Web Browser as an Application Platform: The Lively Kernel Experience

Antero Taivalsaari, Tommi Mikkonen,
Dan Ingalls, Krzysztof Palacz

Web Browser as an Application Platform: The Lively Kernel Experience

Antero Taivalsaari, Tommi Mikkonen,
Dan Ingalls, Krzysztof Palacz

SMLI TR-2008-175 January 2008

Abstract:

For better or worse, the web browser has become a widely-used target platform for software applications. Desktop-style applications such as word processors, spreadsheets, calendars, games and instant messaging systems that were written earlier for specific operating systems, CPU architectures or devices are now written for the World Wide Web, to be used from a web browser by anyone, anywhere, anytime.

The original design of the web browser dates back to the early 1990s. Given that the web browser was originally targeted at displaying static, page-structured documents, it is not surprising that the web browser is not an ideal execution environment for desktop-style applications. In this paper we summarize our experiences in using the web browser as a target platform for real applications. As a concrete example, we use the Sun™ Labs Lively Kernel, a system that pushes the limits of the web browser by implementing a highly interactive web programming environment that runs in a web browser without installation or plug-in components. Based on this work, we analyze the limitations, challenges and opportunities related to the web browser as an application platform. We also provide recommendations for possible future improvements.

Sun Labs
16 Network Circle
Menlo Park, CA 94025

email addresses:
antero.taivalsaari@sun.com
tommi.mikkonen@sun.com
dan.ingalls@sun.com
krzysztof.palacz@sun.com

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 Browser as an Application Platform: The Lively Kernel Experience

Antero Taivalsaari
Tommi Mikkonen

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

Dan Ingalls
Krzysztof Palacz

Sun Microsystems Laboratories
16 Network Circle, MPK16
Menlo Park, CA 94025, U.S.A.

1. Introduction

The widespread adoption of the World Wide Web has fundamentally changed the landscape of software development. In the past few years, the Web has become the *de facto* deployment environment for new software systems and applications. We believe that in the near future the vast majority of new software applications will be written for the Web, instead of conventional target platforms such as specific operating systems, CPU architectures or devices.

In general, the software industry is currently experiencing a *paradigm shift* towards web-based software. In the new era of web-based software, applications live on the Web as services. They consist of data, code and other resources that can be located anywhere in the world. Furthermore, they require no installation or manual upgrades. Ideally, applications should also support user collaboration, i.e., allow multiple users to interact and share the same applications and data over the Internet.

In the era of web-based software, the web browser will take an ever more encompassing, central role in our lives. Among other things, the web browser will take over many roles that conventional operating systems used to have, e.g., in serving as a launchpad for applications, as well as serving as a host platform for applications when they are run. In the eyes of the average computer user, the web browser will effectively be the *de facto* operating system.

In this paper we summarize our experiences in using the web browser as a platform for real, desktop-style applications. As a concrete example, we use the *Sun Labs Lively Kernel* (see http://research.sun.com/projects/lively) – a system that pushes the limits of the web browser by implementing a highly interactive web programming environment that runs in a web browser without installation or any plug-in components whatsoever. Based on this work, we analyze the limitations, challenges and opportunities related to the web browser and web applications more generally. We also provide a number of recommendations for future improvements.

The structure of this paper is as follows. In Section 2, we provide a historical summary of the evolution of the Web, focusing especially on the ongoing transition from web pages towards web applications. We also provide a brief summary of the systems that are currently available for web application development. In Section 3, we provide an overview of the Sun Labs Lively Kernel – a flexible web programming environment designed at Sun Labs. In Section 4, we summarize our experiences in using the web browser as an application platform, taking a look at the various issues that we have discovered. In Section 5, we provide suggestions for future improvement. Finally, Section 6 provides a brief summary of the paper.

2. From Web Pages Towards Web Applications

The World Wide Web has become such an integral part of our lives that it is sometimes difficult to remember that the it did not even exist twenty years ago. The original design documents related to the World Wide Web date back to the late 1980s. The first web browser prototype for the NeXT computer was completed by Tim Berners-Lee in December 1990. The first version of the Mosaic web browser was made available publicly in February 1993, and the first commercially successful browser – Netscape Navigator – was released to the public in late 1994. Widespread commercial use of the Web did not take off until the late 1990s.

