

Web Applications - Spaghetti Code for the 21st Century

Tommi Mikkonen and Antero Taivalsaari

Web Applications - Spaghetti Code for the 21st Century

Tommi Mikkonen and Antero Taivalsaari

SMLI TR-2007-166

June 2007

Abstract:

The software industry is currently in the middle of a paradigm shift. Applications are increasingly written for the World Wide Web rather than for any specific type of an operating system, computer or device. Unfortunately, the technologies used for web application development today violate well-known software engineering principles. Furthermore, 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 established software engineering principles. We argue that current web technologies are inadequate in supporting many of these principles. However, we also argue 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.



Sun Labs
16 Network Circle
Menlo Park, CA 94025

email addresses:

tommi.mikkonen@sun.com
antero.taivalsaari@sun.com

© 2007 Sun Microsystems, Inc. All rights reserved. The SML Technical Report Series is published by Sun Microsystems Laboratories, of Sun Microsystems, Inc. Printed in U.S.A.

Unlimited copying without fee is permitted provided that the copies are not made nor distributed for direct commercial advantage, and credit to the source is given. Otherwise, no part of this work covered by copyright hereon may be reproduced in any form or by any means graphic, electronic, or mechanical, including photocopying, recording, taping, or storage in an information retrieval system, without the prior written permission of the copyright owner.

TRADEMARKS

Sun, Sun Microsystems, the Sun logo, Java, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@sun.com>. All technical reports are available online on our website, <http://research.sun.com/techrep/>.

Web Applications – Spaghetti Code for the 21st Century

Tommi Mikkonen Antero Taivalaari
tommi.mikkonen@sun.com antero.taivalaari@sun.com

Sun Microsystems Laboratories
P.O. Box 553 (TUT)
FIN-33101 Tampere, Finland

1. Introduction

The software industry is currently in the middle of a paradigm shift. In the past few years, the World Wide Web has become the *de facto* deployment environment for new software systems. Consequently, applications are increasingly written for the web rather than for any specific type of operating system, computer or device. Examples of such applications include online productivity applications (such as word processors, spreadsheets and calendars), social networking systems, online games, and so on. 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 paradigm 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, we investigate web application development from the viewpoint of software engineering principles such as modularity, consistency, simplicity, reusability and portability. We argue that current web technologies are inadequate in supporting many of these principles. However, we also argue 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. In Sections 2 and 3, we provide an overview of web technologies, followed by a brief introduction to software engineering principles in Section 4. In Sections 5 and 6, we analyze web development technologies in view of the software engineering principles and some additional topics. In Section 7, we offer possible solutions to the issues presented in Sections 5 and 6. Finally, Section 8 provides a summary of the paper.

2. Brief Background of Web Technologies

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 [Goo06] – the

combination of HTML, Cascading Style Sheets (CSS), the JavaScript scripting language, and the Document Object Model (DOM) – 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. Web 2.0 is mostly a marketing term, surrounded by a lot of hype, but there are some underlying trends behind these technologies that will dramatically change the way people will perceive and use the web and software more generally. In particular, Web 2.0 technologies make it possible to build 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 when something changes. Such desktop-style web applications are sometimes referred to as *Rich Internet Applications*, or RIAs. In this paper, we refer to such applications simply as *web applications*, or *web-based applications*.

Web 2.0 is most commonly associated with systems such as Ajax [CPJ05], Ruby on Rails [Tat06], and Google Web Toolkit (GWT) [Pra07]. Ajax can be viewed as a logical extension of DHTML, with the addition of asynchronous networking and XML protocols to allow web page update requests to be processed asynchronously in the background. Ruby on Rails, in contrast, relies extensively on tool support and a specific application framework to create a custom environment that is especially well suited to the creation of database-backed web sites that utilize the Model-View-Controller (MVC) user interface paradigm [KrP88]. In contrast, Google Web Toolkit augments the Java™ programming language with a new application framework and a set of tools that allow GWT applications to be translated into JavaScript so that they can be executed in any web browser. In addition to Ajax, Ruby on Rails, and GWT, there are dozens of other web application environments, but none of them are currently as popular as the three aforementioned systems.

While Ajax, Ruby on Rails and GWT are major improvements over earlier web development technologies, these technologies are still hybrid solutions and only partial steps in the evolution of the web towards true web applications. For instance, Ajax is a combination of existing, previously mostly unrelated technologies rather than a uniform, coherent application development environment or platform. Ruby on Rails and GWT are based on a single language (Ruby and Java, correspondingly), but they still rely extensively on the underlying technologies in the web browser (HTML, CSS and JavaScript). Furthermore, all these web application technologies are built around the historical document-oriented – rather than application-oriented – model of the web. In other words, they still treat web sites primarily as collections of pages rather than real applications.

3. Web Development – A View from the Trenches

Current web applications rely extensively on a number of technologies that are fundamental components of the web browser. These include the HTML markup language, Cascading Style Sheets (CSS), the JavaScript scripting language, and the Document Object Model (DOM). In the following, we provide a brief summary of each of these technologies:

1. *HTML*, short for *HyperText Markup Language*, 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. HTML is written in the form of textual labels (known as tags), created by less-than signs (<) and greater-than signs (>). HTML can also describe, to some degree, the appearance and semantics of a document, and can include embedded scripting language code which can affect the behavior of web browsers and other HTML processors.

