Web Applications – Spaghetti Code for the 21st Century

Tommi Mikkonen and Antero Taivalsaari
Sun Microsystems Laboratories
P.O. Box 553 (TUT), FI-33101 Tampere Finland

{tommi.mikkonen, antero.taivalsaari}@sun.com

Abstract

The software industry is currently in the middle of a paradigm shift. Applications are increasingly written for the World Wide Web. Unfortunately, the technologies used for web application development today violate well-known software engineering principles, and they have reintroduced problems that had already been eliminated years ago in the aftermath of the "spaghetti code wars" of the 1970s. In this paper, we investigate web application development from the viewpoint of software engineering principles. We argue that current web technologies are inadequate in supporting many of these principles, but also that there is no fundamental reason for web applications to be any worse than conventional applications in any of these areas. Rather, the current inadequacies are an accidental consequence of the poor conceptual and technological foundation of the web development technologies today.

1. Introduction

The software industry is in the middle of a paradigm shift towards web-based software. Consequently, applications are increasingly written for the web rather than for any specific type of operating system, CPU architecture or device. We believe that the trend towards web-based applications will continue and even strengthen in the future, causing a fundamental change in the way people develop, deploy and use software.

Unfortunately, programming languages and software development environments and tools have not yet adapted to this ongoing shift. Today, software development for the web is based on a myriad of technologies and tools with little consistency or elegance. Even worse, many of the technologies violate established software engineering principles, and they have reintroduced problems that had already been eliminated years ago in the aftermath of the "spaghetti code wars" of the 1970s.

In this paper[1], we investigate web application development from the viewpoint of software engineering principles such as modularity, consistency, and portability. We argue that current web technologies are inadequate in supporting many of these principles, but that there is no fundamental reason for web applications to be any worse than conventional applications in any of these areas. Rather, the current inadequacies are just an accidental consequence of the poor conceptual and technological foundation of the web development technologies today.

The structure of this paper is as follows. Section 2 provides an overview of web development, followed by an introduction to software engineering principles in Section 3. Sections 4 and 5 analyze web development technologies in view of the software engineering principles and some additional topics, and Section 6 offers possible solutions to emerging problems. Finally, Section 7 concludes the paper.

2. Web development – A view from the trenches

The World Wide Web has undergone a number of evolutionary phases. Initially, web pages were little more than simple textual documents with limited user interaction capabilities. Soon, graphics support and form-based data entry were added. Gradually, with the introduction of DHTML [12], it became possible to create increasingly interactive web pages with built-in support for advanced graphics and animation.

Today, we are in the middle of another major evolutionary step towards "Web 2.0" technologies, commonly associated with systems such as Ajax [5],

[1] An earlier version of this paper has been published as Sun Labs Technical Report TR-2007-166, June 2007.

978-0-7695-3302-5/08 $25.00 © 2008 IEEE
DOI 10.1109/SERA.2008.16

Ruby on Rails [35], and Google Web Toolkit (GWT) [34]. However, these technologies are still hybrid solutions and only partial steps in the evolution towards using the web as a true application platform. In general, they enable building web sites that behave much like desktop applications, for example, by allowing web pages to be updated one user interface element at a time, rather than requiring the entire page to be updated each time something changes. In this paper, we refer to such applications as *web applications*.

The fundamental components of the web browser include the Hypertext markup language (HTML), Cascading Style Sheets (CSS), the JavaScript scripting language, and the Document Object Model (DOM). In the following, we provide a summary of these technologies:

1. *HTML* is the predominant markup language for the creation of web pages. It provides a means to describe the structure of text-based information in a document – by denoting certain text as headings, paragraphs, lists, and so on – and to supplement that text with interactive forms, embedded images, and other objects.
2. *CSS* is a stylesheet language that is used to describe the presentational aspects of a document written in a markup language. A stylesheet allows the stylistic rules of a web page (e.g. colors and fonts) to be defined independently of the content of the web page.
3. *JavaScript*. In addition to declarative HTML and CSS definitions, a web document can also contain executable code. The most commonly used scripting language on the web today is JavaScript [10]. Syntactically JavaScript resembles the C and Java programming languages. However, in practice JavaScript is a much more dynamic, interpreted language that borrows features from other highly dynamic languages such as Smalltalk [11], Lisp [20] and Self [36].
4. *DOM* is a platform-independent way of representing a collection of objects that constitute a page in a web browser. The DOM allows a scripting language such as JavaScript to programmatically examine and change the web page. Effectively, the DOM serves as an interface – essentially a large shared data structure – between the browser and the scripting language, allowing the two to communicate with each other.