In about 10-15 years, the Web has transformed our lives in numerous ways. These days, everyday artifacts and services such as documents, photos, music, videos and newspapers are widely available on the Web. Online banking and stock trading have become commonplace. Various documents that used to be difficult to access, such as municipal zoning documents, government budget documents or tax records, are now readily available on the Web. For most people the web browser has become the most commonly needed – and often the only – computer program that they use. Various industries such as banking, financial services, electronics retailing and music distribution have undergone dramatic transformations as a consequence.

Compared to how dramatically web usage has increased since the 1990s, it is remarkable how little the web browser has changed since it was introduced. For instance, the common navigation features, such as the "*back*", "*forward*" and "*reload*" buttons of the browser, were present already in the early versions of Mosaic and Netscape Navigator. In contrast, the way the Web is used has evolved constantly from the early days. In the subsection below, we provide a brief summary of the evolution of web usage.

2.1 Evolution of Web Usage

The World Wide Web has undergone a number of evolutionary phases. Initially, web pages were simple textual documents with limited user interaction capabilities based on hyperlinks. 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. Numerous plug-in components – such as Flash, RealPlayer and Shockwave – were then introduced to make it possible to build web pages with visually rich, interactive multimedia content.

At the high level, the evolution of web pages can be presented in three phases or generations (see Figure 1):

1) Simple, "classic" web pages with text and static images only
2) Animated multimedia pages with plug-ins
3) Rich internet applications

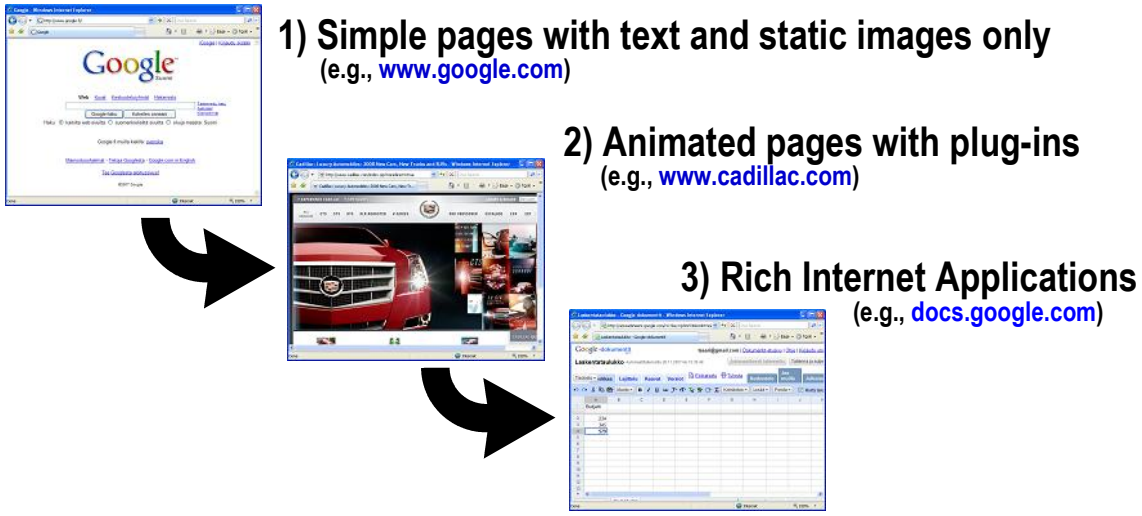

Figure 1: Evolution of web pages

In the first phase, web pages were truly *pages*, i.e., page-structured documents that contained primarily text with some interspersed static images, without animation or any interactive content. Navigation between pages was based simply on hyperlinks, and a new web page was loaded from the web server each time the user clicked on a link. There was no need for asynchronous network communication or any more advanced communication protocols between the browser and the web server. Some pages were presented as *forms*, with simple textual fields and the possibility to use basic widgets such as buttons, radio buttons or pull-down menus. These types of web pages and forms are still very common, characterized by visually simple web sites such as the main page of the Google search engine (http://www.google.com/).