2. *Cascading Style Sheets (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 (for example, colors, fonts and layout) to be defined independently of the content of the web page. CSS definitions are commonly included in an HTML document to introduce definitions for the appearance of buttons, fonts and other presentational details of web pages.
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 [Fla06]. 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, Lisp and Self. Even though JavaScript is a full-fledged programming language, in web pages it is typically used only for relatively small-scale scripts in order to animate or otherwise enliven or “decorate” web pages.
4. *Document Object Model (DOM)* is a platform-independent way of representing a collection of objects that constitute a page in a web browser. Among other things, 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 flexibly.

Ultimately, most web applications that run in a regular web browser today utilize the technologies summarized above. That is, even though the actual language that the web developer uses may be different, such as the case with Ruby on Rails and GWT, the web browser ends up seeing and executing the application as a combination of the technologies above. In a way, the technologies above serve as the logical equivalent of “binary code” of traditional software applications. However, unlike with conventional software applications, whose binary code was machine-readable and intended for execution only on a particular hardware architecture, in web applications the “binaries” are intended for execution in any web browser regardless of the underlying hardware or operating system. The web application “binaries” are also – at least to some extent – human-readable, as we will see in more detail below.

In practice, web pages that utilize the four technologies above are usually rather messy. For instance, consider the source document of the Amazon.com main web page, which quite accurately reflects the status quo in “pre-Web 2.0” web development today.¹ An excerpt of HTML code from the Amazon main page is included in Figure 1.

¹ We have chosen Amazon.com for no other reason than that the document source code reflects the usage of the basic web technologies well. For demonstration purposes, we could have equally well chosen examples from any other major web site such as eBay, Google or Yahoo.

```

<form style="margin-bottom:0;" method="get" action="/s/ref=nb_ss_gw/103-5470213-2621406" name="site-search">
<table border="0" cellpadding="0" cellspacing="0" align="center">
<tr>
<td class="searchtitle" width="50" align="center">Search&nbsp;</td>
<td width="100"><select name="url">
<option value="search-alias=aps" selected="selected">Amazon.com</option>
<option value="search-alias=apparel">Apparel</option>
<option value="search-alias=automotive">Automotive</option>
<option value="search-alias=baby-products">Baby</option>
<option value="search-alias=beauty">Beauty</option>
... a lot of similar definitions removed ...
<option value="search-alias=wireless-plans">Wireless Plans</option>
</select></td>
<td width="5">&nbsp;</td>
<td width="325">
<input type="text" id="twotabsearchtextbox" name="field-keywords" value="" size="25" style="width:100%" />
<td>
<td width="5">&nbsp;</td>
<td><input type="image" src="http://g-ec2.images-amazon.com/images/G/01/buttons/go-orange-trans.gif" width="21" alt="Go" value="Go" name="Go"
height="21" border="0" /></td>
</tr>
</table>
</form>

```

Figure 1. Sample HTML code from the main page of Amazon.com

In addition to simple definitions referring to commonly used elements, the HTML document also defines how cascading style sheets (`<style type="text/css">`) and JavaScript (`<script type="text/javascript">`) are used (not shown here). An excerpt of CSS code from the Amazon.com main page is included in Figure 2, while Figure 3 provides an excerpt of JavaScript code. The total size of the document in Amazon.com main page is about 1,300 lines of source text, or <135 kilobytes.

Usually, HTML definitions, style sheets, and JavaScript code are not represented separately as shown here. Rather, in Amazon.com and in most web sites, HTML definitions, style sheets and JavaScript functions are interspersed and mixed in no specific order – other than the order that was established by the tools that were used to generate the web page. Consequently, the source document of most web

```

<style type="text/css"><!--
BODY { font-family: verdana,arial,Helvetica,sans-serif; font-size: x-small; background-color: #FFFFFF; color: #000000; margin-top: 0px; }
TD, TH { font-family: verdana,arial,Helvetica,sans-serif; font-size: x-small; }
a:link { font-family: verdana,arial,Helvetica,sans-serif; color: #003399; }
a:visited { font-family: verdana,arial,Helvetica,sans-serif; color: #996633; }
a:active { font-family: verdana,arial,Helvetica,sans-serif; color: #FF9933; }
... a lot of similar font definitions removed ...
.indent { margin-left: 1em; }
.half { font-size: .5em; }
.list div { margin-bottom: 0.25em; text-decoration: none; }
.hr-center { margin: 15px; border-top-width: 1px; border-right-width: 1px; border-bottom-width: 1px; border-left-width: 1px; border-top-style: dotted; border-right-style: none; border-bottom-style: none; border-left-style: none; border-top-color: #999999; border-right-color: #999999; border-bottom-color: #999999; border-left-color: #999999; }
.horizontal-search { font-weight: bold; font-size: x-small; color: #FFFFFF; font-family: verdana,arial,Helvetica,sans-serif; }
.horizontal-websearch { font-size: xx-small; font-family: verdana,arial,Helvetica,sans-serif; padding-left: 12px; }
.big { font-size: x-large; font-family: verdana,arial,Helvetica,sans-serif; }
.amabot_right .h1 { color: #c60; font-size: .92em; }
.amabot_right .amabot_widget .headline, .amabot_left .amabot_widget .headline { color: #c60; font-size: .92em; display: block; font-weight: bold; }
.amabot_widget .headline { color: #c60; font-size: small; display: block; font-weight: bold; }
.amabot_left .h1 { color: #c60; font-size: .92em; }
.amabot_left .amabot_widget, .amabot_right .amabot_widget, .tigerbox { padding-top: 8px; padding-bottom: 8px; padding-left: 8px; padding-right: 8px; border-bottom: 1px solid #ADD2E2; border-left: 1px solid #ADD2E2; border-right: 1px solid #ADD2E2; border-top: 1px solid #ADD2E2; }
.amabot_center { font-size: 12px; }
... a lot of similar definitions removed ...
.rightArrow { color: #c60; font-weight: bold; padding-right: 6px; }
.nobullet { list-style-type: none }
.homepageTitle { font-size: 28pt; font-family: 'Arial Bold', Arial; font-weight: 800; font-variant: normal; font-style: bold; color: #80B6CE; }
--></style>