In practice, web pages that utilize the above four technologies – serving as the logical equivalent of "binary code" of traditional applications – are often messy. Even major web sites like amazon.com, ebay.com, or cnn.com contain large chunks of HTML, CSS and JavaScript code, in which HTML definitions, style sheets and JavaScript functions are interspersed in no specific order, other than the order established by the tools used to generate the web page. Consequently, the source document of such web sites is often nearly unreadable to humans. Furthermore, the document usually contains generous hard-coded references to other similar documents and resources such as images or animations. These resources may be located anywhere in the world, and they are usually included in the document in the form of long Uniform Resource Locators (URLs). Some of these entities do not even need to be available, in which case the browser ignores the references and uses a placeholder instead or overlooks the entire item. This is enabled by the general principle to ignore erroneous or incomplete parts of the web documents.

If the document that represents the web site is commonly difficult to read, the actual behavior of the document is even harder to understand. This is because the typical user interaction model of the web is such that a new web page is generated and sent to the browser each time the user clicks on a link or a button on a page. While adequate for displaying documents, the approach is not ideal for Web 2.0 development, in which the web browser runs *applications* that behave like conventional desktop applications, with rich desktop-style user interface and direct manipulation capabilities beyond navigation of pages.

The above technologies do not introduce a coherent foundation for real applications on the web. Rather, they reflect the evolution of the World Wide Web, in which new features have been added on top of existing features. For instance, the I/O model to generate new web pages on the fly when anything changes seems outright arcane compared to the direct manipulation capabilities provided by most desktop applications in the 1980s and 1990s.

In order to compensate for the inadequacies of the underlying technology, today's web development environments rely extensively on *tool support*. Tools such as Adobe Dreamweaver and Microsoft Expression Web are utilized for enhancing the end-user experience and for providing a more reasonable development experience. This has made web development a lot less arduous. However, it has also exacerbated problems: If programmers had been fully exposed to the gruesome details of web development, they would have undoubtedly questioned some of the technological foundations and put more pressure on improving the underlying technology.

3. Software engineering principles

In 1968, Edsger Dijkstra started his crusade against "spaghetti code" [7]. Spaghetti code is a pejorative term for source code that has a complex and tangled control structure, especially one using many gotos, threads, global variables, or other "unstructured" constructs.

As highlighted by the spaghetti code discussions, software engineering remained an undeveloped, unestablished practice until the late 1970s [9, 14]. Many important principles, such as modularity, information hiding, and portability did not exist or were not adopted widely until they were introduced and instituted by Parnas, Clements, Corbató, Hoare, Liskov and many others in the seminal articles in the 1970s and 1980s [3, 26, 6, 27, 28, 29, 30, 31, 32, 23, 24, 18, 19, 13, 39, 4]. MacLennan has summarized many of these principles in his book that focuses on the principles of programming languages [22].

An important milestone in the codification of software engineering is the 1968 NATO Software Engineering Conference [25]. Besides introducing the idea of reusable software and software components [21], the conference was remarkable in other respects. The attendees of the conference agreed that design concepts essential to maintainable systems are *modularity*, *specification*, and *generality*. In the conference, the first formal definition of software engineering was also provided. As summarized in [25], software engineering was defined as the "establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines." Later, IEEE has provided the following, shorter definition: "Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software" [15].

4. Web application development and software engineering principles

As long as the primary purpose of web development was the creation of web sites consisting of documents, pages and forms, there was little reason to apply established software engineering principles to web development. However, the transition towards Web 2.0 technologies will increase the need to treat web development in the same fashion as software development in general. The current *ad hoc*, document-oriented and tool-driven approach seems insufficient for this. Perhaps the most important observation about Web 2.0 technologies – excluding all the hype – is that these technologies bring the need to apply software engineering principles to web development.

In general, web development in its current form resembles software development in the 1970s before real engineering principles were applied. We argue that there is a need to introduce engineering principles to web development.

