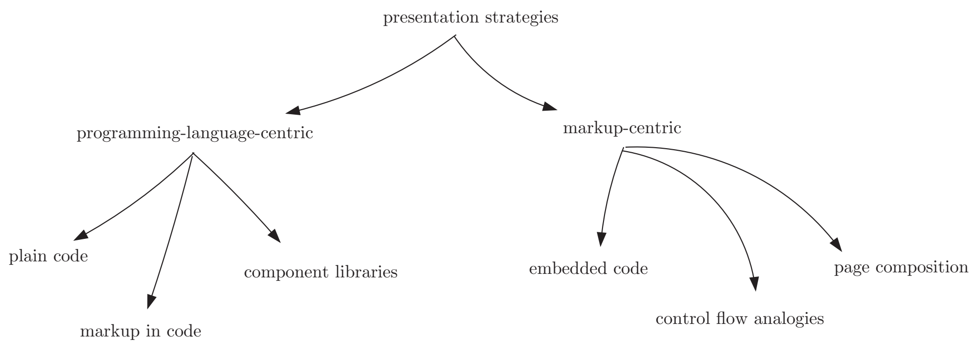

Fig. 1. A taxonomy of strategies for view concerns.

traversal of the nodes in the taxonomy tree, with a description of each node provided (roughly in increasing order of complexity or sophistication).

The basic problem addressed by these strategies is how Web-serving software can generate a response that a Web browser can display as part of the UI of an application. In practice today, this means generating a textual document in a markup language such as HTML, extended hypertext markup language (XHTML), or extensible markup language (XML).[9] Note that we often loosely talk of HTML for brevity where another markup language can be substituted that browsers are able to display.

5.1. Programming-Language-Centric Approaches

Programming-language-centric approaches to generating markup all consist of specifying a program in a general-purpose programming language which generates the markup file.

Sections 5.2, 5.3, and 5.4 describe further specializations of this category.

5.2. Plain Code

The "plain code" strategy is the very simplest of all strategies for generating markup. Code is invoked that either writes the markup as text to a file, or just returns it in a string. This is the strategy used by Java Servlets^{TM} [Coward and Yoshida 2003].

The Python [Python Software Foundation 2005] programming language function, provided next, illustrates the point.

```
def foo():
  print '<body>'
  for word in ['these', 'are', 'words']:
      print '<p>Word: %s</p>' % word

  print '</body>'
```

Note that the "print" statement in Python writes to standard output.[10] The function could also have written to a different file, or have returned a string value.

[9]The interested reader is referred to Raggett et al. [1999], Yergeau et al. [2004], and W3C HTML Working Group [2002] in connection with the details of these standards.
[10]Standard output is often used by scripts for returning output to a browser.

5.3. Markup in Code

Some systems augment a programming language with special semantics that enable one to embed markup in the programming language. This can best be explained by an example.

Quixote is an extension to the Python programming language [Corp. for National Research Initiatives 2005]. An example of a *Quixote* function can be seen in the following code excerpt (which returns the body of an HTML document).

```
def foo [plain] ():
    '<body>'
    for word in ['these', 'are', 'words']:
        '<p>Word: %s</p>' % word

    '</body>'
```

A *Quixote* function always returns a string (even though no return value is explicitly specified). This return value is automatically computed as follows: At first the string is empty, but as each statement in the function is executed, its individual return value is converted to a string and appended to the return value of the function. *Quixote* is helped here by the fact that literal values in Python can be used as statements (with no side-effects, but themselves as return value). Return values of `None` are ignored.[11]

Hence, text can be embedded in a programming language with the latter used to govern the final text that will be returned.

This example code thus yields the same HTML as the previous example.

5.4. Component Libraries

Some frameworks include a library of UI programming language classes similar to GUI framework libraries. With these, a Web page can be composed using the programming language objects. Such a composition can then be rendered as markup.

As an example, here is a Java method returning a window (or page) that contains the "hello world" string as a label, using the *Echo* framework [NextApp, Inc. 2005]. (The example is a stripped-down adaptation of their own "hello world" example.)

```
public Window init() {

    Window window = new Window();

    ContentPane content = new ContentPane();
    window.setContent(content);

    Label label = new Label("Hello, World!");
    content.add(label);

    return window;
}
```

5.5. Markup-Centric Approaches

Markup-centric approaches represent a broad category of strategies comprising a syntax with which a template for an HTML document can be specified. Such a template can

[11]In Python, a return value of `None` denotes a void return value.

then be rendered (or executed) to yield different actual markup documents, depending on the context and parameters with which the template was executed.

This category is further refined and exemplified by strategies in Sections 5.6, 5.7, and 5.8.

5.6. Embedded Code

As far as templates go, a template with embedded code is probably the simplest model to understand. A special syntax is introduced with which programming language code can be embedded in an otherwise normal HTML document. Early JavaServer Pages™ (JSP) (from Sun MicroSystems (Sun)) is an example, as is the open-source *PHP* and active server pages (ASP) (from Microsoft Corporation) [Roth and Pelegrí-Lopart 2003; PHP Group 2005].[12]

```
1  <body>
2    <% String[] wordList = "these", "are", "words";
3       for (int i = 0; i < wordList.length(); i += 1) {
4    %>
5    <p>Word: <% wordList[i] %></p>
6    <% } %>
7  </body>
```

Here, the brackets "<%" and "%>" are used to delimit Java code. Note that logically, line 5 is in the body of the for-loop started in line 3. The semantics are that a copy of line 5 would be included in the output for each iteration of the for loop, yielding the following.

```
<body>
  <p>Word: these</p>
  <p>Word: are</p>
  <p>Word: words</p>
</body>
```

Some of these template languages employ the same semantics but use their own, more lightweight, syntax. Here is an example of *Cheetah* [Rudd 2005].

```
</body>
#for $word in $wordList
  <p>Word: $word</p>
#end for
</body>
```

The foregoing examples are typical of what is found in current server-centric Web frameworks. However, another, much older example of exists, and looks a bit different: server-side includes (SSI) (also known as server-parsed HTML).[13] With SSI, a programmer can put tokens in an HTML file, which would be replaced by the contents of other files, or by the output of a program, before the resulting HTML is sent to a client browser [McCool et al. 1999]. Some modern Web servers (like Apache)

[12]Note that specifications from the J2EE family are cited in this overview as if they were frameworks (notably, JSP and JSF). Many implementations of these specifications are available, but citing the specifications is preferred because of the prominent role of these specifications and because the strategies used by individual implementations merely implement the specifications.

[13]We are grateful to an anonymous referee for drawing our attention to the fact that SSI was developed in 1993 by Henrich [McCool et al. 1999] and used extensively in *ColdFusion*.

allow one to embed quite complex statements controlling such “inclusion logic” in HTML comments.

```
<!--#if expr="$afile" -->
<!--#include virtual="includedfile.txt" -->
<!--#else -->
<!--#exec cgi="programToRun.cgi" -->
<!--#endif -->
```

5.7. Control Flow Analogies

Some markup-centric approaches are attempts to introduce the same semantics expressed using the embedded code approaches (Section 5.6), but in a way (from the point-of-view of the target markup language) less intrusive than embedding the code.

They extend the markup language’s own vocabulary with additional control flow statements. The semantics of these statements is the same as those used to control flow in conventional programming languages: If-statements control the conditional inclusion of parts of the template, and looping statements allow one to include a chunk of the template several times.

The extension of the markup language is usually done either by introducing added tags to the markup language, or by adding attributes to existing tags. The motivation behind this seemingly cumbersome notation is that the source code of the template would then be valid according to its markup language (usually some form of XHTML, such as XHTML), and thus editable in what-you-see-is-what you get (WYSIWYG) editors.[14] JSP restricted to its JSP standard tag library^{TM} (JSTL) is an example [Pierre Delisle 2002; Roth and Pelegrí-Lopart 2003], and is exemplified next.

```
</body>
<c:foreach var="item" items="wordList">
 <p>Word: $item</p>
</c:foreach>
</body>
```

(The ASP.NET framework from Microsoft Corporation is another well-known example in this category when using its “Web Forms” [Microsoft Corp. 2005].)

