Web-based development: Putting practice into theory

Iwan Vosloo
Reahl Software Services (Pty) Ltd
PO Box 14240
Hatfield
0028
iwan@reahl.org

ABSTRACT

Against the backdrop of the current web development arena, this paper sketches an overview of work done by the author towards achieving a higher-level abstraction to program web applications to. The paper illuminates unexpected problems encountered when applying these theoretical ideas in practice and points out a number of directions for future research. In conclusion, the paper takes a brief look at how Reahl currently compares to other web frameworks and Content management systems (CMSs).

1 INTRODUCTION

Web-based development (still) takes a lot of effort—too much: During the development of web applications, a lot of programmer time goes into dealing with low-level technical details. These details are not directly related to the the task at hand, and they have to be dealt with repeatedly.

This observation warrants a hypothesis: isn't it possible to create a higher-level "virtual machine" which exposes abstractions on a level that frees the programmer from these tedious low-level distractions? What would the abstractions of such a machine look like, and how will they be implemented?

With such a machine programmers would be more productive and would need to know less about various low-level topics.

Currently a web developer needs to know quite a bit about these issues, but very few developers actually do have this knowledge—resulting in web applications that are riddled with security flaws, are incompatible across different browsers, and are generally lacking in quality.

The pursuit of a high-level machine or language for building web applications seems quite profitable.

This paper gives a light overview of the author's persuit to date and attempts to point out a number of possible directions for future research in this area.

Dedicated to my Guru, Derrick G Kourie.

2 BACKGROUND

2.1 How web applications are built

There are two predominant ways of building web applications today: use a web framework or use a CMS.

CMSs originated as almost pre-built websites to which authors can add pages with textual and image content without having to know anything about web development.

CMSs also come with pre-built "modules" of functionality that users can incorporate into these sites. Examples of such modules are discussion forums, FAQs, modules for online polls, online shops, etc.

A web framework, on the other hand, is meant for a more technical audience. Web frameworks grew, like other programming frameworks, from programmers who simply put the code for doing repetitive tasks in libraries so that such code could be re-used.

Having started on such a path, a programmer inevitably needs to make plans to solve certain common problems that are encountered by many web developers[1].

A modern web framework is really a collection of such plans made to solve some common problems, together with reusable code libraries and even binaries for certain tasks.

CMSs are usually themselves built using a web framework.

A CMS comes with a lot of pre-built functionality and it allows a non-technical user to be quite productive. But, it is somewhat restrictive when it comes to customising such functionality or look and feel.

Web frameworks, in contrast, provide the flexibility expected by a programmer, at the cost of having to be a pro-

[1]It is out of the scope of this paper to enumerate and explain all of these, but the interested reader is encouraged to read [18] for a comprehensive list.

grammer. Web frameworks also are not bundled with any pre-built end-user modules as is the case with CMSs.

Web frameworks and CMSs really are different approaches towards solving the same problem. Web frameworks evolve from the insight and lessons learnt from the low level environment of the web—from what is possible and what plans can be made to make it possible. CMSs evolve from user requirements—what people generally want to be able to build and with what ease.

2.2 Previous work

The present investigation was motivated by the aim of building a web framework on a high enough level of abstraction to make web development less cumbersome and expensive.

In [19], the typical requirements of a web framework were enumerated. Two of the important requirements were then used as basis for an investigation into a number of open source web frameworks. The approaches taken towards these problems by the different web frameworks were then categorised. The results were presented in the form of two taxonomies.

A visual specification language was proposed in [18] as an attempt to provide a basic foundation for a higher-level web framework. The same work also includes detailed explanations of how this language could be implemented. The aim of the language was to:

- visualise the dynamic structure of the user interface
- see how far one can go while sticking to web standards and best practices

This language was called Harel, after the inventor of statecharts—the formalism upon which the language was based [8].

The bulk of this paper is dedicated to lessons learnt from implementing Harel, and using it in real world scenarios.

To acommodate the reader who is not familiar with Harel, the next section diverts briefly in order to introduce the ideas around which Harel was designed.

2.3 Harel

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 User interface (UI). It is to locations (or pages) in a web UI what control flow is to statements in a programming language.