In the second phase, web pages became increasingly interactive, with animated graphics and plug-in components that allowed richer content to be displayed. This phase coincided with the commercial takeoff of the Web, when companies realized that they could create commercially valuable web sites by displaying advertisements or by selling merchandise or services over the Web. Navigation was no longer based solely on links, and communication between the browser and the server became increasingly advanced. The JavaScript scripting language, introduced in Netscape Navigator version 2.0B in December 1995, made it possible to build animated, interactive content more easily. The use of plug-in components such as Flash, Quicktime, RealPlayer and Shockwave spread rapidly, allowing advanced animations, movie clips and audio tracks to be inserted in web pages. In this phase, the Web started moving in directions that were unforeseen by its designers, with web sites behaving more like multimedia presentations rather than conventional pages. Content mashups[1] and web site cross-linking became increasingly popular. Web sites such as yahoo.com or youtube.com are good examples of the second phase in web evolution. Car manufacturers such as Cadillac (http://www.cadillac.com/) or Toyota (http://www.toyota.com/) also utilize such technologies to create highly visual, multimedia-rich vehicle presentations on the Web.

Today, we are in the middle of another major evolutionary step towards desktop-style web applications, also known as *Rich Internet Applications* (RIAs) or simply as *web applications*. The technologies

1 In web terminology, a *mashup* is a web site that combines content from more than one source (from multiple web sites) into an integrated experience.

intended for the creation of such applications are also often referred to collectively as "*Web 2.0*" technologies. Web 2.0 is mostly a marketing term, surrounded by a lot of hype, but the underlying trends are dramatically changing the way people will perceive and use the Web and software more generally.

In short, Web 2.0 technologies combine two important characteristics or features: *collaboration* and *interaction*. By *collaboration*, we refer to the "social" aspects that allow a vast number of people to collaborate and share the same data, applications and services over the Web. However, an equally important, but publicly less noted aspect of Web 2.0 technologies is *interaction*. 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 something changes. Web 2.0 systems often eschew link-based navigation and utilize direct manipulation techniques familiar from desktop-style applications instead. Perhaps the best known example of such a system today is Google Docs (http://docs.google.com) – a web-based office productivity suite that provides support for desktop-style word processing and spreadsheet use over the Web.

Note that the three phases discussed above are not mutually exclusive. Rather, web pages representing all three phases coexist on the Web today. The majority of commercial web pages today represent the second phase. However, the trend towards web applications is becoming increasingly common, with new web application development technologies and systems being introduced frequently. In the next subsection, we take a quick look at the technologies that are used for web application development today.

2.2 Technologies for Web Application Development

In the past few years, numerous web application development technologies have been introduced. From the commercial viewpoint, the following systems seem most promising at the moment:

- Ajax
- Adobe Integrated Runtime (AIR)
- Google Web Toolkit (GWT)
- Google Gears
- JavaFX
- Microsoft Silverlight
- Ruby on Rails

Ajax [CPJ05] can be viewed as a logical extension of Dynamic HTML (DHTML), with the addition of asynchronous HTTP networking and XML protocols to allow web page update requests to be processed asynchronously in the background. The main strength of Ajax compared to many other web application development technologies is that Ajax utilizes only such technologies that already exist in commercial web browsers today; therefore, Ajax requires no plug-ins or any other additional components for application execution.

Adobe Integrated Runtime (AIR) [FPM08] is a web application platform built around Adobe's earlier web technologies, including Flash, ActionScript scripting language [Moo07], and Flex Builder

development environment. AIR applications are written primarily in ActionScript, a language that can be viewed as a variant or an extension of JavaScript. Adobe AIR currently requires a separate runtime or plug-in component for application execution.

Google Web Toolkit (GWT) [HaT07] is a web programming environment built around the Java programming language. Google Web Toolkit augments the Java™ programming language with a widget set and a set of tools that allow GWT applications to be translated into JavaScript. Since every commercial web browser includes a JavaScript engine, GWT applications can be executed in a regular web browser without plug-ins or other additional components.

Google Gears (http://gears.google.com/) is a JavaScript library that extends the core JavaScript libraries [Fla06] in several important areas, especially local storage and database support. Unlike the other web application development technologies mentioned here, Google Gears is not a complete solution in itself. Rather, Google Gears complements other systems such as GWT or Ajax by providing a flexible storage solution that can be utilized from the other technologies.