5.8. Page Composition

Template languages based on control flow analogies alone (Section 5.7) suffer from the weakness that the low-level control flow constructs used often obscure the design of a complex page. Complex pages are most often designed as being composed from several UI components, analogous to widgets in GUIs. All the lower-level detail present in the control flow constructs obscures a programmer’s view of the logical components on a page.

Current page composition approaches also extend the vocabulary of a markup language by adding to its tags or attributes. However, the aim is not (only) to introduce constructs on a control-flow level, but also to allow a programmer to compose a page from a standard (often extensible) set of UI components. These languages also retain the

[14]It should be noted that WYSIWYG editors will not display the page resulting from executing the template being edited. They can only show the logical structure of the source code of the template itself (which also happens to be valid XHTML). To get around this problem, some template languages are designed so that their source looks like an example of what may be rendered, given some execution of the template. This goal can of course not be wholly attained, since a template, by nature, does not evaluate to a static document.

more basic control-flow-level constructs, but their use is much rarer and the resulting template more readable.

An example is again JSP, but this time using the newer tag libraries provided by JSF [Roth and Pelegrí-Lopart 2003; Pierre Delisle 2002].

```
 1 <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
 2 <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
 3
 4 <html>
 5   <head><title>Example</title></head>
 6   </body>
 7     <f:view>
 8       <h:form id='exampleForm'>
 9         <h:inputText
10                id='message'
11                value='#MessageBean.message'/>
12          <h:commandButton
13                id='changeMessage'
14                action='success'
15                value='Change'>
16        </h:form>
17      </f:view>
18    </body>
19  </html>
```

Lines 1 and 2 merely state that certain libraries of XHTML tags (prefixed with “h” and “f”, respectively) will be used further on in the file. The “f:view” tag contains the specification of a whole window, composed of a form (“h.form”) which in turn contains a text input box (“h:inputText”) and a button (“h:commandButton”).

Although the example uses very simple widget components, more advanced components can be used that encapsulate logic usually coded with conditional and looping statements.

5.9. Discussion on the Taxonomy of View Concerns

At first glance, some frameworks seem to use combinations of the strategies presented as distinct in the taxonomy. On closer inspection, though, not all elements of such a combination are used for expressing view concerns.

In *Wicket*, for example [Wicket 2005], one can express how a page will be rendered in a componentized way, using a typical page composition approach (Section 5.8). Such a specification is augmented by the programmer writing an additional component-library-oriented (Section 5.4) specification of the same page, containing a corresponding set of components which mirror the original specification exactly. The two resulting specifications thus mirror each other, but use different specification techniques.

The templates in the page-composition-oriented specification are used to generate HTML pages. Each component in the template is associated with its programming language counterpart in the additional component library specification of the same page. The component library specification is used on the server: If an event is triggered from the HTML (which was rendered by page the composition-oriented specification), the server would relay the event to that server-side component (as per the component library specification) associated with the page composition component responsible for rendering the HTML which allowed the event to be triggered. (A button, for example, allows an event to be triggered by rendering an HTML button on which a user can click.)

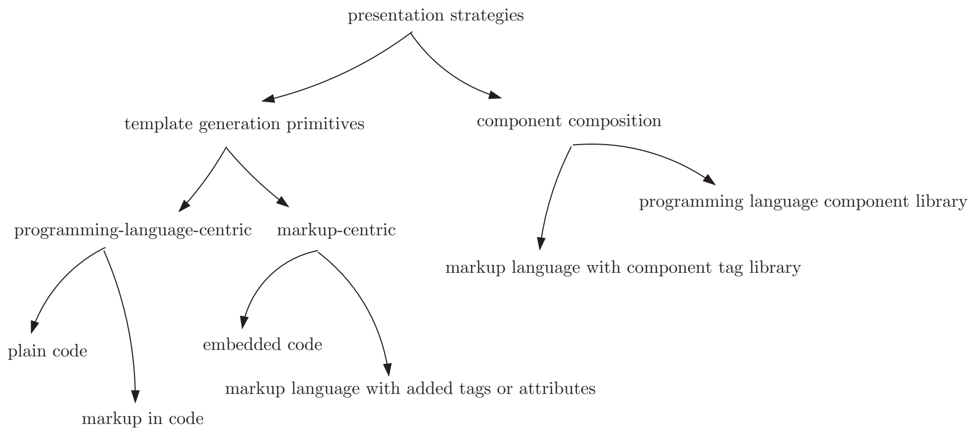
presentation strategies
template generation primitives
component composition
programming-language-centric
markup-centric
programming language component library
markup language with component tag library
plain code
markup in code
embedded code
markup language with added tags or attributes

Fig. 2. An alternative arrangement.

It may be tempting to add another top-level category to the taxonomy called "hybrid models" for such solutions. However, when classifying a framework, care should be taken to distinguish which concerns are addressed by a strategy used. In the current example, the page composition specification is used to express view concerns, and the additional component library specification is used solely to be able to *handle events* in a structured way. The latter is a controller issue, and hence not applicable to this particular taxonomy; thus the framework can be classified according to its page composition specification on this taxonomy.

That said, some frameworks use such a component library mirror for more than merely handling events: Data received from components can be type checked and converted to native data types, and validation can also be specified. The result is that components are not only rendered, but also induce server-side state and server-side behavior. (JSF is an example of such a framework, see McClanahan et al. [2004]).

All of the strategies in the taxonomy embody some combination of the two main choices next described.

—*Intention*. This concerns the choice of whether the specification uses lower-level control-flow-like constructs with which to specify templates, or whether it takes the higher-level approach of composing a page from several (often nested) UI components.

—*Language*. This is the choice referring to whether the language used to specify the view is based on programming language code, or the target markup language.

The foregoing taxonomy lends the second choice more importance, for historical reasons, but both dimensions are in reality of equal importance. Without further explanation, and mostly as an interesting aside, the alternative arrangement is shown in Figure 2.

Note how this arrangement gives more weight to whether an approach is geared towards composing a page from components. From this point-of-view, some programming-language-centric approaches and some markup-centric approaches do not seem as different as one may think at first glance. Regarding the idea of component omposition, this also highly suggests that it may be worthwhile to explore other possibilities along these lines. Most markup-centric approaches with tag libraries still allow an older form of control flow analogies as well. Hence, it may argued that it seems a bit dishonest to classify markup-centric approaches with tag libraries as being *so different* from other markup-centric approaches.

Finally, it is interesting to note that taxonomies can often suggest or call to mind previously unknown strategies. For example, none of the 80 frameworks studied relied on a *template language* with the same basic goal of the page composition variants (Section 5.8), but with a syntax external to the host markup language (such as the example of *Cheetah* mentioned in Section 5.6). The absence of such a category in the aforesaid taxonomy could perhaps be the trigger for developing just such a language.

6. STRATEGIES FOR CONTROL CONCERNS

Controller concerns deal with routing events between the view and the model, keeping these two in sync while taking care that the model and its view stay decoupled. Controller concerns are thus core to the specification of the dynamic behavior of a UI.

Within the client-server environment in which a Web-based UI is executed, it is assumed that the display in a Web browser will only be updated by the browser polling the model via the Web server. Hence, controller concerns typically boil down to deciding how to relay the events generated in the browser to the model, and how to generate a Web page (view) in response. Page-flow-centric approaches (Section 6.11) expand the scope of controller concerns in an important way to include the relationship between pages in a UI.

The abstract perspective this article takes of Web frameworks is that they are analogous to interpreters that execute a specification of a UI in the distributed environment, which is comprised of a combination of Web server and Web browsers. Controller concerns and the strategies used by frameworks for dealing with these can be seen as approaches towards implementing the basic execution model of these “interpreters”, how controller concerns are dealt with thus dictates the basics of how a specification can be executed on such a “distributed virtual machine.” Thus, in the perspective of this article, controller concerns are prominent.