Page flow is closely related to the navigational model, as used in Model driven architecture (MDA) approaches to developing web applications and hypertext systems [20].

The source code of a web application built with a current web framework does not readily reflect the page flow of an application. A programmer has to visualise this aspect of the design by mentally mapping the design to the scattered information from various files, templates and programming code.

A screenshot of a popular tool, Eclipse, is shown in Figure 1 to illustrate the view a Java server faces (JSF) programmer

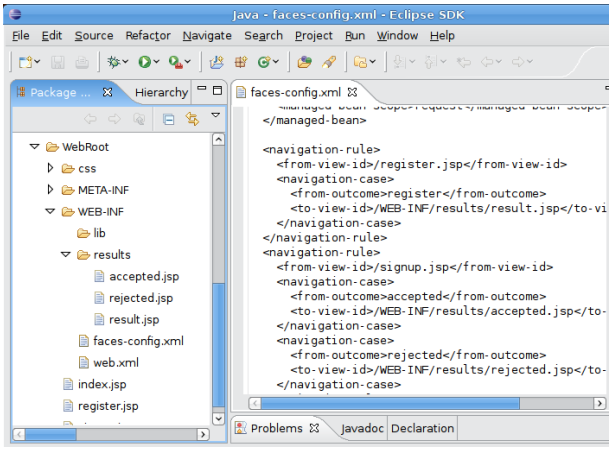

Figure 1: The view a JSF programmer has on source code of a web application

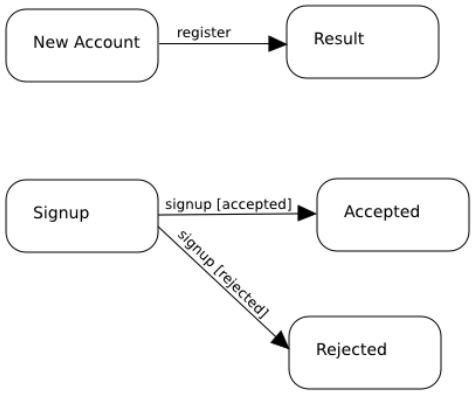

Figure 2: The design of the page flow of the simple JSF application

would have on the source code of a simple web application built using this technology.

However, when the UI of such a web application is built, designs are often done by sketching the page flow diagramatically, as shown in Figure 2.

Harel is an attempt to specify a web application as a whole by structuring its code around a visual design of its page flow. It is based on statecharts [8], in their incarnation as State Diagrams in UML [1].

The basic idea is thus that a programmer should be able to design an application by first designing its page flow, very much as is done in Figure 2. This basic structure can then be annotated to also specify dynamic behaviour. Concepts from the State diagram formalism lend themselves excellently towards specifying this dynamic behaviour.

After having drawn a basic page-flow design, a programmer would then be able to zoom into a particular page in order to specify what that particular page looks like, and

what UI elements it contains.

A web page in Harel is called a "location" and is analogous to a state in state diagrams.

The adoption of statechart concepts also means that the language allows for the creation of re-usable chunks of website. Such reusable chunks are somewhat anologous to procedure calls in a structured language. For example, a user could choose to "add a bank account" from several locations in a web site. This would take the user through a series of screens and return the user to where the choice was made originally to "add a bank account".

Among web frameworks, Harel is unique in terms of: its visual nature; the way in which it allows dynamic behaviour (controller concerns) to be specified in a single place (annotations on the diagram); and the ability to have a reusable chunk of UI logic analogous to a procedure.

3 LESSONS LEARNT

Since the formulation of Harel as a language, several incarnations of it has been implemented. Some fundamental changes were also introduced to the original language in order to cater for problems encountered while using these implementations to build real world web applications for paying customers. The latest permutation of the language is called Reahl.

Upon using Harel in practice, it became apparent that the problems encountered are not confined to a user interface. Many of these problems cannot be satisfactorily solved in isolation either: individual solutions impact one another. Some solutions also span the different architectural layers from presentation down to persistence.

To solve these problems, changes had to be made to Harel (now Reahl), and a lot of additional infrastructure had to be developed that is not directly related to the language itself.