Next, we investigate web development in view of software engineering principles. Our analysis is independent of any particular web development technology, but reflects the technological foundations of the web discussed earlier. We do not claim that our analysis would cover all the possible areas, or would cover any single topic exhaustively. Rather, our point is to highlight the fact that web technologies have deficiencies even in some of the basic areas that were reasonably well understood before the advent of web technologies in the 1990s.

4.1. Modularity and related principles

Modularity is the property of computer programs that measures the extent to which programs are composed out of separate parts called modules. *Module* is generally defined as a self-contained component of a system. It has a well-defined interface to other components. Something is *modular* if it includes or uses modules which can be interchanged as units without disassembly of the module. The internal design of a module may be complex, but this is not relevant; once the module exists, it can easily be connected to or disconnected from the system. Modularity principles were introduced in the 1970s and 1980s by Parnas, Clements, Dahl, Guttag, Hoare, Liskov, Zilles and many others [26, 6, 27, 28, 29, 30, 31, 32, 18, 19, 13, 39].

Separation of Concerns. Separation of concerns is a software development principle that implies that different facets of software development should be kept separate. For instance, separation of concerns implies that one should decompose a system into distinct subsystems that overlap in functionality as little as possible; developing individual subsystems, as well as solving subproblems included in them will then be easier [8]. The use of this construct was originally demonstrated in [6, 28].

Current web development systems do not support separation of concerns well for a number of reasons.

1) *Declarative and procedural development style are mixed up*

Web development technologies mix up different development styles. For instance, web sites combine declarative (CSS and HTML) and procedural

(JavaScript) definitions in no particular order. Among other things, this makes it difficult to follow the flow of program control.

2) *User interface component placement, user interface style elements, event declarations and application logic are mixed up*

In addition to mixing different development styles, current web development technologies also mix up application logic with user interface (UI) details such as widget placement, event handler definitions and style declarations. For instance, event handler definitions, that is, how a web page responds to input from the user, are commonly sprinkled in static HTML definitions as well as in JavaScript code fragments. This violates the well-known MVC (Model-View-Controller) design paradigm [17], which promotes a clean separation between the different facets of a program. The lack of such structure in software is known to be detrimental for the long-term maintainability. Fortunately, many web development systems add support for the MVC via tools and libraries.

3) *Dependence on tool support*

Many web development systems are dependent on tools that introduce conventions and practices to facilitate the development of applications. For instance, Ruby on Rails introduces a set of naming conventions that it automatically applies to programs. For example, if there is a class *Person* in the model, the corresponding table in the database is called *people* by default. Because of such conventions, Ruby on Rails is sometimes referred to as "opinionated software." The use of conventions like this makes the semantics of the system inseparable from the tools.

Well-defined (manifest) interfaces. A central aspect of a modular system is the presence of well-defined interfaces that separate the specification of a component from its implementation details. The interfaces protect the rest of the program from a change when the implementation is revisited [28]. Ideally, all the interfaces should be apparent (manifest) in their syntax [22].

Today's web technologies do not provide good support for well-defined interfaces, as summarized below.

1) *No well-defined interfaces between the browser and other components, apart from the DOM*

The DOM acts as the primary interface between the browser and other components that need access to the data and user interface elements displayed in the browser. Effectively, the DOM serves as a large shared data structure – essentially a global variable, which have been generally considered harmful since the early 1970s [38] – between the browser and external components, allowing scripting languages (such as JavaScript) or external plug-ins to communicate with the browser. Unfortunately, the DOM has evolved in a rather *ad hoc* fashion, and there were significant implementation differences between different browsers especially in the early days of the web. Furthermore, the definition of the DOM itself is not very modular, with no clear distinction between specification-level attributes and more implementation-oriented details.

2) *Hard-coded references and other implementation details used openly*

A fundamental role of interfaces is to separate interfaces from implementation details. In web pages today, there are numerous examples to the contrary. For instance, hard-coded references to servers and directory locations are often exposed although they should not be visible outside public interfaces. Unfortunately, on the web today there are no widely available mechanisms for defining client-side interfaces separately from implementation details.

Information hiding. Information hiding is a principle that goes hand in hand with well-defined interfaces. In order to isolate design decisions and implementation-level issues from the external use of a component, the internals of the component must be hidden and represented separately from the interface. This is not the case with the web.