Figure 3 shows a proposed taxonomy of the strategies for controller concerns as used by the 80 Web frameworks that were studied. The rest of this section is a discussion of the taxonomy, structured as a particular traversal of each node.

At the top level, the taxonomy distinguishes three broad categories: static files (Section 6.1), dynamic content (Section 6.2), and page-flow-centric (Section 6.11).

6.1. Static Files

A Web server serving static files is the most elementary strategy for addressing control concerns; namely to have no dynamic behavior at all. The use of static files is mentioned here for completeness. Many Web sites are built like this even though the concept of document in the Web standards is in no way limited to *static* documents.

A Web server has access to a file system hierarchy containing HTML (and other) files. A GUI directly maps to a path in this hierarchy, ultimately leading to a file. Upon an HTTP request, the server merely sends the file denoted by the request GUI back to the client browser.

6.2. Dynamic Content

Most current Web frameworks fall into this category. Returning a document in response to an HTTP request is still the focal point. Here, however, the document sent back is generated on-the-fly, and various ways are used to invoke other code in the process.

The category of dynamic content is divided into approaches that mix MVC concerns (Section 6.3), and those that do not (Section 6.7).

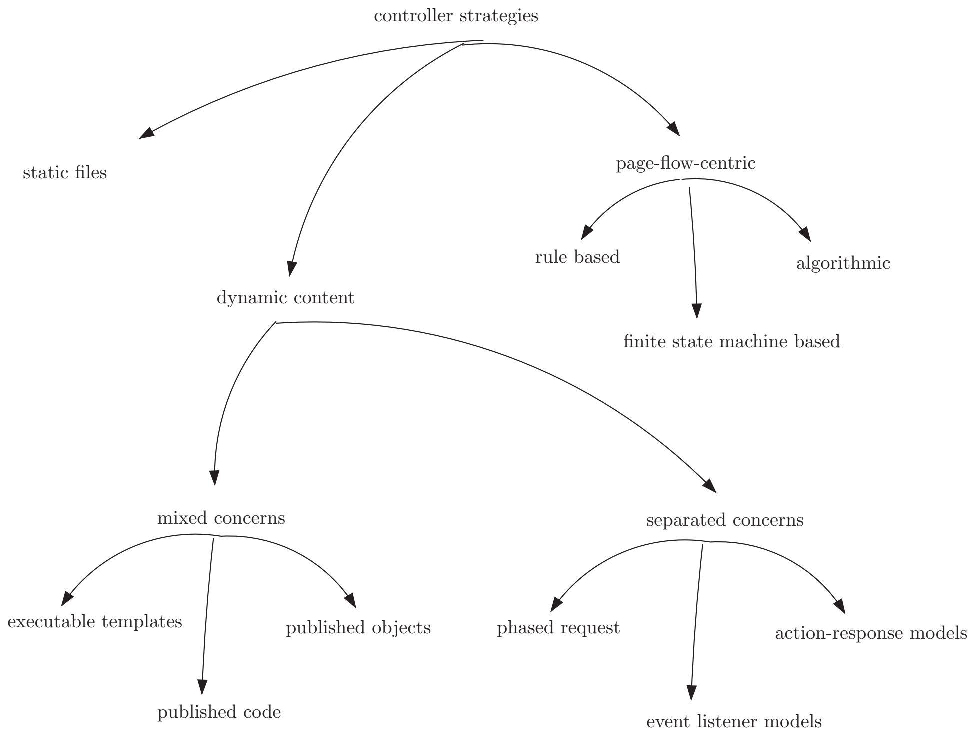

Fig. 3. A taxonomy of strategies for control concerns.

6.3. Mixed Concerns

Strategies in this category do not provide formal support for a programmer to separate logic that has to do with the generation of a dynamic page from logical actions and other code that have to be executed as a result of user-triggered events.

More detailed examples of strategies with mixed concerns are presented in Sections 6.4, 6.5, and 6.6.

6.4. Executable Templates

Languages such as *PHP* and the early versions of JSP are often used in this way [Roth and Pelegrí-Lopart 2003; PHP Group 2005]. Instead of having static files, a collection of templates can be organized in a file system (or similarly structured schema). Such templates are executable (some interpreted, some compiled) and yield (possibly different) documents as a result of their execution. Upon an HTTP request, the template file denoted by the URL is executed and the resulting document is sent back to the browser in response.

The programmer should specify any other code using facilities provided by the particular template language used, mingled with the view logic for which such templates are geared.

6.5. Published Code

Published code is very closely related to templates that are executed. Here, the URL is mapped to a script or function written in a programming language. Upon a request, the corresponding code is invoked. Such code will then either write text to be returned to a file or return the text as a return value which the Web server then can send back to the client.

Optionally, the script or function can be declared with a signature specifying the arguments with which it should be called. These named arguments would then be extracted automatically from values submitted with the request, and would be passed to the invoked code as actual parameters to match the function's formal parameters. (There are several variations on this theme of how the GUI, and possibly a form, can be mapped to named and/or positional arguments declared by a function or similar construct. See the related, but object-oriented, example in Section 6.6.)

6.6. Published Objects

A slightly more structured approach is for a server to maintain a hierarchy of objects or classes (in the object orientation (OO) sense) instead of a file system hierarchy. The GUI of an incoming request is then seen as a path along this object hierarchy, leading to a particular object. Objects usually have an inheritance hierarchy, but they could also be said to have a containment hierarchy (via attributes or other methods). Either of these hierarchies can be used for mapping a request to an object.[15]

Depending on the system, a special method of the object (such as render()) can then be called by the request.

The GUI can also be taken to denote a method of an object in the hierarchy, so that each request results in a method call (as with published code approaches, addressed in Section 6.5). Again, method signatures can aid in specifying and type-casting request parameters for use in the body of the method.

In the following example (adapted from an example in the *CherryPy* tutorial), a POST request to the GUI '/doLogin' would execute the following code.

```
class Root:
    def doLogin(self, username=None, password=None):
       # values for the keys 'username' and 'password' would
       # be extracted from the request and assigned to
       # the variables username and password in the
       # body of this method
```

An object may have attributes other than methods, and, although these are generally private and thus not directly addressable via a GUI, a method so invoked can access these private attributes (some of which could be templates).

By using such an organized collection of methods and templates (and in fact many different kinds of objects), a programmer has access to many tools that are available for structuring the way in which a request is handled and for constructing a reply.

Note that objects have state (per user or application-wide). Thus, having to keep them on a server implies that a server needs to store information pertaining to each connected client (see the discussion of session state in Section 4.5).

6.7. Separated Concerns

In contrast to these mostly unstructured approaches (Section 6.3), attempts have been made to provide for a more structured way to specify view and other code separately.

[15]For example: *Zope* and *CherryPy* use containment hierarchies [Zope Corp. 2003; CherryPy 2005b].

Typically, "other code" is vaguely defined by these approaches as being code that has to do with "logic," with "business logic," or with the model or controller in MVC.

This class in the taxonomy represents such Web frameworks: ones that focus primarily on the problem of generating a response to a single request while keeping concerns separated.

Three subcategories are discerned in this category of strategies: phased request approaches (Section 6.8), action-response approaches (Section 6.9), and event listener approaches (Section 6.10).

6.8. Phased Request

Some approaches have a well-defined "request processing cycle" during which specific pieces of code can be called at certain stages of handling the request (e.g., on entering the page, or on displaying it). *Albatross* is an example [Object Craft P/L 2005].

6.9. Action-Response Models

Action-response models separately define view pages (often called views), controller-specific code, and other code. A URL is mapped to specific controller code (actions in *Struts*, for example [Apache Software Foundation 2005b]). Upon a request, the invoked controller code in turn invokes model code and then decides, based on the outcome of such an invocation, which view to render in response to the request. Instead of immediately returning a response, such controller code can also instruct the browser to fetch a view from a different URL.

In the Java world, the basic models of "model-2 architectures" [Seshadri 1999; Draheim and Weber 2005; Westkämper 2004] roughly operate in this way. A Model 2 architecture includes Java programming language code (some of which is used to handle controller concerns explicitly), and separate view code (typically templates, such as JSP) for rendering HTML pages.