Because solutions to these problems had to be found quickly, the particular solutions were shaped very much by the specific client environments the problems presented themselves in.

The rest of this section briefly introduces some of the more important problems and the current solutions implemented in Reahl.

3.1 Dealing with static content

Harel was aimed at dealing with complex interactive websites. Its initial design overlooked the fact that websites are rarely *only* complex and dynamic. Websites typically have a lot of static, book-like content that eventually leads to parts with more interesting dynamic behaviour.

A web application built with Reahl is a program—something that has to be tested, versioned, packaged and distributed in a rigorous fashion.

It is cumbersome to deal with small changes to static content in this way. Moreover, some clients employ their own web designers and it was necessary to come up with a way for these people to be able to add and change static pages using the tools they are used to and have invested in.

Not all pages are *only* static—sometimes pages need to be static with a bit of dynamic content added. The static pages created by the client web designers needed to be dealt with as if they were part static and part dynamic.

3.1.1 Allowing for simple structure

Static websites generally have hierarchical structure between pages.

State diagrams allows one to build hierarchies, since each composite state in the diagram can contain substates. However, such a diagram is much more than a hierarchy, because of the transitions and additional annotations relating to dynamic behaviour.

Simple states of State diagrams cannot contain further substates, and thus are leaves in a hierarchy. And since they do not have substates they also do not have transitions or any other of the dynamic features.

In Reahl, simple locations[2] were changed to be able to contain other locations, but without the ability to have transitions between contained locations.

This addition makes it possible to create simple tree structures found in other web frameworks—something that was initially overlooked. Moreover, such simple tree structures can be merged with parts that have dynamic behaviour, since a simple location can also contain a composite location.

3.1.2 Locations that are not part of source code

The next problem tackled is that of being able to deal with static parts of a site as data—so that it could be changed without influencing the source code of the site.

To solve this issue the framework was changed so that each site would have a "static root". A static root is merely a directory containing static files, exactly like the document root of a web server like Apache. The static root use by a site is set by configuration.

The framework was changed to behave differently should an Uniform resource locator (URL) be requested which it cannot find a location for in site. It will first check whether it can map the URL to an existing file relative to the static root of the site. If this is possible, the framework dynamically, but temporarily, adds a new kind of location for such a static file to the site itself.

This allows Reahl to serve static files even though they are not part of the source code.

Web designers use programs such as Adobe Dreamweaver and Microsoft Frontpage to create static websites. Using such programs a hierarchy of directories and static files can be built and uploaded to a directory on a server via a number different means. With the right configuration on a web server such as Apache, it is possible to grant a web designer access to the static root of a Reahl website. This way designers are able to use their own familiar tools to build static content, and upload it to a Reahl website.

3.1.3 Rendering static pages dynamically

One more piece of the puzzle was missing. The static pages located in a static root of a Reahl website would be rendered

[2]In Reahl, "location" is the term analogous to "state" in a State Diagram.

by sending the static file unchanged back to a web browser. This is not always what is desired.

For example, it may be necessary to display on any given page of a site whether the current user is logged in or not. While static pages can be made to look exactly like other dynamic pages in the same site, they are static and cannot change depending on whether a user is logged in or not.

To solve this problem, the framework had to be changed so that it would not merely render a dynamically added static location unchanged. Rather, it would render the normal dynamic page template used for the site and cherry-pick parts out of the static page on the file system to be inserted in the right places on a dynamic page.

While this plan worked well with static Hypertext markup language (HTML) pages, accompanying static files, such as pictures still needed to be rendered unchanged. The framework is able to distinguish between these cases by the extension of the files.

This addition to the framework now allows the web designers of a client to design static pages, upload them to a Reahl site, but still have them displayed as if they were dynamic. Dynamic elements, such as the login status of the user, or even banner ads continue to work. This last addition makes a static page look and feel 100% like the rest of the site.

3.2 Reusable libraries

One CMS feature sorely missed while using the Reahl framework was the ability to pre-build components with domain functionality that can be re-used in many websites.

Using Reahl happens in the context of a company with many small clients who have overlapping needs. It makes sense to build solutions for such needs once, and use the solution for several clients.