```

Figure 2. Sample CSS code from the main page of Amazon.com

```

<script>
function FGSimplePopLocate(oHotspot) {
  var popX = oHotspot.ableft;
  var popY = oHotspot.abstop + oHotspot.height;
  var nHAdj = this.nHAdjust;
  var nVAdj = this.nVAdjust;

  if (nHAdj == 'c') {
    nHAdj = parseInt(oHotspot.width/2 - this.width/2);
  } else if (nHAdj == 'r') {
    nHAdj = oHotspot.width-this.width;
  }
  var x = popX + nHAdj;
  var xWindowLeft = goN2U.getScrollLeft();
  var xWindowRight = xWindowLeft + goN2U.getInsideWindowWidth() - (goN2U.isMozilla5() ? 19 : 0);
  x = Math.min(x, xWindowRight-this.width-4);
  x = Math.max(x, 4, xWindowLeft);
  this._doLocate (x, popY + nVAdj);
}
n2RunThisWhen(n2sRTWTBS,
function() {
  FGSimplePop = new N2SimplePopover();
  goN2Events.registerFeature('findGift', 'FGSimplePop', 'n2MouseOverHotspot', 'n2MouseOutHotspot');
  FGSimplePop.initialize('FGSimplePopDiv', 'FGSimplePop', gaTD, null, 'below', 'c');
  FGSimplePop.locate = FGSimplePopLocate;
},
'init popover' );
function parseName(){
var nameField = document.registrysearch['field-name'];
var typeField = document.registrysearch.type;
var name = nameField.value;
var name_array = name.split(" ");
if (typeField.value != 'wishlist'){
  var lastNameField = document.registrysearch['field-lastname'];
  var firstNameField = document.registrysearch['field-firstname'];
  if (name_array.length == 1 ){
    var lastNameField = document.registrysearch['field-lastname'];
    lastNameField.value = name;
  }
  else if (name_array.length > 1 ){
    lastNameField.value = name_array[name_array.length - 1];
    name = "";
    for (var i = 0; i < name_array.length - 1; i++)
    {
      name = name + " " + name_array[i];
    }
    firstNameField.value = name;
  }
}
}
}
</script>

```

Figure 3. Sample JavaScript code from the main page of Amazon.com

sites is often rather unreadable to humans. Furthermore, the document usually contains generous hard-coded references to other similar documents and resources such as graphical 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). In fact, some of these entities do not even need to be available, in which case the browser simply 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.

As can be seen in the code samples in Figures 1, 2 and 3, the source code of the web documents is not very easy 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. That is, each time the user performs a navigational action on the user interface, the document that the browser displays is generated anew.

As long as the primary purpose of the web browser is to display *documents* in the form of pages or forms, the current approach works reasonably well. However, the approach is far from ideal for Web 2.0 development, where the goal is to use the web browser to run *applications* that behave much like conventional desktop applications, with rich desktop-style user interface and intuitive direct manipulation capabilities beyond simple navigation of pages.

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

The role of tool support. 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 (formerly FrontPage) are utilized for enhancing the end-user experience as well as for providing a more reasonable development experience for the web developer. For instance, instead of editing HTML or CSS definitions manually, the programmers commonly use an integrated development environment to facilitate the task. This has made web development a lot less arduous. However, it has also exacerbated many of the problems. If the 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.

Furthermore, various tool-related complications exist. Currently, tools and techniques available for web development tend to be domain (or vendor) specific, with several systems requiring the pre-installation of additional plug-in components to the browser. Because of differences in the tools and development models, portability of developer experience is very poor. For instance, even though the ultimate execution environment is the same – the web browser – an Ajax developer cannot easily transfer his expertise and become a Ruby on Rails or Google Web Toolkit developer without significant additional training.

Despite the central role of the tools, the generated code and the underlying technologies supported by the web browsers still ultimately determine the semantics of the applications. In general, there is currently no widely accepted theoretical foundation or well-established conceptual model for web development. Rather, web development is based on the collection of technologies summarized earlier, as well as on tools that hide the gory details from the developer.

In the following sections of this paper, we will investigate the foundations of web development from the viewpoint of widely established software engineering principles.