6.10. Event Listener Models

Event listener models are abundant in the Java world [Wicket 2005; McClanahan et al. 2004; BarracudaMVC 2006; NextApp, Inc. 2005]. These provide "listener classes": classes which a programmer can subclass (or interfaces that can be implemented) to provide custom implementations of special call-back methods. An instance of such a class (or an instance of an implementor of such an interface) can register its interest in events of a particular kind with a UI component, such as a button or selection box. If the UI component detects the occurrence of that kind of event, it will call the appropriate call-back method of the relevant listener instance, and so invoke the custom supplied event handling code.

In Web-based UIs, this means that one needs a programming-language-based model of the components of a page on the server (such as used with component libraries; see Section 5.4). These should also be maintained in session scope, so that they can retain their state between different requests from a particular browser.

Incoming requests are then handled in one of two ways, detailed in the following.

—The request is seen as the occurrence of a single event (usually a button having been clicked, or JavaScript being triggered by the browser). The event is related to the programming-language-based component to which it logically belongs, and that component is notified that the event took place. The component can then call any listeners for that event in response.

—A whole batch of UI events are inferred from a single request. An example best illustrates the point. Assume a page with two components: a button and a selection box. Upon its initial rendering, the components representing this logical layout can store their initial values. If the user selects a different value in the selection box, the browser does not generate any events to the server. However, when the user eventually clicks on the button that is also provided, it results in the whole form being submitted. Upon receiving this request, the values held by each server-side programming language component can be compared to the corresponding values in the submitted form, and multiple events can be derived from the single request. In this example, the selection box can infer that a "selection changed" event took place, since it can detect that the value submitted as part of the form is different from the stored value previously selected. The button can infer a "button clicked" event. And both events are handled separately by their respective event listeners, but in one batch as a result of the submitted form.

6.11. Page-Flow-Centric

Approaches in the category of separated concerns (Section 6.7) recognize the importance of keeping the specification of different MVC concerns separate. However, they still focus on the problem of how a particular response is generated to an incoming HTTP request for a single page, a narrow focus for an architectural framework which aims to specify and run a *complete* UI for a Web application. In most of the projects surveyed, a programmer cannot take a wider view and clearly specify the relationship *between different pages* in a Web site: namely, the possible paths a user could use to traverse them.

Pages are linked to each other by HTML links or buttons (or a similarly scoped mechanism in server-side code) which are embedded into the specification of view of each individual dynamically generated page. The resulting structure between pages is difficult to visualize and often quite chaotic (see the discussion of page flow in Section 4.4).

Page-flow-centric approaches recognise that controller concerns include the control of page flow. They have explicit notations for composing a UI from different pages, specifying how these can be traversed by a user.

Page-flow-centric approaches are significant because they allow a programmer to specify a UI in its entirety, as opposed to specifying a UI as a loose collection of individual pages (and other components) with interrelationships that are not made explicit.

Note that although these approaches all specify page flow explicitly, they tend to deal with a page abstractly as "a logical location in the UI." Generally speaking, they do not map this conception of logical location strictly to the URL displayed by the browser.

There are three kinds of page-flow-centric strategies: rule-based, finite-state-machine-based, and algorithmic approaches. These are discussed in Sections 6.12, 6.13, and 6.14, respectively.

6.12. Rule-Based

Rule-based approaches (e.g., JSF; see McClanahan et al. [2004]) have a central set of rules that state, for each location in the UI, all the paths to other locations.

In JSF, such locations are called views. Each view can be visited by a Web browser, and upon such a visit, application code is invoked. Such application code should return an outcome; that is, a string denoting the result of the execution. The rules can then state which outcome leads to which successive view.

This sounds very similar to action-response models (Section 6.9). The difference is that the rules here span several requests: When a request is received, the server tracks which view the browser is at, and computes the next view based upon the combination of the current view and the outcome of executing some business code. Maintaining the knowledge of which view is currently active is done for the programmer.

6.13. Finite-State-Machine (FSM)-Based

Some approaches model page flow as some form of FSM. The semantics and types of the FSM vary, but generally states represent either an action to be performed, a decision that has to be made on the server, or a view that has to be presented via the browser. Transitions specify the valid paths in the UI between states (the transition to be followed can be decided on the outcome of a state,[16] or a named event that occurred).

An example is *SpringWebFlow*, which allows specification of a site as a flow: a number of states that are interconnected with transitions [Ervacon 2004]. *SpringWebFlow* defines five different kinds of state (listed next). One state in the flow is designated as the starting point for the flow. Upon the application entering any of the states, some kind of action occurs, mostly yielding some sort of outcome for that state. This outcome (or some other method) is then used to determine what state to transition to next.

—*Action States.* An action state is a state that, when visited, executes some application code. This code should return a string which is interpreted as an outcome of the execution of the action. This outcome is used to determine which transition should be followed to the next state in the flow.

—*View States.* A view state, when visited initially, merely renders a page to the client browser, causing the UI to pause in the view state, waiting for user input. Button clicks on the rendered page result in a form submittal. During such a form submission, the name of the actual button clicked is used as the outcome to use in determining which transition to take next.

—*Decision States.* A decision state works slightly differently in that it is not coupled with the concept of an outcome. It merely specifies which state should be transitioned to next, depending on certain conditions.

—*Subflow States.* A subflow state is a way of nesting another flow inside the current one: Transitioning to a subflow state means following the flow defined by it, starting from its start state (each flow specifies which of its states is a start state). Subflow states facilitate reuse of flows in a number of other "calling" flows.

—*End States.* An end state itself denotes the outcome for a flow as a whole. Should a transition be followed to an end state, the flow is said to terminate with the end state's identifier as a outcome. When used as a subflow state, this would be the outcome of that subflow state, used by its parent flow to decide which transition to take next.

Another, different, example in this category is *Expresso* [Jcorporate Ltd. 2005].

Finally, a Web framework relying on a specification language that is based on statecharts is proposed in Vosloo [2005].[17]

6.14. Algorithmic Approaches

The same sort of specification expressed by an FSM or a set of rules can be expressed in an algorithm using a programming language. Tasks in the Ada language may be regarded as an analogy of what such algorithms will look like. The concept is best

[16]States representing code executed on the server may be said to have an outcome. When the code finishes executing, it returns a value which is interpreted as an outcome. This is also the time at which the transition for that outcome is triggered.

[17]This work has since been launched as an open-source venture (see `http://www.reahl.org`).

illustrated by an example (using the *Cocoon* framework [Apache Software Foundation 2005b], in the JavaScript programming language).

```
 1 function example()
 2 {
 3   var result;
 4
 5   cocoon.sendPageAndWait("firstPage.html");
 6   result = cocoon.request.get("someFormValue");
 7
 8   if (result == "a")
 9     cocoon.sendPage("aPageForA.html")
10    else
11     cocoon.sendPage("aPageForOther.html")
12  }
```

Cocoon and similar approaches exemplify a programming language concept called "continuations." (The interested reader is referred to Strachey and Wadsworth [2000], and Belapurkar [2004]). When the function in this example is executed and reaches line 5 (sendPageAndWait), the state of the program is saved (in session scope), and the specified page is sent to a client browser. If the browser then submits a form in reply (the user having pushed a button, for example), then the state of the program is restored and processing continues at line 6.

All the powerful semantics of the host programming language constructs are thus available (exception handling, for example) for specifying page flow.

CherryFlow is another such example, using the Python programming language [CherryPy 2005a].

6.15. Separated Concerns vs Page-Flow-Centric

At first glance, it may be difficult to see the distinction between some page-flow-centric approaches (Section 6.11) and those with separated concerns (Section 6.7), particularly the action-response models (Section 6.9). Many of these allow a programmer to specify which "view" should be shown depending on, say, the result (or outcome) of some action. The difference, though sometimes subtle, is important (as indicated by their respective positions in our taxonomy).