Reahl was designed from the beginning to be able to ship parts of the website UI as reusable modules. But such modules only contained UI code. What was needed was a kind of component that included these screens as well as application and database code. In other words, the module needs to cut across architectural layers all the way from presentation through to persistence.

All of Reahl is built using the Python programming language. An extension is available in Python which makes it possible to package Python code in something reminiscent of a Java Jar file. In Python these are called Python Eggs [13]. A Python Egg includes more metadata than a Jar file. The metadata in a Python Egg allows for functionality akin to the Open services gateway initiative (OSGI) standards [15]:

- each egg has a name and version;
- one can specify which other (versioned) eggs an egg is dependent on; and
- eggs can publish interfaces that can be discovered at run-time and used by other eggs.

This mechanism was adopted by Reahl for packaging a reusable component. Such an egg would then comprise Reahl code and Python code (which database logic is written in). With very little effort it was thus possible to extend Reahl with an OSGI-like component system that is aware of dependencies between components, and is readily packaged for distribution on various platforms.

Having such packaged components, however, is not enough. They, as well as the Reahl framework underneath, keep changing while in active use on several client websites. These changes create a lot of scope for error and a lot of overhead when it is necessary to upgrade an egg that is currently in use on several sites. Most of the problems here result from the fact that a new version of a component often entailed changes to the database schema previously used. The underlying database schema needs to be migrated as part of an upgrade. And this needs to happen for each distinct site—but only for the modules used by that specific site.

To solve this problem, a bit of infrastructure was built, based on the functionality provided by Python eggs. A Reahl website is also packaged as an egg. Each egg component advertises an interface that can be used by automated tools for all sorts of database-related tasks, such as migrating the database schema for that egg, creating an initial schema or even to run daily maintenance tasks.

From the dependency information stored in each egg, a dependency tree can be derived. This tree is used by automated tools to determine which modules form part of a particular site. The tree also is used to determine the logical order in which the database migration for different eggs should be executed.

On the web UI side of things, another small innovation was needed in order to allow the re-use of modules. Normally, web frameworks would allow one to have a "master template" which will be used for any page rendered. The template for each page typically consists of "including" the master template, and inserting the page-specific details in it.

This means that the name of the master template used is hard-coded in the source code of each page. For a page that may form part of different websites, each with their own "master template", this would not do. In Reahl it is now possible to create a "master template" which defines a number of slots. This template is inherited down the hierarchy of pages, and any page can use this template regardless of where it came from, and without having to hard-code its name. Master templates merely need to be written with the correct slots, the content for which will be provided by the actual page.

3.3 Security and access control

In any multiuser system security is a tricky subject. Somehow one needs to specify who has access to what information (or object), and exactly what that access entails.

The security requirements of applications vary in sophistication and granularity.

Take the example of an investment company where a user can invest online. The the company would keep a record of each customer portfolio, perhaps viewable on a particular URL with the URL parameters indicating which portfolio needs to be displayed.

A simple solution is to allow only particular users access to that specific URL. This, however would mean that anyone who can see a portfolio can see the portfolio of anyone

else, should they be able to craft the correct URL. This is probably not a desired feature. In such a case security needs to be informed by the domain data as well, so that the current user can only see his own portfolio. How the domain model informs the security model can become quite complicated and depends on the domain. There may be advisors who are able to see the portfolios of other users. But the system should restrict an advisor to see only the portfolios of users she had a contract with. It may also be that some advisors have additional power of attorney to effect changes in a portfolio—but only certain types of changes.

A complex enough system usually has a mixture of such access control needs. Deciding what is possible, and how to implement it is not a trivial problem. Any web application large enough to matter needs to provide a way to be able to solve this problem.

In a web application one also needs to be careful of how you implement access control code. For example: a particular screen could have a link to "change portfolio", depending on the rights of the current user. If the user clicks on "change portfolio", she will be presented with a screen where she can perform all sorts of actions on that portfolio.

To merely remove the link on the first page for users who should not have access to the portfolio is not an acceptable solution. An attacker could know the URL and be able to gain access to the "change portfolio" screen regardless of whether a link was presented or not. The "change portfolio" screen itself also needs to prevent access to itself in order to stop an attacker.