JavaFX™ [Wea07] is a suite of web application development technologies developed by Sun Microsystems. Java FX is built around Sun's existing technologies such as the Java programming language and the Swing user interface library. Java FX includes a declarative scripting language called *Java FX Script* that makes it easier to define web user interfaces. Like the Java system in general, Java FX requires the Java runtime environment (JRE) plug-in in order to run in a web browser.

Microsoft Silverlight is Microsoft's response to Adobe AIR and other web application development technologies. Silverlight defines a web-based subset of Windows Presentation Foundation (WPF), making it possible to define Flash-style web applications using the same APIs as in .NET applications written for Microsoft Windows. The system is built around JavaScript and Microsoft's XAML (eXtensible Application Markup Language), and it has been closely integrated with Microsoft's Visual Studio development environment. Like many other technologies, Silverlight currently requires a separate plug-in for application execution in a web browser.

Ruby on Rails [Tat06] is a web development environment built around the Ruby programming language [FlM08]. Ruby on Rails 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]. Like many other web application environments, Ruby on Rails requires a special runtime component to run in a web browser.

In addition to the systems mentioned above, there are dozens of other web application development systems. Most of them have little chance to ever enjoy major commercial success. However, a few other systems are worth mentioning. In the past year or so, social networking systems such as *MySpace* (http://www.myspace.com/) and *Facebook* (http://www.facebook.com/) have become wildly popular. As a consequence, the application development capabilities built into these systems have also become well known and very widely used. For instance, Facebook has its own application description language that can be used for creating web applications for Facebook pages. Even though these systems are not full-fledged, general-purpose web application development technologies such as those mentioned earlier, the huge popularity of Facebook makes it an interesting contender in the web application area. For many people, Facebook may well be the first web application environment that they ever use.

2.3 General Observations and Trends

In analyzing the web application development technologies mentioned above, it quickly becomes obvious that all these technologies are still rather different from each other. In fact, one might half-jokingly say that the only common aspect between these systems is that they are all different. However, there are some common themes that have started to emerge:

- *Trend toward dynamic languages* (see [Pau07]). Most of the systems above rely on dynamic, interpreted languages at least at some level. In some systems, such as Ajax or Ruby on Rails, applications are written entirely in a dynamic language (JavaScript and Ruby, respectively.) Other systems, such as Google Web Toolkit, depend on a dynamic language (JavaScript) for program execution inside the web browser.

- *Technology mashups*. Most of the systems mentioned above are hybrid solutions in the sense that they combine various existing, sometimes previously unrelated technologies. In doing so, they resemble the content mashups that are common on the Web today; hence we use the term *technology mashup*. For instance, Ajax is a combination of a number of existing technologies – HTML, CSS, DOM, JavaScript, asynchronous HTTP networking and XML protocols – rather than a uniform, coherent application platform.

- *Dependence on tools*. Most of the systems mentioned above are heavily dependent on tools and integrated development environments. For instance, Ruby on Rails introduces a set of naming conventions that are automatically applied by the development tools. AIR and Silverlight developers are also heavily assisted by integrated development environments.

Another general observation about web application development today is that they often violate well-known software engineering principles. For instance, the JavaScript language has very limited support for modularity or information hiding. We have summarized our observations in this area in more detail in another paper [MiT07a].

3. Sun Labs Lively Kernel

At Sun Labs, we have developed a new web programming environment called the *Sun Labs Lively Kernel*. The Lively Kernel supports desktop-style applications with rich graphics and direct manipulation capabilities, but without the installation or upgrade hassles that conventional desktop applications have. The system and the applications written for it run in a regular web browser without installation or plug-in components. The system even includes development tools that can be used inside the system itself.

For the purposes of this paper, the Lively Kernel is especially interesting because the system pushes the limits of the web browser as an application platform further than any other system. In addition to supporting desktop-style applications that can run in a web browser, the Lively Kernel can also function as an integrated development environment (IDE), making the whole system self-supporting and able to improve and extend itself dynamically. Yet the entire system requires nothing more for its execution than a web browser.

3.1 Motivation and Goals

The Lively Kernel is built around the following three assumptions/observations:

1) The World Wide Web is the New Software Platform
2) The Web Browser is the New Operating System
3) JavaScript is the *de facto* Programming Language of the Web

The World Wide Web is the most powerful medium for information sharing and distribution in the history of humankind. As such, 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.