To a separated concerns (Section 6.7) strategy (particularly the action-response models of Section 6.9), the notion of "view" is basically the template to be used to generate the resulting document for the current request. Basically, these approaches attempt to generate a response for a request, using a template: They just allow the programmer to specify code separately from the template and provide a means for that code to influence which template will be used (so there is not a strict one-to-one mapping between the template used and the request URL). These approaches are the most sophisticated, most recent descendants in the lineage of static files (Section 6.1), executable templates (Section 6.4), separated concerns (Section 6.7).

A page-flow-centric strategy (Section 6.11) sees a view more strongly as a logical location in the UI and keeps track of where the user currently is: When a request comes in, the server will have some way of knowing where the user is (in terms of its conception of logical location in the UI). Its specification of page flow specifies not which template should be shown in response to a request; it has a wider, higher-level scope: It states which logical location can be transitioned to from others, depending on certain conditions.

While strategies with separated concerns are essentially presented as techniques for specifying Web sites with dynamic pages, page-flow-centric approaches are presented from the important perspective of aiming to specify UI which happen to be delivered via the Web.

6.16. Combined Strategies

It is interesting to note that the strategies for control concerns that have been presented here as distinct are often combined in a Web framework. For example, an implementation of a phased request (Section 6.8) can be built by having published objects (Section 6.6), each mandated to have methods named for particular phases in the request. The appropriate method of the object denoted by the request URL will then be called during the appropriate phase of the request. When categorizing such a combination of strategies, the differentiation is based on the basic intention of the whole combination, and not how that intention is achieved.

7. CLIENT-SIDE EXECUTION

Server-centric frameworks, by definition, rely mostly on code that is executed on the server-end of the client-server architecture of the Web. The architecture of the Web is defined in Fielding [2000] as an architectural style called representational state transfer (REST). REST is defined in terms of constraints. An example of such a constraint is that the protocol between client and server should be stateless: There are no rules governing the order of requests made from client to server. The intent of these constraints is that they will lead to certain properties in the final whole system. The statelessness constraint, for example, means that each request sent to the server contains all the information needed by intermediary components to deal with the request. Caching software, for example, can determine whether to cache a response to the request and just return the cached response to other clients making the same request. In this way, caches can help to alleviate load on a server, and minimize the usage of bandwith.

The constraints REST places on the server-end of an interaction naturally limit what can be done by a server-centric framework. These limitations are perceived to be causing problems in the resulting UI that these frameworks are able to provide: A UI developed using a server-centric framework suffers from perceived lack of responsiveness because most user actions imply another round-trip to the server. The UIs provided lack many features, such as drag-and-drop, to which users are accustomed in a normal GUI.

For such reasons, technologies (standardized and proprietary) are currently blossoming that attempt to do more at the client side of this interaction. Perhaps the most widespread of these is Adobe's proprietary product called Flash [Adobe Systems, Inc 2006].

The open-standard scalable vector graphics (SVG) is also relevant to this discussion. SVG allows for the construction of complex vector graphics and can thus be used to render a more pleasing UI than possible with HTML. By using JavaScript in combination with SVG, an SVG document can be made interactive and dynamic [W3C 2003].[18] Another open standard, synchronized multimedia integration language (SMIL), can also be used in conjunction with SVG to extend it with interactive capabilities specifically suited to multimedia [W3C 2005]. Such a combination of SVG with a scripting language is the open-standards-based alternative to Flash.

[18]The discussion in this section about JavaScript and asynchronous JavaScript technology and XML (AJAX) is thus applicable to the use of SVG, as well.

At present, browser support for SVG is still problematic [Lane 2007]. There is some anecdotal evidence to suggest that the adoption of SVG has slowed down, since the proprietary Flash is such a popular de facto standard.

As mentioned in Section 2, JavaScript, which adheres to the ECMAScript standard, is currently the only viable standardized means that a framework can use for shifting the execution of code to the client browser, and thus to the client side of a REST interaction. Furthermore, only recently have browser implementations of JavaScript become sufficiently compliant with the standard and sufficiently similar to each other for critical components of a framework to rely on JavaScript as an implementation vehicle.

The topic of JavaScript and how it relates to server-centric frameworks thus warrants a brief explanation here.

Standards such as HTTP and HTML make it possible to embed code inside a document which is sent to a browser for display. Such code is then executed by an interpreter at the client browser. The ECMAScript standard for JavaScript defines both language and, to a very limited extent, the programming libraries available for use in this way [ECMA General Assembly 1999].

Without the use of JavaScript (or something like it), each action of a user on an HTML page needs to be relayed to the server (as an HTTP request) where code can be executed to deal with it. The server responds in turn by sending a complete document back to the client.

JavaScript makes it possible that some user actions can be handled without this round-trip to the server. For example, JavaScript may be used (even by server-centric frameworks) to render UI widgets with some client-side execution, or it may be used to validate entered data on the client browser before submitting it to the server [Yahoo!, Inc. 2006; Mochi Media, LLC 2006; Lycos, Inc. 2006].

Using JavaScript, a programmer can attach code to several UI elements on a form which is being displayed in a browser. Upon clicking on a link, for example, JavaScript code can be executed *instead* of the click automatically resulting in a new HTTP request to the server.

AJAX is a term which refers to the technique of using JavaScript to intercept such UI events on the browser, and to then make a request (also via HTTP) to a Web server in the background.[19] Note that all this is done asynchronously in the background and without disturbing the original document being displayed (or the current state of the browser). A user can thus invoke other AJAX-supported actions while waiting for a response.[20] Upon receipt of the response to this background request, JavaScript is again used to *change* parts of the contents of the original document, all while the document is still being displayed.

Note that such a background request is a request for more information from the Web server, not for a complete (new) page to be rendered. The currently rendered page is merely modified, not rerendered. For this purpose, such requests usually return XHTML documents, which are nothing more than containers for whatever information the AJAX code in the browser needs in order to be able to correctly update the displayed document.[21] For example, such information can be used to populate a selection box, in response to a user having changed the state of a radio button.

[19]It should be noted that AJAX techniques require the use of the XMLHttpRequest object in the browser's JavaScript library, which is *not* part of the ECMAScript standard. Its presence in current browsers has become common enough, though, that its use has become viable.

[20]This is typically done in order to increase the perceived responsiveness of an application.

[21]Note that XML serves the purpose of an on-the-wire representation in this scenario. It is useful in this role because the JavaScript XMLHttpRequest object provides the functionality with which this representation can be parsed and manipulated on the receiving end.

AJAX thus involves HTTP in the role of a remote procedure call mechanism and JavaScript code in the client to exercise control over the UI and UI/user interaction. This shifts the locus of control of the UI to code that executes in the client-side browser.

Previously, implementations of JavaScript/ECMAScript in browsers varied so much that extensive use of this technology was seen by many as being impractical. The successful use in recent times of AJAX, by prominent companies such as Google, has prompted many programmers to develop sites and frameworks using AJAX and other JavaScript techniques. Many browser-centric frameworks are being developed based on AJAX.

Several problems remain to be dealt with by frameworks using such techniques. For example: Browser implementations of JavaScript still differ, and some techniques depend on JavaScript that is not included in the ECMAScript standard. From a security point-of-view, framework designers cannot trust that code sent to a browser will be executed as specified, since the browser has control over such execution and could potentially execute such code incorrectly (or thwart its execution selectively) in order to bypass the control implemented in JavaScript code. The asynchronous, concurrent nature of AJAX, in particular, requires more careful design of JavaScript code (and more skilled programmers).

As stated earlier, there are important reasons for exploiting the possibilities of using these approaches, despite the problems. Implementation problems such as the scalability of a server could be alleviated if more client resources could be used instead of server resources. Programming can be simplified by keeping state information relating to a user's UI session on a browser in a JavaScript program. As mentioned previously, the promise of being able to create UIs in browsers with more advanced features, such as found in GUIs, also holds allure for proponents of these techniques.

8. NOTES ON EXECUTION

This article focuses on how a programmer can specify a UI using different frameworks. The choices made by various framework designers regarding how such specifications are executed are also significant.

While not the focus of this article, a few notes in this regard are given here in order to illustrate the significance.