A more subtle version of this problem is also possible. Security-sensitive code may be written in JavaScript. But JavaScript is executed on the client browser, and an attacker could modify a browser to execute the given code in unintended ways, or replace it with his own.

Ideally a programmer would want to specify security and access control restrictions in a central location.

But the programmer has to take cognisance of the access that apply in several different places: when rendering the first screen (to be able to know whether to render a link or not); when the second screen is requested (to stop a possible attacker); and when requests relating to these screens are received by a server.

3.3.1 Access control in Reahl

Currently, Reahl does not have an elegant high-level solution for such security issues. A module was built containing the necessary domain model, based on analysis patterns proposed by Martin Fowler in [6]. This domain model allows the system to keep track of which roles are played by particular parties.

Access to any domain object from the web takes place via an interface (this solution follows the Facade pattern, [5]). This strategy renders the interface a central point where access control can be enforced.

For the generation of screens, a separate mechanism is used, called adaptive widgets:

Reahl includes a set of objects that can be rendered as HTML UI widgets. These are widgets like text fields, or text input boxes, or selection boxes, etc. Each widget is adorned upon creation with functions it can call to determine whether the current user:

- is able to see it at all; or
- is able to change it (which implies the first).

When a page is generated, a widget first calls its functions in order to determine whether it is visible at all, read-only or writable for the current user. It is then rendered accordingly.

It is unfortunate that a separate mechanism is needed to deal with access control in the cases of screen rendering and executing actions. However, frameworks typically do not even provide explicit support for this problem to the extent done in Reahl.

Controlling access at an interface facade is also not ideal. In an object-oriented world, for example, it is possible to call one method on an interface, which returns a domain object. And once the object has passed through interface, any method can be called on such object without the protection afforded by the interface. A good measure of programming discipline makes this shortcoming passable in practice, but a better solution is necessary in the long term.

The current solutions in Reahl for these problems are not elegant, and can be cumbersome and error prone. Both the screen generation and the interface access control mechanisms could share security specifications that are specified once—at the relevant domain object. How to accomplish this elegantly is still an unsolved problem. The problem is interesting, since it spans concerns related to presentation and domain model.

Not much effort has been devoted in the Reahl camp to research the issue, save for the experience the author had using some other web frameworks.

The idea of adaptive widgets is an idea not seen in other web frameworks, and they also do not typically have cohesive ways to deal with other access control issues.

3.4 Validation of form fields

Most web frameworks incorporate some mechanism for validating user input. This is also something that ideally should be specified in one central location. When rendering a screen to be displayed to a user, this information is necessary, for example, to be able to indicate which fields of a form are required and which are optional. It may also be desirable to generate some JavaScript code as part of rendering the screen so that some fields can be validated as they are being typed.

Assuming such a form has been rendered, though, the validation information is needed at another stage of the process. When the user enters data and submits the form back to the server, the submitted input values need to be validated again. It may be that a malicious user has circumvented all other ways of validating the input before this state. It could also be that previous validation happened in JavaScript code, but that the user's browser has JavaScript turned off.

Most web frameworks allow one to create some kind of server-side UI element, such as a form, with fields and other elements on it. On this element you can typically specify validation rules, and this element is what is used to generate the initial screen, as well as to parse and validate the values when the form is submitted.

Reahl employs a similar mechanism. However, experience has shown that such a scheme allows for a lot of duplication:

A particular domain object may appear on several different UI forms. A programmer thus needs to duplicate, on each form, the same specification of how the fields of such an object are to be validated. Maintaining this duplication is very cumbersome and error-prone.

Once again, it would be better if a solution can be found where validation could be specified in one central place: the relevant element of the domain model.

4 CATERING FOR A RICHER EXPERIENCE

Traditionally, web applications were limited in that it was not feasable to use much client-side computing power. All control and processing happened at the server, which would send pages to the client user agent merely for display. Any action by the user would result in another round-trip to the server where the system would react, and send back another page for display to the client.

The reason for this limitation is that JavaScript implementations varied to such an extent in different browsers that it was impractical to generate JavaScript code that would work on all browsers. The same was true for several other web standards, such as Cascading style sheets (CSS) and even how different browsers interpreted HTML.