We believe that the massive popularity of the Web will make it the *de facto* platform for software applications as well. Consequently, the web browser will take over various roles that the conventional operating systems used to have. For the average computer user, the web browser will effectively be the operating system; after all, most of the applications and services that they need will be available on the Web. We believe that the increasing importance of the browser will, in turn, make the JavaScript language the *de facto* programming language on the Web. After all, JavaScript (also known by the name *ECMAScript* [ECM99]) is the only language that is supported by all the commercial web browsers. As we have argued in another paper [MiT07b], the use of JavaScript is not necessarily limited to scripting; rather, it can be used as a real programming language as well.

In designing the Lively Kernel, we have followed the following four goals:

1) No installation or upgrades
2) Support for direct manipulation and desktop-style user experience
3) Uniform and consistent development experience
4) Malleability and support for dynamic changes throughout the system

In general, the key goal of the Lively Kernel is to support desktop-style applications with rich graphics and direct manipulation capabilities, but without the installation or upgrade hassles that conventional desktop applications have.

A key difference between the Lively Kernel and other systems in the same area is our focus on *uniformity*. Our goal is to build a platform using a minimum number of underlying technologies. This is in contrast with many current web technologies that utilize a diverse array of technologies such as HTML, CSS, DOM, JavaScript, PHP, XML, and so on. In the Lively Kernel we attempt to do as much as possible using a single technology: JavaScript. We have chosen JavaScript primarily because of its ubiquitous availability in the web browsers today and because of its syntactic similarity to other highly popular languages such as C, C++ and Java. However, we also want to leverage the dynamic aspects of JavaScript, especially the ability to modify applications at runtime. Such capabilities are an essential ingredient in building a malleable web programming environment that allows applications to be developed interactively and collaboratively.

3.2 Overview and Main Features

In short, the Sun Labs Lively Kernel is a truly interactive, "zero-installation" web application development environment that has been written entirely in JavaScript. It runs in an ordinary web browser without installation or plug-in components; it supports desktop-style applications with rich user interface features and direct manipulation capabilities; it enables application development and deployment in a web browser with no installation or upgrades, using nothing more than existing web technologies. In addition to its application execution capabilities, the Lively Kernel can also function as an integrated development environment (IDE), making the whole system self-supporting and able to improve and extend itself dynamically. A screen snapshot of the Lively Kernel is provided in Figure 2.

Figure 2: The Lively Kernel running in the Safari browser

The Lively Kernel system consists of the following four components:

- *The JavaScript programming language*. We have used the JavaScript programming language as a fundamental building block for the rest of the system. A JavaScript engine is available in all the commercial web browsers today.

- *Asynchronous HTTP networking*. All the networking operations in the Lively Kernel are performed asynchronously using the *XMLHttpRequest* feature familiar from Ajax. The use of asynchronous networking is critical so that all the networking requests can be performed in the background without impairing the interactive response of the system. Support for asynchronous HTTP networking is available in all the commercial web browsers today.

- *Morphic user interface framework and widgets*. The Lively Kernel is built around a rich user interface framework called *Morphic*. The Morphic framework consists of about 10,000 lines of uncompressed JavaScript code that is downloaded to the web browser when the Lively Kernel starts. The Morphic framework is described in more detail below.

- *Built-in tools for developing, modifying and deploying applications on the fly*. The Morphic UI framework includes tools (such as a class browser and object inspector; see Figure 3) that can be used for developing, modifying and deploying applications from within the Lively Kernel system itself. These features have been implemented using the reflective capabilities of the JavaScript programming language, and can therefore be used inside the web browser without any external tools or IDEs. The features make it possible, e.g., to write new JavaScript classes, modify or delete existing methods, or change the values of any attribute in the system. Finally, it is possible to export objects or entire web pages, so that the applications written inside the Lively Kernel can also be run as standalone web pages.

A unique characteristics of the Lively Kernel is a user interface framework called *Morphic.* Morphic supports composable graphical objects, along with the machinery required to display and animate these objects, handle user inputs, and manage underlying system resources such as displays, fonts and color maps. Sophisticated built-in mechanisms are provided for object scaling, rotation, object style editing, as well as for defining user interface themes. The central goal of Morphic is to make it easy to construct and edit interactive graphical objects, both by direct manipulation and from within programs. The Morphic user interface was originally developed for Self [Mal95, MaS95], and it became popular later as part of Squeak [IKM97].