One issue is, for example, whether the notion of location (or view) in the Web UI corresponds with that of the browser. Browsers are built on the assumption that a particular URL denotes a particular location. Standard browser functionality is built around this assumption: Bookmarks can be placed for locations (by storing a URL) and a user can go backwards and forwards along a traversal path of such locations. If a framework is ignorant of the notions of a browser in this respect, it renders such basic browser functionality useless or confusing.

Many frameworks do not accommodate the perspective of the browser very well in their basic models, thus they often have problems resulting from such discrepancies. For instance, the popular and well-known *Struts* as well as the high-profile JSF do not correlate the "view" shown with the URL displayed by a browser.

The execution of some strategies also depends on functionality in a client browser that is not well standardized, if at all. As stated in Section 7, AJAX techniques rely on JavaScript functionality which falls outside the scope of the ECMAScript standard. Even in cases where such functionality is only reliant on standard ECMAScript code, the user has control over whether JavaScript code should be executed, and to what extent: JavaScript is an optional addition to the standard.

It is widely advocated on informal forums that such optional functionalities should be used with care [Tobias 2004; Korpela 2002]. In particular, a Web site should still be able to function in the absence of such functionalities, albeit with a less rich UI. Frameworks that are dependent on such optional functionality cannot degrade gracefully in the absence of such functionality, they simply cease to be able to operate.

9. CONCLUSION

This article gives an overview of server-centric Web frameworks that can render a Web-based UI using standardized components of the Web: Web browsers, servers, and standard means of communication between these. Section 7 alludes to a pull towards shifting the execution of such UI more to the browser instead of the server.

This pull towards client-side execution is relevant for server-centric frameworks also: The reliance on JavaScript (in whichever way) need not be restricted to browser-centric frameworks. Server-centric frameworks can also use these techniques fruitfully, and, depending on how much of their execution is shifted to the client browser, their classification as strictly server-centric could become problematic.

Nevertheless, server-centric frameworks continue to be in wide use in industry. These frameworks (and thus the different strategies they implement towards solving the problems in their domain) are more mature than newer classes of framework.

A UI framework cannot complete its task without relying on software installed at a client machine. The ubiquitous installed base of software that is compliant with Web standards provides a strong incentive for framework compliance with these standards.

The maturity of server-centric frameworks, and the importance for any framework to be compliant with open standards, motivated the focus of this study: standards-compliant, server-centric Web frameworks. The result is two taxonomies that provide a structured overview of server-centric approaches.

A similar investigation of browser-centric frameworks would be a fruitful next step. Since relevant examples of these use proprietary technology, such an investigation should not be limited to standardized technology only.

There are inherent problems with both server-centric and browser-centric frameworks. These problems derive from the limitations imposed on software on either side of the client-server divide dictated by the REST-style architecture of the standard Web components that make Web frameworks possible.

The REST architecture does not merely imply constraints, rather its constraints are designed to cause desirable properties in the resulting system. Moreover, these properties are probably directly responsible for the technical success of the Web [Fielding 2000].

This work is a start towards understanding what the possibilities and limitations are for specifying and executing UIs in an environment provided by a REST-based architecture such as seen in the Web.

APPENDIX

A. SURVEYED FRAMEWORKS AND RELATED PROJECTS

Note that the projects surveyed are to a large extent open-source projects (precisely for the reason that the designs of these are easily accessible). The effect is that authoritative information on these projects is Web-based. In this appendix each of the surveyed projects are listed by name, together with the project Web site (or equivalent). Each entry in the list contains the title of the referenced Web site. This is followed by one of the following, the copyright holder or author of the project, or the copyright holder of the Web page (if such information is available). Each entry ends with the URL of the project Web site. All these URLs have been referenced within the period of January to August 2005, except where indicated.

ActionServlet ActionServlet. Petr Toman and Mark D. Anderson.
`http://www.actionframework.org`

Albatross Albatross—A toolkit for stateful Web applications. Object craft P/L.
`http://www.object-craft.com.au/projects/albatross`

AppServer Faces AppServer faces. Dataosoft.
`http://www.dataosoft.com/asf`

Aquarium Aquarium.
`http://aquarium.sf.net`

ASPy ASPy—Active server Python. Bradley Schatz.
`http://archive.dstc.edu.au/aspy`

Barracuda Barracuda presentation framework.
`http://barracudamvc.org`

Beehive Beehive. The Apache Software Foundation.
`http://incubator.apache.org/beehive`

Bento Bentodev.org—Home of the Bento language. bentodev.org.
`http://www.bentodev.org`

Bishop Bishop. Johan Redestig.
`http://bishop.sourceforge.net`

Castalian Castalian. Stuart Langridge.
`http://www.kryogenix.org/code/castalian`

Cheetah Cheetah—The Python-powered template engine. Tavis Rudd.
`http://www.cheetahtemplate.org`

CherryFlow CherryFlow.
`http://subway.python-hosting.com/wiki/CherryFlow`

CherryPy CherryPy—A Pythonic, object-oriented Web development framework. CherryPy team.
`http://cherrypy.org`

Chrysalis Chrysalis. Paul Strack.
`http://chrysalis.sourceforge.net`

Cocoon The Apache Cocoon project. The Apache Software Foundation.
`http://cocoon.apache.org`

ColdFusion Adobe—Products : ColdFusion MX 7. Adobe Systems Incorporated.
`http://www.adobe.com/products/coldfusion`

DWR DWR—Easy AJAX for Java. Joe Walker and Getahead.
`http://www.getahead.ltd.uk/dwr`

Echo NextApp . Echo. NextApp Incorporated.
`http://www.nextapp.com/products/echo`

Echo2 NextApp . Echo2. NextApp Incorporated.
`http://www.nextapp.com/platform/echo2/echo`
(last accessed July 2006)

ECS Jakarta ECS—Element construction set. The Apache Software Foundation.
`http://jakarta.apache.org/ecs`

EmPy EmPy. Erik Max Francis.
`http://www.alcyone.com/software/empy`

Expresso Expresso framework project. Jcorporate Limited.
`http://jcorporate.com/expresso.html`

FormKit dAlchemy—FormKit : A Webware form library. dAlchemy, Incorporated.
`http://dalchemy.com/opensource/formkit`

FreeMarker	FreeMarker. The FreeMarker Project. `http://freemarker.sourceforge.net`
HTMLgen	HTMLgen. Robin Friedrich. `http://starship.python.net/crew/friedrich/HTMLgen/html/main.html`
Jaguar	Jaguar for Python: Lean, fast and mean. Terrel Shumway. `http://jaguar.sourceforge.net`
JBanana	JBanana. JBanana. `www.jbanana.org`
JonPy	Jon's Python modules. Jon Ribbens. `http://jonpy.sf.net`
JPublish	JPublish. Anthony Eden. `http:///www.jpublish.org`
JSPWidget	SPWidget Open Source Project. `http://edu.uuu.com.tw/jspwidget/default.jsp`
Jucas	Jucas object oriented pull MVC Web-framework. Jucas. `http://jucas.sourceforge.net`
Karrigell	Karrigell. Pierre Quentel. `http://karrigell.sourceforge.net`
Keel	Keel framework. Keel Groups Limited. `http://www.keelframework.org`
Maverick	Maverick. Infohazard.org. `http://mav.sourceforge.net`
Melati	Melati—Java SQL Website development engine. PanEris. `http://www.melati.org`
Millstone	Millstone—Free open source Web UI library and reusable components for J2EE and Java. IT Mill Limited. `http://millstone.org`
Nevow	Nevow: A Web application construction kit. Donovan Preston. `http://www.divmod.org/projects/nevow`
Niggle	Niggle Web application framework. Jonathan Revusky. `http://niggle.sourceforge.net`
Myghty	Myghty—High performance Python templating framework. Michael Bayer. `http://www.myghty.org/index.myt`
OpenLaszlo	OpenLaszlo—The premier open-source platform for rich Internet applications. Laszlo Systems, Inc. `http://www.openlaszlo.org` (last accessed July 2006)
PEAK.Web	PEAK—The Python enterprise application kit. Phillip J. Eby. `http://peak.telecommunity.com`
PHP	PHP: Hypertext processor. The PHP Group. `www.php.net`
PMZ	PMZ—Poor man's Zope. Andreas Jung. `http://pmz.sourceforge.net`
PSP	Python server pages (PSP). Angell Enterprises, Inc. `http://www.ciobriefings.com/psp`
PyHP	Python hypertext preprocessor. Lethalman and Christopher A. Craig. `http://freshmeat.net/projects/pyhp`