1) *Source code of applications exposed*

A unique characteristic of web applications is that the entire source code of the web pages is usually available to everybody via the browser's 'View Source' function. The availability of source code facilitates the reuse of web content in other contexts, for instance, in the form of *mashups* that combine content from more than one source into an integrated experience. On the other hand, in the absence of proper information hiding mechanisms the internal implementation details of the web site are openly exposed to the world. This makes mashups and other “leveraged” web sites extremely brittle: Even a slight change to the implementation details of a web site can break all the other web sites that depend on its content.

2) *No privacy mechanisms in JavaScript*

The JavaScript language – the most prevalent scripting language supported by web browsers – has limited means for controlling information hiding. For

instance, the currently widely used versions of the JavaScript language do not have support for declaring variables or functions *protected* or *private*. The future versions of JavaScript have been proposed to include such features, but the standardization work has proceeded slowly in recent years.

4.2. Consistency, simplicity, and elegance

Consistency. According to [22], one of the fundamental principles in software development is *consistency*: There should be a minimum number of concepts with simple rules for their combination; things that are similar should also look similar, and different things should look different. Unfortunately, web development systems today are not particularly consistent, as summarized below.

1) *Several ways to perform the same function*

Syntactic consistency of web applications is generally poor. Similar things need not look similar, and different things can be accomplished with nearly identical syntax. For instance, event declarations can be defined either using HTML or JavaScript, and user interface item placement can be performed using declarative or procedural style.

2) *Extensive reliance on side effects*

Current web technologies rely extensively on side effects. For instance, graphics changes are generally performed by tweaking various attributes in the DOM tree. Instead of such an approach, it would be far more consistent to use explicit function calls to communicate between the different components, for instance, when updating the display from JavaScript.

Simplicity and Elegance. One of the desirable qualities in software development is *simplicity*. Corbató concluded in his Turing Award lecture in the 1990s that all complex systems will ultimately fail, and that to have any chance for success at all, it is absolutely necessary to avoid complexity and strive for simplicity and elegance in design [4].

A closely related principle is *elegance*. As Knuth, Dijkstra and Bach have summarized [16, 9, 2], one view of quality in software is aesthetic view, where quality is elegance, an ineffable experience of goodness. Current web systems are hardly simple or elegant, for the following reasons.

1) *Current web applications are unstructured and difficult to read*

Without appropriate tool support current web applications are unstructured and hard to read. The behavior is defined as a combination of declarative HTML and CSS definitions and small procedural scripts sprinkled here and there. Consequently, the control flow of the applications is usually difficult to follow. Thus, web applications are primarily machine-readable rather than human-readable.

2) *Different types of technologies (HTML, JavaScript, CSS, XML) mixed up*

Current web technologies utilize a host of technologies that have been combined together. For instance, Ajax is based on Dynamic HTML (DHTML), augmented with asynchronous networking support and XML protocols. DHTML itself is a collection of technologies – HTML, CSS, JavaScript and DOM – to facilitate the development of interactive and animated web sites. All these technologies are different, and they were not designed to form an elegant, solid foundation for applications. In some ways, it would be appropriate to refer to web technologies as "mashups" also at the implementation level.

4.3. Reusability and portability

Reusability. *Reusability* refers to the ability to use already existing pieces of software in other contexts. In comparison to traditional software, web applications are quite reusable. Given that web sites are usually represented in textual form without any advance compilation, static binding or linking, it is quite easy to combine content from multiple web sites. However, some problems in this area remain.

1) *Elements of reuse are scattered and mixed with the rest of the application*

The lack of well-defined interfaces and the mix-up of technologies and notations are impediments for reuse. Since the different facets of an application are commonly mixed with the application logic, it is often difficult to reuse them in other contexts. However, this is not really a fundamental problem with web technologies. With some careful planning and design by the software developer, it is reasonably easy to keep the different facets separate. In other words, as with software development more broadly, the construction of reusable web software requires careful advance planning and separation of concerns.

2) *Hard-coded references and other implementation details exposed*

Hard-coded references to specific servers, file structures or other implementation or environment

specific details harden the reuse web page elements in different contexts. By avoiding excessive exposure of implementation details, the reusability of web software could be enhanced.