UIs limited in this way are impoverished compared to what users are used to in Graphical user interfaces (GUIs). Users cannot drag and drop; there's no concept of multiple windows; UI elements cannot be animated; every significant user action results in a round-trip to a server—which makes for a slow user experience; etc.

For such reasons, technologies (standardised 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.

Web applications built with these technologies are sometimes colloquially termed Rich Internet applicationss (RIAs).

In the last few years, however, browsers have converged to such an extent that large Internet companies like Google started leading the way towards using client-side code in order to provide a better user experience.

One popular method is called Asynchronous JavaScript and XML (AJAX) [7]. AJAX simply means that JavaScript is used at the client side to fire off events to the server in the background, leaving the client browser ready and responsive. Once the result of such a background request is received back, the client-side JavaScript code can modify the current page in-place.

Changing server round-trips to be asynchronous (and concurrent) gives a user the perception that the user interface is more responsive. Since only parts of a page need to be transferred in such a round-trip, less time is spent on transferring pages to the user.

A JavaScript program running in a client browser could also be used to keep track of the current state of the UI—freeing the central server which would have had to keep track of such information for each of its many concurrent clients. This possibility holds the promise of better scalability.

Using the AJAX technique, however, is problematic from other perspectives.

JavaScript-rich sites are not always traversable by search engines, as explained in [10]. And it is important for most sites that they should be indexed by search engines. Similarly, AJAX taken to the extreme results in a whole site to be written as if it is one single, changing page from the browser's point of view. Not only can it be difficult (to impossible) for search engines to index such a site, but important browser capabilities, such as the ability to bookmark a page, or to use the back button can be thwarted by the AJAX technique [4]. The various security restrictions in JavaScript makes it difficult to implement an AJAX site that caters for all these needs.

AJAX sites are also vulnerable to cross-site-scripting attacks [3].

The asynchronous, concurrent nature of AJAX also result in complex scenarios that need to be dealt with in the client-side JavaScript.

AJAX is currently the only viable way to provide a rich UI on the web which is standards-compliant. But many companies attempt proprietary RIA technologies. Adobe's Flash is the most widespread of these, since its proprietary rendering engine has a tremendously wide installed base [2]. A proprietary company can control their software easier than a standards body can control implementations of a standard. Companies are not hampered by a plethora of incompatible implementations.

Reahl, Harel, and all research that went into it disregarded the RIA side of web-based UI. An important step forward would be to incorporate such techniques into the high-level approach taken by Reahl. Doing so still warrants a lot of research: literature exists on user interface patters for the web [11, 17, 21, 16]. Many different JavaScript libraries have seen the light, each with its own set of capabilities and UI elements. A valuable next step would be to survey all of this literature and libraries, so as to be able to abstract the essence of what UIs can be built. Another taxonomy of such approaches might perhaps allow one to see the wood for the trees in this instance.

With such background one would be able to attempt to integrate these UI patterns and RIA techniques into the current statechart-based model of Reahl. Reahl specifically focussed on web-based UIs that were not "rich". Building a GUI is another matter entirely, since GUIs are much more complex. However, statecharts have already been employed for building GUIs as well [9]. It remains for us to investigate such approaches, and to see how these approaches can be merged with Reahl's current approach, combined with methods such as AJAX.

5 CONCLUSION

The initial ideas behind Reahl were initiated several years ago. In the mean time CMSs have started becoming more and more flexible. They allow more (albeit still limited) control over look and feel, and also expose more of their internals, allowing a programmer to extend the functionality offered. It is tempting to think that a high-level virtual machine for building web interfaces would look rather a lot like a flexible CMS.

In the past few years some web frameworks have also evolved to provide an explicit way of specifying page flow

[12]. At least one even has a graphical tool with which the programmer can visualise and edit such a flow [14].

Changes incorporated into Reahl often resulted from pressure on it, as a web framework, to be able to cater for needs traditionally dealt with using a CMS.

Reahl currently compares well with the converging approaches taken by web frameworks and CMSs. Reahl's method of specifying page flow could be argued to be superiour to those found in other competing frameworks[3]. Reahl also is a framework which now includes some functionality traditionally found in a CMS.