PyML PyML—Python HTML pre-processor. David Snopek.
`https://gna.org/projects/pyml`

PyServ PyServ—A servlet-like engine for Python. Steve Purcell.
`http://pyserv.sourceforge.net`

Python Pages Ramu's Python page. Ramu Chenna.
`http://www.embl-heidelberg.de/~chenna/pythonpages`

pyWeb pyWeb HOME. David McNab.
`http://www.freenet.org.nz/python/pyWeb`

PyWX PyWX: Python for AOLserver.
`http://pywx.idyll.org`

Quixote Quixote. Corporation for National Research Initiatives.
`http://www.mems-exchange.org/software/quixote`

RIFE RIFE: About. The RIFE team.
`http://rifers.org`

Roadkill Roadkill—Embedded Python.
`http://roadkill.sourceforge.net`

RubyOnRails Ruby on rails. David Heinemeier Hansson.
`http://rubyonrails.com`

Seaside Seaside. Avi Bryant.
`http://seaside.st`

Sitemesh SiteMesh. OpenSymphony.
`http://www.opensymphony.com/sitemesh`

SkunkWEB SkunkWEB—News. Drew Csillag et al.
`http://skunkWeb.sf.net`

Smarty Smarty : Template engine. New Digital Group, Inc.
`http://smarty.php.net`

Smile Smile, the open source JavaServer Faces implementation. Dimitry D'hondt, Edwin Mol and Steve van den Buys.
`http://smile.sourceforge.net`

Snakelets SNAKELETS—Python Web application server. Irmen de Jong.
`http://snakelets.sourceforge.net`

Sofia Sofia—JSP GUI Java development framework. Salmon LLC.
`http://www.salmonlcc.com/sofia`

Spring Spring framework.
`http://www.springframework.org`

SpringWebFlow Spring Web flow. Ervacon.
`http://www.ervacon.com/products/springWebflow`

Spyce Spyce—Python server pages (PSP). Rimon Barr.
`http://spyce.sf.net`

Struts Struts. The Apache Software Foundation.
`http://jakarta.apache.org/struts`

Subway Subway.
`http://subway.python-hosting.com`

Swinglets Javelinsoft. Javelin Software.
`http://www.javelinsoft.com/swinglets`

Tapestry Jakarta tapestry. The Apache Software Foundation.
`http://jakarta.apache.org/tapestry`

Teaservlet	Tea trove project. Walt Disney Internet Group. `http://teatrove.sourceforge.net`
Tiles	Tiles. Cedric Dumoulin and The Apache Software Foundation. `http://www.lifl.fr/~dumoulin/tiles`
Turbine	Jakarta turbine. The Apache Software Foundation. `http://jakarta.apache.org/turbine`
TwistedMatrix	Twisted Matrix Laboratories. `http://twistedmatrix.com`
TwistedWeb	Overview of twisted Web. `http://twistedmatrix.com/projects/Web/documentation/howto/Web-overview.html`
Velocity	Velocity. The Apache Software Foundation. `http://jakarta.apache.org/velocity`
VRaptor	VRaptor—Simple Web MVC framework. Arca. `http://www.vraptor.org`
Wasp	Wasp documentation. Robin Parmar. `http://www.execulink.com/~robin1/wasp/readme.html`
WebMacro	Web macro. Semiotek Inc. `http://www.Webmacro.org`
WebOnSwing	WebOnSwing—Multi environment application framework. Frebes. `http://Webonswing.sourceforge.net`
Webware	Webware For Python Wiki. `http://wiki.w4py.org/`
Webwork	Webwork. OpenSymphony. `http://www.opensymphony.com/Webwork`
Wicket	Wicket. Wicket developers. `http://wicket.sourceforge.net`
wingS	wingS—LGPL open source. `http://wings.mercatis.de`
Woven	Woven. `http://twisted.sourceforge.net/TwistedDocs-1.2.0rc3/howto/woven.html`
Zope	Zope.org. Zope Corporation. `http://www.zope.org`

REFERENCES

ADOBE SYSTEMS, INC. 2006. Adobe—Flash player. `http://www.macromedia.com/software/flash/about/` (last accessed August 2006).

BARRACUDAMVC. 2006. Barracuda mvc. `http://barracuda.enhydra.org` (last accessed July 2006).

BELAPURKAR, A. 2004. Use continuations to develop complex Web applications: A programming paradigm to simplify MVC for the Web. `http://www-128.ibm.com/developerworks/library/j-contin.html` (last accessed August 2005).

CERI, S., FRATERNALI, P., AND BONGIO, A. 2000. Web modeling language (WebML): A modeling language for designing Web sites. In *Proceedings of the 9th International World Wide Web Conference on Computer Networks. Int. J. Comput. Telecommun. Netw*. North-Holland Publishing, Amsterdam, The Netherlands, 137–157.

CHERRYPY TEAM. 2005a. CherryFlow. `http://subway.python-hosting.com/wiki/CherryFlow` (last accessed August 2005).

CHERRYPY TEAM. 2005b. CherryPy—A pythonic, object-oriented Web development framework. `http://cherrypy.org` (last accessed August 2005).

COPELAND, D., CORBO, R., FALKENTHAL, S., FISHER, J., AND SANDLER, M. 2000. Which Web development tool is right for you? *IT Profession. 2*, 2 (Mar./Apr.), 20–27.

CORPORATION FOR NATIONAL RESEARCH INITIATIVES. 2005. Quixote 2.1. `http://www.mems-exchange.org/software/quixote` (last accessed August 2005).

COWARD, D. AND YOSHIDA, Y. 2003. *Java^{TM} Servlet 2.4 Specification*. Sun Microsystems.

CRESPO, A., CHANG, B.-W., AND BIER, E. A. 1997. Responsive interaction for a large Web application: The meteor shower architecture in the Webwriter II editor. In *Selected Papers from the 6th International Conference on World Wide Web*. Elsevier Science, Essex, UK, 1507–1517.

DEMICHIEL, L. G. 2003. *Enterprise JavaBeans^{TM} Specification, Version 1.2*. Sun Microsystems.

DIJKSTRA, E. W. 1968. Letters to the editor: Go to statement considered harmful. *Commun. ACM 11*, 3, 147–148.

DRAHEIM, D. AND WEBER, G. 2002. An overview of state-of-the-art architectures for active Web sites. Tech. Rep. Institute of Computer Science, Free University Berlin.

DRAHEIM, D. AND WEBER, G. 2005. Modeling form-based interfaces with bipartite state machines. *Interact. Comput. 17*, 2 (Mar.), 207–228.

DÜRST, M. J. 2005. The HTTP charset parameter. `http://www.w3.org/International/O-HTTP-charset.html` (last accessed August 2005).

ECMA GENERAL ASSEMBLY. 1999. *ECMAScript Language Specification*, 3rd ed. European Computer Manufacturers Association.

ERVACON. 2004. Spring Web flow. `http://www.ervacon.com/products/springWebflow` (last accessed August 2005).

FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. 1999. Hypertext transfer protocol – HTTP/1.1. Internet Engineering Task Force. RFC2616.

FIELDING, R. T. 2000. Architectural styles and the design of network-based software architectures. Ph.D. thesis, University of California, Irvine.

FRATERNALI, P. 1999. Tools and approaches for developing data-intensive Web applications: A survey. *ACM Comput. Surv. 31*, 3 (Sept.), 227–263.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA.

GINIGE, A. AND MURUGESAN, S. 2001. Web engineering: An introduction. *IEEE Multimed. 8*, 1 (Jan./Mar.), 14–18.