Portability. *Portability* refers to the ease of adapting a program so that it can be run in a host environment that is different from the one for which the program was originally designed. Web applications are generally quite portable compared to traditional software, e.g., because there is much less dependence on specific binary formats or CPU architectures. Still, some problems in this area remain.

1) *There are still significant differences between browsers and browser versions*

While simple web pages usually render themselves in the expected fashion in most browsers, significant differences exist between browsers and their different versions when running more complex content. Many of these problems can be traced back to the incompatibilities of the DOM implementations. With more advanced Web 2.0 content, the problems tend to get worse. For instance, advanced graphics or networking capabilities are not available in all the browsers, and web page update algorithms may behave differently with asynchronous network updates.

2) *Portability of experience is poor*

Few people write raw HTML, CSS or other low-level web content as such. Instead, web development is usually based on toolsets. Experience in creating web applications using one technology does not easily transfer to others. While there are common elements in all the web technologies, the lowest common denominator is still formed by low-level technologies that are seldom written directly. Thus, web development is still an immature area with no established principles to be taught independently of the actual technology. The establishment of such principles would be a prerequisite for a true "web engineering" discipline.

5. Additional challenges

In addition to the issues related to engineering principles, web technologies have further limitations that hinder the creation of real, desktop-style applications. In this section we take a look at some additional topics based on our personal experience in building web applications. We focus especially on two areas: *usability* and *development style*. In addition to these areas, web technologies have known challenges also in other areas such as security and scalability. We have intentionally left out such topics, as that would have extended the scope of our analysis significantly and made this paper considerably longer.

5.1. Usability

The term *usability* refers to the elegance with which the interaction between a human and a computing system is designed. Usability is a complex topic that cannot be measured entirely objectively. Rather, the measure of goodness depends on the context or the intended goal in using a system. Below we list only usability issues that are related to the use of the web browser as a host for "real" applications, as opposed to the conventional use of the web browser to view pages, forms or other "non-applications".

1) *The browser I/O model is poorly suited to desktop-style applications*

In current systems a scripting engine communicates with the browser primarily by modifying DOM attributes that represent the graphics objects on the screen. Alternatively, the scripting engine can construct HTML pages as strings on the fly and then send those strings to the browser with the expectation that the browser will update its screen accordingly. This is clumsy compared to desktop systems in which programs can manipulate the screen directly.

The page-based display update model of the browser is also an impediment to application usability. This model, in which the entire display is updated in response to a successful user-initiated network request, is antiquated. The model is dramatically less interactive than the direct manipulation user interfaces that were in widespread use in desktop computers already in the 1980s. With asynchronous communication, the page-based update model is gradually being replaced with a finer-grained interaction model.

2) *The semantics of many browser features are unsuitable for applications*

The web browser has a number of historical features that have poorly defined semantics for applications running in the browser. Consider the '*reload*', '*stop*', '*back*' and '*forward*' buttons. While such navigational features make sense for viewing documents and forms, these features have unclear semantics for applications that have a complex internal state and highly dynamic interaction with a web server. In addition to predefined browser buttons, many of the predefined browser menu items – especially those that are displayed when right-clicking objects on the screen – are meaningless for

real applications. Therefore, web applications should be able to override such features with application-specific behavior.

5.2. Development style

The power of the World Wide Web stems largely from the absence of static bindings. For instance, when a web site refers to another site or a resource, or when a JavaScript program accesses a function or DOM attribute, references are resolved at runtime without static checking. This dynamic nature enables flexible combination of content from multiple sites and, more generally, for the web to be "alive" and evolve constantly with no central control.

For an application developer, the dynamic nature of the web poses new challenges, causing some fundamental changes in the development style. The style must be based on stepwise refinement [37]; such a style is closer to the "exploratory" programming style used in the context of dynamic programming languages rather than the style of more static, widely used languages like C, C++ or Java.

1) *No transitive closure of program structures is available statically*

Web applications are so dynamic that it is impossible to know statically if all the necessary structures will be available at runtime. While web browsers are designed to be error-tolerant, the absence of the necessary elements can lead to fatal problems that are impossible to detect before execution. Furthermore, with scripting languages such as JavaScript, the application can modify itself on the fly; there is no way to detect all possible errors ahead of execution. Consequently, web applications require more testing to make sure that all the possible behaviors and paths of execution are tested comprehensively.