4. Software Engineering Principles

Back in 1968, Edsger Dijkstra started his crusade against “spaghetti code” [Dij68]. Spaghetti code is a pejorative term for source code that has a complex and tangled control structure, especially one using

many gotos, exceptions, threads, global variables, or other "unstructured" constructs. It is named such because program flow tends to look like a bowl of spaghetti – twisted and tangled. It is also used in pejorative sense to imply that a given piece of work is difficult to understand.

As highlighted by the spaghetti code discussions, software engineering remained an undeveloped, unestablished practice until the late 1970s [Dij77, Hoa84]. Many important principles, such as modularity, information hiding, separation of concerns (especially the separation of specification from implementation), manifest interfaces, reusability and portability did not exist or were not adopted widely until they were introduced and instituted by Parnas, Clements, Corbató, Dahl, Guttag, Hoare, Morris, Liskov, Zilles and many others in the seminal articles in the 1970s and 1980s [Cor68, Par71, DDH72, Par72a, Par72b, Par76, Par79, PCW83, PaC86, Mor73a, Mor73b, LiZ74, LiZ75, Gut77, Zil73, Cor91]. MacLennan has summarized many of these principles in his book that focuses on the principles of programming languages [Mac99].

An important milestone in the codification of software engineering is the 1968 NATO Software Engineering Conference [NaR68]. Besides introducing the idea of reusable software and software components [McI68], the conference was remarkable in many other respects. The attendees of the conference agreed that design concepts essential to maintainable systems are *modularity* (to isolate functional elements of the system), *specification* (of the interface as opposed to the implementation; supplying documentation needed to train, understand, and provide maintenance), and *generality* (required for extensibility). In that conference, the first formal definition of software engineering was also provided. As summarized by Bauer in [NaR68], software engineering was then 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” [IEEE99].

5. Web Application Development in View of 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. The web browser, with its original design dating back to 1990, is quite well-suited to displaying documents and supporting simple navigation from page to page.

However, with the current transition towards Web 2.0 technologies, web pages are increasingly taking the form of desktop-style applications, with richer user interface and direct manipulation capabilities, as well as more advanced asynchronous communication protocols between the clients and servers. This 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 to web development seems insufficient in this regard. We crystallize this observation as follows:

Perhaps the most important observation about Web 2.0 technologies – excluding all the hype and marketing hoopla – is that these technologies bring the need to apply rigorous software engineering principles to web development.

In general, web development – in its current form – is reminiscent of software development in the 1970s before real engineering principles were applied to the development of software. The applications that we see on the web today have a complex and tangled structure – just like spaghetti programs in the 1960s and 1970s. We argue that there is a need to introduce sound engineering principles to web development. We refer to the outcome of applying engineering principles to web development as *web engineering* in the same spirit as the application of engineering principles to software development eventually led to software engineering.

In the following subsections, we investigate web application 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 that were discussed earlier. We have divided our investigation into the following categories:

- 1) Modularity and related principles
- 2) Consistency, simplicity and elegance
- 3) Reusability and portability

Each of the categories will be discussed in the subsections below. 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.

5.1 Modularity and Related Principles

Modularity is the property of computer programs that measures the extent to which programs have been composed out of separate parts called modules. *Module* is generally defined to be a self-contained component of a system, which has a well-defined interface to the 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, Gutttag, Hoare, Liskov, Zilles and many others [Par71, DDH72, Par72a, Par72b, Par76, Par79, PCW83, PaC86, LiZ74, LiZ75, Gut77, Zil73].

In general, modularity is a form of abstraction that allows programs to be composed of distinct parts, independently of implementation details. Modularity builds upon a number of important principles such as separation of concerns, well-defined interfaces and information hiding. Below we examine web technologies in view of these principles.

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 [Dij74]. The use of this construct in programming was originally demonstrated by Dahl and Hoare [DDH72] and Parnas [Par72b].

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

Current web development technologies mix up different development styles. For instance, as seen earlier in our Amazon.com example, web sites combine declarative (CSS and HTML) definitions and procedural (JavaScript) definitions liberally 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 [KrP98], which promotes a clean separation between the different facets of a program: the internal application structure and data (= Model), aspects related to the user interface and visualization in general (= View), and the interaction with the user and other external parties (= Control). The lack of such structure in software is known to be detrimental for the long-term maintainability of software. Fortunately, many web development environments such as Ruby on Rails add support for the MVC design paradigm via tools and libraries.

3) Dependence on tool support

Many web development systems today are highly dependent on tools that introduce specific 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, which has been a point of contention for many critics.

Well-defined (manifest) interfaces. A central aspect of a modular system is the presence of well-defined interfaces. Well-defined interfaces separate the specification of a component from its implementation details, allowing the implementation to be changed as necessary without impacting the external use of the component. The interfaces protect the rest of the program from a change when a design decision and the corresponding implementation is revisited [Par72b]. Ideally, all the interfaces should be apparent (manifest) in their syntax [Mac99].

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

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

As already discussed earlier, the Document Object Model (DOM) acts as the primary interface between the browser and the 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 between the browser and external components, allowing scripting languages (such as JavaScript) or external plugins 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 are used openly*

A fundamental role of interfaces is to separate interfaces (specification) from implementation details. In web pages today, there are numerous examples to the contrary, for instance, in the form of hard-coded references to specific servers, directory locations or other implementation details that are exposed openly. Such details should not be visible outside the public interfaces. Unfortunately, on the web today there are no widely available mechanisms for defining interfaces separately from implementation details, apart from using PHP scripts and other server-side technologies to hide the server-side details from the external users.