Some solutions in Reahl are still felt by its developers to be unsatisfactory, but the solutions are on a par with their counterpart solutions by other web framework and CMS competitors.

A lot of the hastily added functionality in Reahl does not extend its model as elegantly as the author would have wished for. Furthermore, some CMS-like functionality in Reahl is primitive compared to the counterparts found in a CMS where such ideas originated.

The large-scale adoption of AJAX and other RIA approaches warrant a new look at how such functionality could be incorporated into Reahl and its statechart based approach. This opens up the doors to explore an already existing body of knowledge on using statecharts in more flexible GUI interfaces—a body of knowledge that previously related less well to the web due to the limitations of the web at that time. Many of these limitations can be overcome currently using AJAX or other approaches to RIA on the web.

From this experience, it would seem that the world of web development still has a number of difficult problems to solve. Reahl is ideally positioned to experiment further with such problems and can cater for the audiences of web framework users and CMS users.

REFERENCES

[1] Object management group (OMG). *Unified Modelling Language v1.5*. OMG, March 2003.

[2] Adobe Systems Inc. Adobe Flash and Shockwave Players: Adoption Statistics. `http://www.adobe.com/products/player_census/`, 2008. (last accessed June 2008).

[3] Brian Dillard. Do try this at home: Ajax bookmarking, cross-site scripting, and other web 2.0 browser hacks. In *web 2.0 Expo*, `http://en.oreilly.com/webexsf2008/public/schedule/detail/1186`, April 2008. O'Reilly.

[4] Field Expert. Ajax best practices. `http://www.fieldexpert.com/ajax-best-practices/`, 2008. (last accessed June 2008).

[5] M. Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, 1997.

[6] M. Fowler. Dealing with roles, 1997.

[7] Jesse James Garrett. Ajax: a new approach to web applications. `http://www.adaptivepath.com/ideas/essays/archives/000385.php`, February 2005. (last accessed June 2008).

[8] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[9] Ian Horrocks. *Constructing the User Interface with Statecharts*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[10] Google Inc. A spider's view of web 2.0. `http://googlewebmastercentral.blogspot.com/2007/11/spiders-view-of-web-20.html`, 2007. (last accessed June 2008).

[11] Sari A. Laakso. User interface design patterns. `http://www.cs.helsinki.fi/u/salaakso/patterns/`, 2003. (last accessed June 2008).

[12] Craig McClanahan, Ed Burns, and eds Roger Kitain. *JavaServer^{TM} Faces Specification, v1.1*. Sun Microsystems, Inc., February 2004.

[13] Phillip J. Eby. The quick guide to python eggs. `http://peak.telecommunity.com/DevCenter/PythonEggs`, 2007.

[14] Spring Framework. Spring ide 2.0. `http://www.springframework.org/springide/release-20`, 2007. (last accessed June 2008).

[15] The OSGi Alliance. Osgi alliance specifications. `http://www.osgi.org/Specifications/HomePage`, 2000-2005.

[16] Jenifer Tidwell. *Designing Interfaces: Patterns for Effective Interaction Design*. O'Reilly, 2005.

[17] Anders Toxboe. User interface design pattern library. `http://ui-patterns.com`, 2008. (last accessed June 2008).

[18] Iwan Vosloo. A web application user interface specification language based on statecharts. Master's thesis, University of Pretoria, Pretoria, South Africa, 2005.

[19] Iwan Vosloo and Derrick G. Kourie. Server-centric web frameworks: An overview. *ACM Comput. Surv.*, 40(2), 2008.

[20] Marco Winckler and Philippe Palanque. StateWebCharts: A formal description technique dedicated to navigation modelling of web applications. In *Interactive Systems. Design, Specification, and Verification: 10th International Workshop, DSV-IS 2003, Funchal, Madeira Island, Portugal, June 11-13, 2003. Revised Papers*, volume 2844/2003 of *Lecture Notes in Computer Science*, pages 61–76, GmbH, December 2003. Springer-Verlag.

[21] Yahoo! Inc. Yahoo! Design Pattern Library. `http://developer.yahoo.com/ypatterns/`, 2005-2008. (last accessed June 2008).

[3]The details of this comparison was deemed outside of the scope of this article, though.