Like the original Morphic environment for Self and Squeak, the JavaScript implementation of Morphic supports collaborative editing. This feature is not fully functional yet, but the intent is that objects in the Lively Kernel can be shared and manipulated simultaneously by multiple users over the Internet.

3.3 Pushing the Limits of the Web Browser as an Application Platform

The Lively Kernel presents web pages as containers of live objects that can be manipulated directly. Applications are grouped into *worlds*, each of which can contain a number of applications and widgets. Compared to conventional web pages, which are relatively static, the pages in the Lively Kernel are active, accessible, changeable; in short, lively.

For the purposes of this paper, the Lively Kernel is especially interesting because it demonstrates the ability to run a self-supporting, truly visual and interactive application development environment on top of a small kernel that has been written entirely in JavaScript. By *self-supporting* we mean that in the Lively Kernel it is possible to "live inside" the web browser and perform all the necessary object creation and manipulation operations using the JavaScript language and the Lively Kernel itself. The system does not require anything other than an available web browser. At the implementation level, the Lively Kernel relies on the underlying graphics capabilities of the browser. A browser with Scalable Vector Graphics (SVG) support, such as Apple's *Safari* browser or Mozilla's *Firefox*, is currently required.

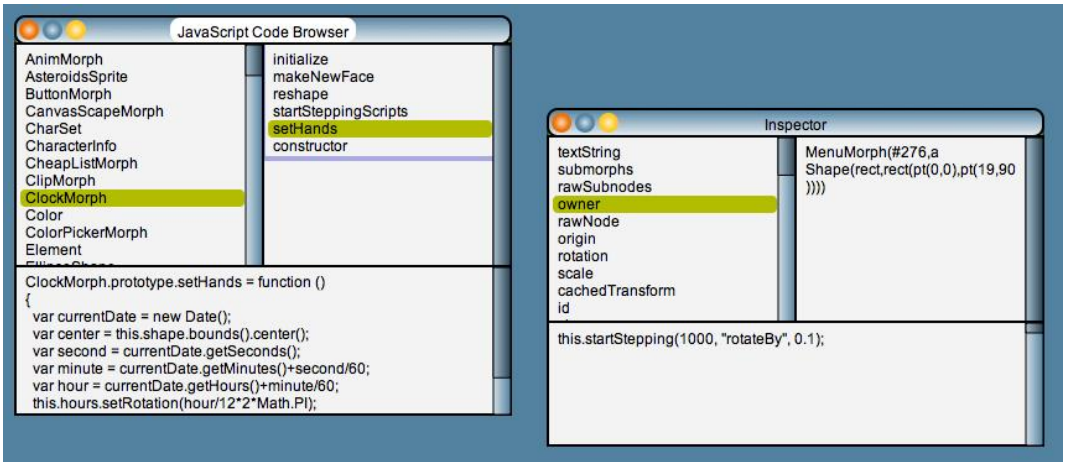

Figure 3: Class browser and object inspector in the Lively Kernel

Compared to other Rich Internet Application development systems, the Lively Kernel goes several steps further in the sense that it allows software development to occur inside the web browser (see Figure 3). No external tools are required, although external tools (such as a conventional source code editor or an IDE) can certainly be used if the programmer prefers a more conventional development style. In contrast with many other web application development systems today, the Lively Kernel also allows multiple applications to be run simultaneously. In some ways, the system illustrates that the entire computer desktop, including all the commonly used tools and applications, can be moved to the Web and run in a web browser as well.

Given these additional capabilities, and the fact that the system can run in an ordinary web browser, the Lively Kernel is an ideal research vehicle for studying the capabilities of the web browser as an application platform. The Lively Kernel can also serve as a testbed for studying the use of the JavaScript language as a general-purpose programming language (see [MiT07b]). Some of the work that we have done with JavaScript could even be described as systems programming. In general, the Lively Kernel proves that there is a lot of latent, unleashed power in the web browser; for instance, the true potential of JavaScript as a dynamic, reflective programming language inside the web browser has not been fully utilized so far.

4. Experiences, Observations and Lessons Learned

In this section we provide a summary of our experiences with the web browser as a host platform for applications. We have categorized our experiences and observations as follows.