Information hiding. *Information hiding* is a principle that goes hand in hand with well-defined interfaces. Basically, 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 preferably represented separately from the interface. This is not the case with the web technologies today.

1) *DOM tree is exposed and manipulated through side effects*

In current web systems, the DOM tree forms the center of cooperation between the browser and associated components such as the JavaScript scripting engine. For instance, whenever a script wants to modify the visual aspects of a web page, it accesses the DOM tree to locate and change the corresponding attributes in the DOM tree. The browser will then observe the changes and re-render the screen accordingly. In essence, communication happens in the form of *side effects*: the external components (such as the scripting engine) tweak the DOM tree, and – as a side effect – the browser will pick up the changes and update its display. Effectively, the DOM tree behaves as a shared data structure or a large tree of global variables. In software engineering, the use of such global variables (and side effects in general) has been discouraged for a long time; the first paper arguing against the use of global variables was published by Wulf and Shaw in the early 1970s [WuS73].

2) *Source code of applications is exposed*

A unique characteristic of web applications compared to traditional desktop software 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 is generally useful, as it facilitates the reuse of web content in other contexts, for instance, in the form of *mashups*. In web terminology, a *mashup* is a web site that combines content from more than one source into an integrated experience. However, the availability of source code can also be problematic: 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 easily break all the other web sites that leverage its content. In order to reduce such troubles (and to protect their intellectual property from reverse engineering), many web sites publish their source code in obfuscated form.

3) *No privacy mechanisms available in JavaScript*

The JavaScript language – the most prevalent scripting language supported by web browsers – has limited means for controlling information hiding. For instance, unlike the Java programming language, 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 the JavaScript language have been proposed to include such features, but JavaScript standardization has proceeded rather slowly in recent years,

5.2 Consistency, Simplicity and Elegance

Consistency. According to MacLennan, 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 [Mac99]. Unfortunately, web development systems today are not particularly consistent, as summarized below.

1) *There are several ways to perform the same function*

Syntactic consistency of web applications is generally poor. Similar things do not necessarily look similar at all, and different things can be accomplished with nearly identical syntax. For instance, event declarations for web applications can be defined either using HTML or JavaScript. Similarly, user interface item placement or style definition can be performed using either declarative or procedural style. Examples of such characteristics abound, reflecting the mostly accidental historical evolution of the web, rather than a carefully planned, consistent approach.

2) *Things should happen explicitly rather than through side effects*

As already discussed above, current web technologies rely extensively on the use of side effects, effectively using the DOM tree as a large shared data structure. 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 browser 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 early 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 [Cor91]. Or, as C.A.R Hoare has eloquently put it, "There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies."

A closely related principle is *elegance*. As Knuth, Dijkstra and Bach have summarized [Knu68, Dij77, Bac95], one view of quality in software development is *aesthetic view*, where quality is elegance, an ineffable experience of goodness.

It is difficult to argue that current web systems would be simple or elegant.

1) Current web applications are unstructured and hard to read

As we have discussed earlier, without appropriate tool support current web applications are unstructured and hard to read. Basically, the behavior of the application 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, and is hardly any cleaner than control flow in the dreaded spaghetti programs of the 1960s and 1970s. In general, web applications are primarily machine-readable rather than human-readable, and their internal structure (or lack thereof) reflects primarily the functionality of the browser rather than any well-established software engineering principles.

2) Different types of technologies (HTML, JavaScript, CSS, XML) are 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 rather different, and they were not originally designed together to form an elegant, solid foundation for web applications. In some ways, it would be appropriate to refer to web technologies as “mashups” also at the implementation level – after all, these technologies have been mixed up together in a seemingly *ad hoc* fashion.

5.3 Reusability and Portability

Reusability. An important quality of good software is *reusability*. Reusability refers to the ability to use already existing pieces of software in other contexts. In comparison to traditional software technologies, web applications are actually 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 different technologies and notations are impediments for reuse. Since the different facets of an application, such as items related to user interface placement or style, are commonly mixed with the rest of the application logic, it is often difficult to reuse those elements 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 would be reasonably easy to keep the different facets separate. In general, there is no free lunch in software reuse. 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 are exposed

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

specific details make it difficult to reuse web page elements in different contexts. However, as the idea of content mashups becomes more popular, people will become increasingly aware of problems associated with exposing such implementation details. By avoiding excessive exposure of implementation details, the reusability of web software could be enhanced substantially.

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. Generally speaking, web applications are quite portable compared to traditional binary software. After all, one of the fundamental assumptions behind web technologies is that the same web pages are viewable in any web browser, independently of the underlying computer architecture or operating system. However, there are still some problems in this area.

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

Web pages should generally be viewable in all the browsers and different browser versions. While simple web pages usually render themselves in the expected fashion in most browsers, there are still significant differences between browsers and their different versions especially 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, the more advanced graphics or networking capabilities are not necessarily available in all the browsers, and web page update algorithms may behave differently when running content with asynchronous network updates.