2) *No support for static verification or type checking*

In the absence of well-defined interfaces and static type checking, the development style used for web application development is rather different from conventional software development. Since there is no way to detect during the development time whether all the necessary components are present or have the expected functionality, applications have to be written and tested piece by piece, rather than by writing tens of thousands of lines of code before execution.

6. Possible solutions

Given the impressive capabilities of World Wide Web and its increasingly ubiquitous presence, it is not surprising that the use of the web has spread to many new areas outside its original intended use. Because of its impressive power, it is easy to overlook the deficiencies of the web in areas that are beyond its intended usage. In some ways, the use of the web as a software application platform resembles the fairy tale about emperor's new clothes [1]: the deficiencies are so blatantly visible that they should be obvious to everybody. Yet the web is so powerful that people are willing to look the other way. We share Les Hatton's concern – although taken slightly out of context [33] – that "computing has proven to be a fashion industry with little or no relationship with engineering." The use of the web as an application platform undermines the work that has been done in the software engineering area in the past thirty years or so.

However, we believe that the problems of the web are not fundamental or irreversible. Even though the conceptual and technological foundations of the web are somewhat shaky – especially with respect to using the web as an application platform – with some concentrated effort many of the problems could be fixed. At Sun Labs, we have designed a new web programming system called the *Sun Labs Lively Kernel* (http://research.sun.com/projects/lively) that provides our proposed solutions to the problems.

Usability issues. The usability issues of the web seem reasonably easy to fix. Basically, in order to support applications with direct manipulation and desktop-style user interaction, the I/O model of the web needs to be enhanced and complemented with capabilities familiar from the world of desktop applications. The problems in this area boil down to two issues: the troublesome browser I/O model and the presence of browser features that are semantically problematic in the context of real applications. These problems could be fixed by adding support for a direct 2D/3D graphics interface that can be used programmatically from JavaScript or other dynamic programming languages. Such a graphics model would allow the developer to bypass declarative user interface definitions in situations in which direct manipulation would be preferred. In fact, many web browsers provide an experimental *Canvas* graphics model supporting such functionality. The Canvas model, when associated with an event model that allows an application to handle its input directly, provides a reasonable framework to reintroduce the best characteristics of desktop applications to the web browser. The *Scalable Vector Graphics* (SVG)

standard supported by many browsers is also a good starting point for a decent programmatic graphics model.

In addition to enhancements in the graphics capabilities of the browser, some other changes should be made. For instance, in order to support desktop-style applications better, it should be possible to disable browser features that make little sense for applications. Furthermore, it should be possible to override the menus of the browser in order to customize the user experience to the needs of the applications. As a possible further enhancement, it should be feasible to run applications in "standalone" model, outside the conventional web browser altogether.

Modularity issues. The modularity problems of web applications are more difficult to solve. After all, the power of the web stems from its openness and the lack of static bindings. Today's web sites are like "glass boxes", with implementation details openly exposed to the world. The glass box approach is beneficial in the sense that it allows the creation of mashups. However, current mashups are often brittle, as there is no standardized way for a web site to publish a summary of its intended external interface, i.e., to provide a description of those features that are intended to be used publicly and possibly reused in other sites.

In principle, there is no reason for such interface description mechanisms not to be available. Ideally, there should be a standard interface description mechanism that would allow each web site to expose only its interface, and provide a summary of features that are intended to be publicly accessible and reusable. The implementation details of the sites should not be visible. At least it should be possible to represent them separately or hide them without resorting to solutions such as obfuscation.

In addition to the solutions proposed above, the programming languages used on the web need to evolve as well. Given that scripting languages such as JavaScript are increasingly used for real programming rather than just scripting, these languages must evolve in a direction in which they provide proper modularity and information hiding mechanisms. Such mechanisms are well understood and have been widely available in other programming languages, and they could be added to JavaScript as well.

Consistency issues. The web has evolved in a rather accidental fashion, without any careful master planning. This has resulted in a set of technologies that have been combined together in a seemingly random fashion. At the same time, the use of the web has rapidly spread into areas that are beyond its original intended usage, including the development and deployment of desktop-style applications.