1) Usability and user interaction issues
2) Networking and security issues
3) Browser interoperability and compatibility issues
4) Development style and testing issues
5) Deployment issues
6) Performance issues

4.1 Usability and User Interaction Issues

A lot of our frustration with web applications today arises from the document-oriented nature of the Web. On the Web today, applications are second class citizens that are relegated to a side role amongst markup languages, style sheets, and other vestiges of the document-oriented approach that dominates the Web today. In general, it is still clear that application support on the Web is an afterthought rather than a carefully designed, integral feature. This is especially true when it comes to usability and user interaction. In this subsection we take a look at a number of issues in this area. Many of these comments are familiar from our earlier paper [MiT07a].

The browser I/O model is poorly suited to desktop-style applications. In current web technologies a scripting language and other external components communicate with the web browser primarily via the Document Object Model (DOM). DOM is effectively a large tree data structure that allows external tools to inspect and manipulate the data that is displayed in the browser, typically by reading and modifying the DOM attributes that represent the graphical objects on the screen. In addition to tweaking the DOM tree, an external tool such as a scripting language can construct HTML pages as strings (containing HTML markup) on the fly and then send those strings to the browser with the expectation that the browser will update its screen accordingly. This is very cumbersome compared to traditional desktop systems in which a programming language can usually manipulate the screen directly by using a well-defined graphics API that supports 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 – in which 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.

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. Consider the '*reload*', '*stop*', '*back*' and '*forward*' buttons, for instance. While such navigational features make sense when viewing documents and forms, these features have unclear semantics for applications that have a complex internal state and highly dynamic interaction with the 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 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.

4.2 Networking and Security Issues

The document-oriented history of the web browser is apparent also when analyzing the restrictions and limitations that web browsers have in the area of networking and security. Many of these limitations

date back to conventions that were established early on during the design of the browser. Some of the restrictions are “folklore” and have never been fully documented or standardized.

The “Same Origin” networking policy is problematic. A central security-related limitation in the web browser is the “same origin policy” that was introduced originally in Netscape Navigator version 2.0. The philosophy behind the same origin policy is simple: it is not safe to trust content loaded from arbitrary web sites. When a document containing a script is downloaded from a certain web site, the script is allowed to access resources only from the same web site but not from other sites. In other words, the same origin policy prevents a document or script loaded from one web site ("origin") from getting or setting properties of a document from a different origin.

The same origin policy has a number of implications for web application developers. For instance, a web application loaded from a web site cannot easily go and access data from other sites. This makes it difficult to build (and deploy) web applications that combine content (e.g., news, weather data, stock quotes) from multiple web sites. Special proxy arrangements are usually needed on the server side to allow networking requests to be passed on to external sites. Consequently, when deploying web applications, the application developer must be closely affiliated with the owner of the web server in order to make the arrangements for accessing the necessary sites from the application.

Only a limited number of simultaneous network requests allowed. Early on in the history of the Web another design convention was established that prevents a web browser from creating too many simultaneous HTTP requests. Such limitations were introduced to prevent too much web traffic from being created. Even today, most web browsers allow only a limited number (e.g., two or four) of network requests to be created simultaneously. With highly interactive web applications that require a lot of data from several sites asynchronously, such as the Lively Kernel, such limitations can cause problems, especially if some of the sites do not respond to requests as quickly as expected.

No access to local resources or host platform capabilities. Web documents and scripts are usually run in a sandbox that places various restrictions on what resources and host platform capabilities the web browser can access. For instance, access to local files on the machine in which the web browser is being run is not allowed, apart from reading and writing a limited amount of data in cookies. Among other things, the sandbox security limitations prevent a malicious web site from altering the local files on the user's local disk, or from uploading files from the user's machine to another location.

Unfortunately, the sandbox security limitations of the web browser make it difficult to build web applications that utilize local resources (e.g., the local file system) or other host platform capabilities. Consequently, it has been nearly impossible to write web applications that would, e.g., be usable also in offline mode without an active network connection. These problems are gradually being solved with the introduction of libraries such as WebDAV [Dus03] and Google Gears (http://gears.google.com/). For the Lively Kernel, we have implemented a storage solution using WebDAV.