2) Portability of experience is poor

Very few people write raw HTML, CSS or other low-level web content manually. Instead, web development is usually based on a specific toolset such as DreamWeaver or Expression Web. Experience in creating web applications using one technology does not easily transfer to other technologies. For instance, experience in building Ruby on Rails applications is not particularly helpful for Ajax or GWT development. Conversely, an Ajax developer cannot be expected to become a Ruby on Rails developer overnight. While there are common elements in all the web technologies, the lowest common denominator is formed by technologies such as HTML, CSS and JavaScript that most developers do not use directly. In general, web development is still an immature area with no widely established general principles that could be taught independently of the actual technology. The establishment of such principles would be a prerequisite for a true “web engineering” discipline.

6. Additional Challenges in Web Application Development

In addition to the issues related to fundamental software engineering principles, web technologies have further limitations and challenges 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.

6.1 Usability

The term *usability* refers to the elegance and clarity with which the interaction between a human user 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 certain system. Below we list only such 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, documents or other “non-applications”. Web browsers are known to have many other usability issues, but they are out of scope for this paper.

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

In current web technologies a scripting engine communicates with the browser primarily via the DOM, by modifying the 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. Such a model is clumsy compared to traditional desktop systems in which a programming language can usually manipulate the screen directly by using a graphics library supporting direct drawing and direct manipulation.

The page-based display update model of the web browser is also an impediment to application usability. The page-based model – where the entire display is updated in response to a successful user-initiated network request – is outright antiquated and reminiscent of the I/O model of the IBM 3270 series terminals from the 1970s. Such a model is dramatically less interactive than the direct manipulation user interfaces that were in widespread use in desktop computers already in the 1980s. With Ajax and other Web 2.0 technologies supporting asynchronous network communication, the page-based update model is gradually being replaced with a finer-grained interaction model, but it is still hard to implement user interaction capabilities that would be on a par with desktop applications.

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, for instance. 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. For example, it is difficult to define meaningful semantics for an online stock trading or a banking application's response to the 'reload' or 'back' button while processing a financial transaction. The presence of such features can be outright dangerous if a web-based application is used for controlling a medical system or a nuclear plant.

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 applications. Web applications should preferably be able to override such features with application-specific behavior.

6.2 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 such as a bitmap image, or when a JavaScript program accesses a certain function or DOM attribute, the references are resolved at runtime without static checking. It is this dynamic nature that makes it possible to flexibly combine content from multiple web sites and, more generally, for the web to be “alive” and evolve constantly with no central planning or control.

For an application developer, the extreme dynamic nature of the web poses new challenges, causing some fundamental changes in the development style. Basically, the development style needs to be based on stepwise refinement [Wir71]. Such a style is closer to the “exploratory” programming style used in the context of dynamic programming languages such as Lisp [McC62], Smalltalk [GoR83] or Self [UnS87] rather than the style used with more static, widely used languages such as C, C++ or Java. This is a complex topic that would offer enough material for a separate paper or a book. Here we will only touch the surface of the topics related to development style.

1) No transitive closure of program structures is available statically

Web applications are generally so dynamic that it is impossible to know statically if all the structures that the program depends on will be available at runtime. While web systems are designed to be error-tolerant and will ignore incomplete or missing elements, in some cases 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 even modify itself on the fly; there is no way to detect the possible errors resulting from such modifications ahead of execution. Consequently, web applications require significantly more testing (especially coverage testing) to make sure that all the possible application behaviors and paths of execution are tested comprehensively.

2) There is no support for static verification or static 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 ahead of the first execution. Such piecemeal, stepwise development style is similar to the style used with programming languages that are specifically geared towards exploratory programming. Examples of such languages include Smalltalk and Self.

7. Possible Solutions to the Deficiencies of the Web as an Application Platform

The World Wide Web is the most powerful medium for information sharing and distribution in the history of humankind. Given its impressive capabilities 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, including the distribution of photographs, music, videos and so on. More recently, the web has emerged as the *de facto* target platform for software applications as well.

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 [And37] – the deficiencies are so blatantly visible that they should be obvious to everybody. Yet the web is so powerful and popular that people are willing to look the other way; after all, the web has already taken such an encompassing role in our lives. In general, we share Les Hatton's concern – although taken slightly out of context [Pau07] – 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 – we believe that with some concentrated effort many of the problems could be fixed. Below we provide a brief summary of our proposed solutions. We will start with the usability issues (which we believe are the easiest to fix) and then summarize the proposed solutions in areas such as modularity, consistency, and other areas.

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. As we summarized earlier in Subsection 6.1, the problems in this area boil down to two issues: the troublesome browser I/O model and the presence of some browser features that are semantically problematic in the context of real applications.

The problems summarized in Section 6.1 could be fixed fairly easily by adding support for a direct 2D and/or 3D graphics interface that can be used programmatically from JavaScript or other dynamic programming languages. Such a graphics model would allow the web developer to bypass HTML and other declarative user interface definitions in situations in which desktop-style user interaction and direct manipulation would be preferred. In fact, many web browsers already provide an experimental “Canvas” graphics model supporting such functionality (see http://en.wikipedia.org/wiki/Canvas_%28HTML_element%29). 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. Additional standardization work is still required before such capabilities will be widely available in all the web browsers, though.