The consistency problems related to web application development are already partially being solved with tools, higher-level languages and frameworks. Systems such as GWT introduce a higher level of abstraction on top of the base web technologies. The main problems in this area are related to standardization, as it may still be premature to try to get the industry to agree on a common higher-level solution. Until then, web application development will remain "consistently inconsistent."

In terms of programming language choices for web applications, there are a number of candidates available. However, currently there seems to be a lot of convergence towards JavaScript (or more generally, the ECMAScript family of languages). Given its ubiquitous presence in web browsers today, JavaScript is an obvious choice, in spite of its deficiencies in areas such as modularity and information hiding. Even though JavaScript standardization has proceeded slowly in recent years, the language has a lot of potential to evolve in a direction where it would have all the necessary capabilities for real application development. Syntactically, the resemblance of JavaScript to highly popular programming languages such as C, C++ and Java gives it an edge against many other scripting languages.

Reusability and portability issues. Web applications are already quite reusable and portable compared to traditional applications that are delivered in binary form. Still, there is room for improvement. The presence of a proper interface definition mechanism and information hiding capabilities would improve the reusability and portability of web applications. In the absence of such mechanisms, the details of web documents are widely exposed. Furthermore, the creation of reusable, truly portable software requires careful planning and design from the programmer's side.

Development style issues. The development style used for creating web applications is far more dynamic than the style used for developing conventional desktop software. The absence of static, strong type system means that there is less need for time-consuming "edit-compile-link-run-crash-debug" cycles when doing web development. In contrast, the possibility of errors is manyfold, since there is no way to know statically if all the necessary pieces of the web application will be present at runtime, or if the components really have the expected behavior. Even though exploratory programming techniques have been used for years, mainstream software developers are not generally familiar with them. This issue can be solved primarily through education. Static verification tools

(such as 'jslint' for JavaScript) can play an important role in helping analyze the integrity of the application ahead of execution. To some extent, integrated development environments can also help in avoiding obvious mistakes.

7. Summary

In this paper, we have provided a summary of software engineering issues and challenges related to web-based application development. We started with a brief overview of web technologies and software engineering principles, and then analyzed the web technologies in view of the software engineering principles. We argued that there are no fundamental reasons for web applications to be any worse than conventional applications in any of the investigated areas. Finally, we offered some suggestions on how to eliminate the key deficiencies.

9. References

[1] Andersen, H.C., Emperor's New Clothes. Houghton Mifflin, 2004 (the original story published in 1837).

[2] Bach, J., The challenge of "good enough" software. *American Programmer* Vol 8, No 10 (Oct) 1995, pp.2-11.

[3] Corbató, F.J., Sensitive issues in the design of multi-use systems. *Unpublished lecture transcription of February 1968*, Project MAC Memo M-383.

[4] Corbató, F.J., On building systems that will fail. *Communications of the ACM* Vol 34, June 1991, pp.72-81.

[5] Crane, D., Pascarello, E, James, D., *Ajax in Action*. Manning Publications, 2005.

[6] Dahl, O-J., Dijkstra, E.W., Hoare, C.A.R., *Structured Programming*. Academic Press, 1972.

[7] Dijkstra, E.W., Go to statement considered harmful. *Communications of the ACM*, Vol. 11, No 3, March 1968, pp.147-148.

[8] Dijkstra, E.W., On the role of scientific thought, August 1974. Published in *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, 1982, pp.60-66.

[9] Dijkstra, E.W., Programming: from craft to scientific discipline. In Morlet, E., Ribbens, D. (eds), *Proceedings of the International Computing Symposium* (Liege, Belgium, April 4-7, 1977), North Holland Publishing Company, 1977.

[10] Flanagan, D., JavaScript: The Definitive Guide. O'Reilly Media, 2006.

[11] Goldberg, A., Robson, D., Smalltalk-80: the language and its implementation. Addison-Wesley, 1983.

[12] Goodman, D., Dynamic HTML: The Definitive Reference. O'Reilly Media, 2006.

[13] Guttag, J., Abstract data types and the development of data structures. *Communications of the ACM* Vol. 20, No 6 (Jun) 1977, pp.396-404.

[14] Hoare, C.A.R., Programming: sorcery or science? *IEEE Software* Vol. 1, No 2, April 1984, pp.5-16.