GÜELL, N., SCHWABE, D., AND VILAIN, P. 2000. Modeling interactions and navigation in Web applications. In *Proceedings of the Workshops on Conceptual Modeling Approaches for E-Business and The World Wide Web and Conceptual Modeling (ER)*. Springer-Verlag, London, UK, 115–127.

HELMAN, T. AND FERTALJ, K. 2003. A critique of Web application generators. In *Proceeding of the 25th International Conference on Information Technology Interfaces (ITI)*. IEEE Computer Society, Washington, DC, 639–644.

JCORPORATE LTD. 2005. Expresso framework project. `http://jcorporate.com/expresso.html` (last accessed August 2005).

JOURAVLEV, M. 2004. Redirect after post. `http://www.theserverside.com/articles/article.tss?l=RedirectAfterPost` (last accessed August 2005).

KNAPP, A., KOCH, N., MOSER, F., AND ZHANG, G. 2003. ArgoUWE: A CASE tool for Web applications. In *Proceedings of the 1st International Workshop on Engineering Methods to Support Information Systems Evolution (EMSISE)*.

KOCH, N. AND KRAUS, A. 2002. The expressive power of UML-based Web engineering. In *Proceedings of the 2nd International Workshop on Web-Oriented Software Technology (IWWOST)*, D. Schwabe et al., Eds. 105–119.

KORPELA, J. 2002. Augmentative authoring—A different look at "graceful degradation" in Web authoring. `http://www.cs.tut.fi/~jkorpela/html/augm.html` (last accessed August 2005).

KRASNER, G. AND POPE, S. 1988. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *J. Obj. Orient. Program. 1*, 3, 26–49.

LANE, D. 2007. Scalable vector graphics. *J. Online Math. Appl. 7*, Article ID 1381 (Feb.).

LYCOS, INC. 2006. Webmonkey: Javascript_code_library. `http://www.Webmonkey.com/Webmonkey/reference/javascript_code_library` (last accessed August 2006).

MCCLANAHAN, C., BURNS, E., AND ROGER KITAIN, E. 2004. *JavaServer^{TM} Faces Specification, v1.1*. Sun Microsystems.

MCCOOL, R., FIELDING, R., AND BEHLENDORF, B. 1999. How the Web was won. *Linux Mag*.

Mellor, S., Clark, A., and Futagami, T. 2003. Model-Driven development—Guest editor's introduction. *Softw., IEEE 20*, 5 (Sept./Oct.), 14–18.

Microsoft Corporation. 2005. ASP.NET Web: The official microsoft ASP.NET site. `http://www.asp.net` (last accessed September 2005).

Mochi Media, LLC. 2006. Mochikit—A lightweight javascript library. `http://www.mochikit.com` (last accessed August 2006).

NextApp Inc. 2005. NextApp . Echo. `http://www.nextapp.com/products/echo` (last accessed August 2005).

Object Craft P/L. 2005. Albatross—A toolkit for stateful Web applications. `http://www.object-craft.com.au/projects/albatross` (last accessed August 2005).

Pierre Delisle, E. 2002. *JavaServer Pages*^{TM} *Standard Tag Library*. Sun Microsystems.

Python Software Foundation. 2005. Python programming language. `http://www.python.org` (last accessed August 2005).

Qin, S. F., Harrison, R., West, A. A., Jordanov, I. N., and Wright, D. K. 2003. A framework of Web-based conceptual design. *Comput. Ind. 50*, 2, 153–164.

Radke, A. 1999a. Http digest authentication. `http://static.userland.com/userLandDiscussArchive/msg012483.html` (last accessed September 2005).

Radke, A. 1999b. Re: Http digest authentication. `http://static.userland.com/userLandDiscussArchive/msg012533.html` (last accessed September 2005).

Raggett, D., Hors, A. L., and Jacobs, I. 1999. *HML 4.01 Specification*. W3C. W3C Recommendation.

Rode, J., Rosson, M. B., and Perez-Quinones, M. A. 2004. End-Users' mental models of concepts critical to Web application development. In *Proceedings of the IEEE Symposium on Visual Languages—Human Centric Computing (VLHCC)*. IEEE Computer Society, Washington, DC, 215–222.

Roth, M. and Pelegrí-Lopart, E. 2003. *JavaServer Pages*^{TM} *2.0 Specification*. Sun Microsystems.

Rudd, T. 2001–2005. Cheetah—The python-powered template engine. `http://www.cheetahtemplate.org` (last accessed August 2005).

Rushing, S. 2005. Medusa: A high-performance internet server architecture. `http://www.nightmare.com/medusa/medusa.html` (last accessed August 2005).

Schranz, M. W., Weidl, J., Gschka, K. M., and Zechmeister, S. 2000. Engineering complex world wide Web services with JESSICA and UML. In *HICSS '00: Proceedings of the 33rd Hawaii International Conference on System Sciences (HICSS)*, vol. 6. IEEE Computer Society, Washington, DC, 6068.

Seshadri, G. 1999. Understanding JavaServer Pages model 2 architecture: Exploring the MVC design pattern. `http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-jspmvc.html` (last accessed August 2005).

Spolsky, J. 2003. The absolute minimum every software developer absolutely, positively must know about unicode and character sets (no excuses!). `http://www.joelonsoftware.com/articles/Unicode.html` (last accessed August 2005).

Strachey, C. and Wadsworth, C. P. 2000. Continuations: A mathematical semantics for handling fulljumps. *Higher Order Symbol. Comput. 13*, 1-2, 135–152.

The Apache Software Foundation. 2000–2005a. Struts. `http://jakarta.apache.org/struts` (last accessed August 2005).

The Apache Software Foundation. 2003–2005b. The Apache Cocoon project. `http://cocoon.apache.org` (last accessed August 2005).

The PHP Group. 2001–2005. PHP: Hypertext processor. `http://www.php.net` (last accessed August 2005).

Thomas, D. 2004. MDA: Revenge of the modelers or UML utopia? *Softw., IEEE 21*, 3 (May-Jun.), 15–17.

Tobias, D. R. 2004. Dan's Web tips: Graceful degradation. `http://Webtips.dan.info/graceful.html` (last accessed August 2005).

Vosloo, I. 2005. A Web application user interface specification language based on statecharts. M.S. thesis, University of Pretoria, Pretoria, South Africa.

W3C HTML Working Group. 2002. *XHTML*^{TM} *1.0 The Extensible HyperText Markup Language: A reformulation of HTML in XML 1.0*, 2nd ed. World Wide Web Consortium (W3C). W3C Recommendation.

W3C SVG Working Group. 2003. Scalable vector graphics (SVG) 1.1 specification. `http://www.w3.org/TR/2003/REC-SVG11-20030114/`. W3C Recommendation.

W3C SYMM Working Group. 2005. Synchronized multimedia integration language (SMIL 2.1). `http://www.w3.org/TR/2005/REC-SMIL2-20051213`. W3C Recommendation.

Westkämper, T. 2004. Architectural models of J2EE Web tier frameworks. M.S. thesis, University of Tampere.

WICKET DEVELOPERS. 2004–2005. Wicket. `http://wicket.sourceforge.net` (last accessed September 2005).

WINCKLER, M. AND PALANQUE, P. 2003. StateWebCharts: A formal description technique dedicated to navigation modeling of Web applications. In *Proceedings of the 10th annual International Workshop on Interactive Systems Design, Specification, and Verification (DSV-IS)*, Funchal, Madeira Island, Portugal, Revised Papers. Lecture Notes in Computer Science, vol. 2844/2003. Springer-Verlag, GmbH, 61–76.

YAHOO!, INC. 2006. Yahoo! UI library. `http://developer.yahoo.com/yui` (last accessed August 2006).

YERGEAU, F., COWAN, J., BRAY, T., PAOLI, J., SPERBERG-MCQUEEN, C. M., AND MALER, E. 2004. *XML 1.1*. W3C. W3C Recommendation.

ZOPE CORPORATION. 2003. Zope.org. `http://www.zope.org` (last accessed August 2005).

Received September 2005; revised April 2007; accepted May 2007