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 those browser features that make little sense for applications, such as the 'back', 'forward', 'reload' and 'stop' buttons. 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. Such standalone applications could run in the same fashion as traditional desktop applications, except that these applications are downloaded in a portable format from the web rather than executed as binaries from the local disk.

Modularity issues. The modularity problems of web applications are somewhat more difficult to solve. After all, the power of the web stems largely from its openness and the lack of static bindings. However, the absence of static bindings *does not necessarily have to imply the lack of abstractions or well-defined interfaces*. Today's web sites are very much like “white boxes”, or worse yet, “glass boxes”, with their implementation details openly exposed to the world. The glass box approach is beneficial in the sense that it allows the creation of mashups – web sites that combine content flexibly from more than one source into an integrated experience. However, currently mashups are extremely brittle, for there is no standardized way for a web site to publish a summary of its intended external interface, that is, to provide a description of those features that are truly intended to be used publicly and possibly reused in the context of other sites. The current approach, which allows the separation of specification from implementation details only through obscurity or obfuscation, is simply not good enough.

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 publicly expose only its interface, and provide a concise summary of those features that are intended to be publicly accessible and reusable to other sites. In contrast, the implementation details of the sites should not be visible at all, or 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 already in the past decades. We see no reason why they could not be added to JavaScript as well.

Consistency issues. One of the characteristics of the web today is its inconsistency. 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 way 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 like Ruby on Rails and GWT introduce a higher level of abstraction on top of the base web technologies such as HTML, CSS and JavaScript. 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 a rather 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, we think that JavaScript has a lot of potential to evolve in a direction where it would have all the necessary features and 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 other scripting languages such as Perl, PHP, Python or Ruby.

Reusability and portability issues. Web applications are already quite reusable and portable compared to traditional desktop applications that are delivered in binary form. After all, web content is by nature designed to be independent of any specific operating system, computer architecture or device. Nevertheless, there is a lot of work left in this area. The presence of a proper interface definition mechanism and information hiding capabilities would improve the reusability and portability of web applications significantly. In the absence of such mechanisms, the implementation details of web documents are exposed too widely. Furthermore, the creation of reusable, truly portable software requires careful planning and design from the programmer's side. As we noted earlier, there is no free lunch in this area. Rather, education is necessary in order to increase the programmers' awareness of the factors contributing to reuse and portability.

Development style issues. The development style used for creating web applications is far more dynamic than the style used for developing conventional desktop software. In many ways, web application development resembles the exploratory programming style used with dynamic programming languages such as Lisp, Smalltalk or Self. The absence of a 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 'jshint' for JavaScript) can play an important role in analyzing the integrity of the application ahead of execution. To some extent, integrated development environments can also be helpful in guiding the programmer through the most treacherous waters and avoiding the most obvious mistakes.

8. Summary

In spite of all the criticism that we have presented in this paper, we are avid supporters of web-based application development. We believe that the trend towards web-based applications is real, and will continue and even strengthen in the future. This will cause a paradigm shift in the way people develop and use software, in the same fashion as the web has already transformed the sharing and distribution of documents, books, photos, music, videos and so many other areas. The long-term implications of this paradigm shift will be at least as significant as the dramatic transformations that are currently taking place in the entertainment and publishing industries.

The transition towards web-based applications means that the web browser will become the primary target platform for software applications, displacing conventional operating systems and specific computing architectures and platforms from the central role that they used to have. As a consequence, software developers will increasingly write software for the web rather than for a specific operating system such as Linux or Windows, or specific hardware architecture such as x86 or the ARM.

As we have argued in this paper, the evolution of the web has been rather accidental and has not resulted in an ideal platform for applications. The web was initially created for displaying documents and pages, and its capabilities for supporting real applications with rich user interaction capabilities and desktop-style direct manipulation are still limited. Furthermore, the underlying technologies in the web browser – especially HTML, CSS, JavaScript and the DOM – were not designed for the development and deployment of serious software applications. The use of such technologies for web applications has led to a situation in which widely established software engineering principles, such as modularity, consistency, simplicity, reusability and portability, have been overlooked.

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 and issues.

In conclusion, we are excited by the rapid emergence of the web as a platform for real applications. Web-based applications will open up entirely new possibilities in software development, and will ideally combine the best of both worlds: the excellent usability of conventional desktop applications and the enormous potential of the World Wide Web. While current web application technologies are still limited and in conflict with many established software engineering principles, we are confident that over time these issues will be resolved.

References

- And37 Andersen, H.C., *Emperor's New Clothes*. Houghton Mifflin, 2004 (the original story published in 1837).
- Bac95 Bach, J., The challenge of "good enough" software. *American Programmer* vol 8, nr 10 (Oct) 1995, pp.2-11.
- Cor68 Corbató, F.J., Sensitive issues in the design of multi-use systems. Unpublished lecture transcription of February 1968, Project MAC Memo M-383.
- Cor91 Corbató, F.J., On building systems that will fail. *Communications of the ACM* vol 34, June 1991, pp.72-81.
- CPJ05 Crane, D., Pascarello, E, James, D., *Ajax in Action*. Manning Publications, 2005.
- DDH72 Dahl, O-J., Dijkstra, E.W., Hoare, C.A.R., *Structured Programming*. Academic Press, 1972.
- Dij68 Dijkstra, E.W., Go to statement considered harmful. *Communications of the ACM*, vol. 11, no. 3, March 1968, pp.147-148.
- Dij74 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.
- Dij77 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.
- Fla06 Flanagan, D., *JavaScript: The Definitive Guide*. O'Reilly Media, 2006.