[15] *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Std. 610-1990, IEEE Standards Software Engineering, Volume 1, The Institute of Electrical and Electronics Engineers, 1999.

[16] Knuth, D., *The Art of Computer Programming*, Addison-Wesley, 1968.

[17] Krasner, G.E., Pope, S.T., A cookbook for using Model-View-Controller user interface paradigm in Smalltalk-80, *Journal of Object-Oriented Programming*, 26-29, August 1988.

[18] Liskov, B.H., Zilles, S.N., Programming with abstract data types. In Proceedings of *ACM SIGPLAN Conference on Very High Level Languages*, *ACM SIGPLAN Notices* Vol. 9, No 4 (Apr) 1974, pp.50-59.

[19] Liskov, B.H., Zilles, S.N., Specification techniques for data abstractions. *IEEE Transactions on Software Engineering* Vol. SE-1, No 1 (Mar) 1975, pp.7-19.

[20] McCarthy, J., *LISP 1.5 Programmer's Manual* (with Abrahams, Edwards, Hart, and Levin). MIT Press, Cambridge, Massachusetts, 1962.

[21] McIlroy, M.D., Mass produced software components. In [25], pp.88-98.

[22] MacLennan, B.J., *Principles of Programming Languages: Design, Evaluation, and Implementation*, 3rd edition, Oxford University Press, 1999.

[23] Morris, J.H.Jr., Protection in programming languages. *Communications of the ACM* Vol. 16, No 1 (Jan) 1973, pp.15-21.

[24] Morris, J.H.Jr., Types are not sets. In Proceedings of *the ACM Symposium on Principles of Programming Languages* (Boston, Massachusetts, October 1-3), ACM Press, 1973, pp.120-124.

[25] Naur, P., Randell, B. (eds): *Working Conference on Software Engineering* (1968 NATO Conference on Software Engineering, Garmisch, Germany, October 7-11, 1968), January 1969.

[26] Parnas, D.L., *Information distribution aspects of design methodology*, Technical Report, Department of Computer Science, Carnegie-Mellon University, February 1971.

[27] Parnas, D.L., A technique for software module specification with examples. *Communications of the ACM* Vol. 15, No 5 (May) 1972, pp.330-336.

[28] Parnas, D.L., On the criteria to be used in decomposing systems into modules. *Communications of the ACM* Vol. 15, No 12 (Dec) 1972, pp.1053-1058.

[29] Parnas, D.L., On the design and development of program families. *IEEE Transactions on Software Engineering* Vol. SE-2, No 1 (Mar) 1976, pp.1-9.

[30] Parnas, D.L., Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering* Vol. SE-5, No 2 (Mar) 1979, pp.128-137.

[31] Parnas, D.L., Clements, P.C., Weiss, D.M., Enhancing reusability with information hiding. In Proceedings of *the ITT Workshop on Reusability in Programming* (Newport, Rhode Island, September 7-9), 1983, pp.240-247.

[32] Parnas, D.L., Clements, P.C., A rational design process: why and how to fake it. *IEEE Transactions on Software Engineering* Vol. SE-12, No 2 (Feb) 1986, pp.251-256

[33] Paulson, L.D., Developers shift to dynamic programming languages. *IEEE Computer*, February 2007, pp.12-15.

[34] Prabhakar, C., *Google Web Toolkit: GWT Java Ajax Programming*. Packt Publishing, 2007.

[35] Tate, B., *Ruby on Rails: Up and Running*. O'Reilly Media, 2006.

[36] Ungar, D., Smith, R.B., Self: the power of simplicity. In *OOPSLA'87 Conference Proceedings* (Orlando, Florida, October 4-8), ACM SIGPLAN Notices vol 22, nr 12 (Dec) 1987, pp.227-241.

[37] Wirth, N., Program development by stepwise refinement. *Communications of the ACM* Vol. 14, No 4 (Apr) 1971, pp.221-227.

[38] Wulf, W., Shaw, M., Global variable considered harmful. *ACM SIGPLAN Notices* Vol. 8, No 2, February 1973, pp.28-34

[39] Zilles, S.N., Procedural encapsulation: a linguistic protection technique. ACM SIGPLAN Notices vol 8, nr 9 (Sep) 1973, pp.142-146.