- GoR83 Goldberg, A., Robson, D., Smalltalk-80: the language and its implementation. Addison-Wesley, 1983.
- Goo06 Goodman, D., *Dynamic HTML: The Definitive Reference*. O'Reilly Media, 2006.
- Gut77 Guttag, J., Abstract data types and the development of data structures. *Communications of the ACM* vol 20, nr 6 (Jun) 1977, pp.396-404.
- Hoas84 Hoare, C.A.R., Programming: sorcery or science? *IEEE Software* vol 1, nr 2, April 1984, pp.5-16.
- IEEE99 *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.
- Knu68 Knuth, D., *The Art of Computer Programming*, Addison-Wesley, 1968.
- KrP88 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.
- LiZ74 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, nr 4 (Apr) 1974, pp.50-59.
- LiZ75 Liskov, B.H., Zilles, S.N., Specification techniques for data abstractions. *IEEE Transactions on Software Engineering* vol SE-1, nr 1 (Mar) 1975, pp.7-19. Also in *Proceedings of the 1975 International Conference on Reliable Software* (Los Angeles, California, April 21-24), ACM SIGPLAN Notices vol 10, nr 6 (Jun) 1975, pp.72-87.
- McC62 McCarthy, J., *LISP 1.5 Programmer's Manual* (with Abrahams, Edwards, Hart, and Levin). MIT Press, Cambridge, Massachusetts, 1962.
- McI68 McIlroy, M.D., Mass produced software components. In [NaR68] pp.88-98.
- Mac99 MacLennan, B.J., *Principles of Programming Languages: Design, Evaluation, and Implementation*, 3rd edition, Oxford University Press, 1999.
- Mor73a Morris, J.H.Jr., Protection in programming languages. *Communications of the ACM* vol 16, nr 1 (Jan) 1973, pp.15-21.
- Mor73b 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.
- NaR68 Naur, P., Randell, B. (eds): *Working Conference on Software Engineering* (1968 NATO Conference on Software Engineering, Garmisch, Germany, October 7-11, 1968), January 1969.
- Par71 Parnas, D.L., Information distribution aspects of design methodology, Technical Report, Department of Computer Science, Carnegie-Mellon University, February 1971.
- Par72a Parnas, D.L., A technique for software module specification with examples. *Communications of the ACM* vol 15, nr 5 (May) 1972, pp.330-336.
- Par72b Parnas, D.L., On the criteria to be used in decomposing systems into modules. *Communications of the ACM* vol 15, nr 12 (Dec) 1972, pp.1053-1058.
- Par76 Parnas, D.L., On the design and development of program families. *IEEE Transactions on Software Engineering* vol SE-2, nr 1 (Mar) 1976, pp.1-9.

- Par79 Parnas, D.L., Designing software for ease of extension and contraction. IEEE Transactions on Software Engineering vol SE-5, nr 2 (Mar) 1979, pp.128-137.
- PCW83 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.
- PaC86 Parnas, D.L., Clements, P.C., A rational design process: why and how to fake it. IEEE Transactions on Software Engineering vol SE-12, nr 2 (Feb) 1986, pp.251-256
- Pau07 Paulson, L.D., Developers shift to dynamic programming languages. IEEE Computer, February 2007, pp.12-15.
- Pra07 Prabhakar, C., *Google Web Toolkit: GWT Java Ajax Programming*. Packt Publishing, 2007.
- Tat06 Tate, B., *Ruby on Rails: Up and Running*. O'Reilly Media, 2006.
- UnS87 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.
- Wir71 Wirth, N., Program development by stepwise refinement. Communications of the ACM vol 14, nr 4 (Apr) 1971, pp.221-227.
- WuS73 Wulf, W., Shaw, M., Global variable considered harmful. ACM SIGPLAN Notices vol 8, nr 2, February 1973, pp.28-34
- Zil73 Zilles, S.N., Procedural encapsulation: a linguistic protection technique. ACM SIGPLAN Notices vol 8, nr 9 (Sep) 1973, pp.142-146.

About the Authors

Dr. Tommi Mikkonen is a Visiting Professor at Sun Labs, and a professor of computer science at Tampere University of Technology, Finland. Tommi is a well-known expert in the area of software engineering. He has arranged numerous courses on software engineering and mobile computing, and he just recently published a book on mobile software development with Wiley & sons.

Dr. Antero Taivalsaari is a Principal Investigator and Senior Staff Engineer at Sun Labs. Antero is best known for his work on the Java™ Platform, Micro Edition (Java ME). Before returning back to Sun Labs in August 2006, Antero spent seven years working in Sun's Java Software organization, witnessing the evolution of the Java ME platform from a two-person research project to one of the most popular commercial software platforms in the world, with over a billion Java ME devices worldwide. Antero has received Sun's Chairman's Award twice (in 2000 and 2003) for his work on Java ME technology.