A more fine-grained security model is missing. The key point in all the limitations related to networking and security is that there is a need for a more fine-grained security model for web applications. On the Web today, applications are second-class citizens that are on the mercy of the classic, “one size fits all” sandbox security model of the web browser. This means that decisions about security are determined primarily by the site (origin) from which the application is loaded, and not by the specific needs of the application itself. Even though some interesting proposals have been made

[YUM07], currently there is no commonly accepted finer-grained security model for web applications or for the Web more generally.

4.3 Browser Interoperability and Compatibility Issues

Incompatible browser implementations. A central problem in web application development today is browser incompatibility. Commercial web browsers have incompatibilities in various areas. For instance, the DOM implementations vary from one browser to another. DOM attribute names can vary from browser to browser; even seemingly trivial attributes such as window width and height have different names in different browsers. The JavaScript implementations have known differences, e.g., in the area of how event handlers can be triggered programmatically. The graphics libraries supported by the browsers have also been implemented differently. All these differences make it difficult to implement cross-platform, cross-browser web applications that would run identically on all browsers.

Disregard for official standards. Some browser vendors have a tendency to favor their own technologies in lieu of official World Wide Web Consortium (W3C) or ECMA standards. For instance, the JavaScript graphics libraries in the Lively Kernel depend on the W3C Scalable Vector Graphics (SVG) standard. Unfortunately, SVG support is not yet available in one of the commercially most important web browsers.

Lack of standards for important areas such as advanced networking, graphics or media. The Java programming language has exceptionally rich class libraries that have been standardized over the years using the Java Community Process℠ (http://www.jcp.org/). In contrast, during the development of the Lively Kernel we noticed that JavaScript libraries available for web application development are still surprisingly immature and incomplete. No widely accepted standards exist for important areas such as advanced networking, 2D/3D/vector graphics, audio, video and other advanced media capabilities. Even though such libraries have been defined as part of external JavaScript library development activities such as Dojo (http://www.dojotoolkit.org/), no officially accepted W3C or ECMA standards for these areas exist yet.

4.4 Development and Testing Issues

As we have argued in an earlier paper [MiT07a], 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. The dynamic nature of the Web has various implications for application development and testing.

Evolutionary, stepwise development style is needed. 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" or evolutionary 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.

Completeness of applications is difficult to determine. Web applications are generally so dynamic that it is impossible to know statically, ahead of application execution, if all the structures that the program depends on will be available at runtime. While web browsers are designed to be error-tolerant and will ignore incomplete or missing elements, in some cases the absence of 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, and 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.

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 (e.g., Smalltalk or Self).

Incremental testing is required. Due to the highly permissive, error-tolerant nature of JavaScript, JavaScript programming requires an incremental, evolutionary approach to testing as well. Since errors are reported much later than usual, by the time an error is reported it is often surprisingly difficult to pinpoint the original location of the error. Error detection is made harder by the dynamic nature of JavaScript, for instance, by the possibility to change some of the system features on the fly. Furthermore, in the absence of strong, static typing, it is quite possible to execute a program and only at runtime realize that some parts of the program are not yet present. For all these reasons, the best way to write JavaScript programs is to proceed step by step, by writing (and immediately testing) each new piece of code. If such an incremental, evolutionary approach is not used, debugging and testing can become quite tedious even for relatively small JavaScript applications. In general, the programming style required by JavaScript is closer to the exploratory programming style used in the context of other dynamic programming languages such as Smalltalk or Self.

Code coverage testing is important. The dynamic, interactive nature of JavaScript makes testing deceptively easy. In the presence of an interactive command shell and the '*eval*' function, each piece of code can be run immediately after it has been written. However, the use of such immediate testing approach does not guarantee the program to be bug-free or complete. In a static programming language, many simple errors will be caught already during the compilation of the program. However, in a dynamic language, it is not possible to know statically if a piece of code that has never been executed will actually run without problems. Since programs may contain numerous rarely executed branches – for instance, exception handlers – *code coverage testing* is very important. Code coverage measures to which degree the source code of a program has been tested. Code coverage is commonly measured as a percentage of the source code that has undergone execution and testing. Any dynamic program with less than 100% code coverage testing can still contain undiscovered problems. Even with 100% code coverage, it is still possible that further problems will be found.

4.5 Deployment Issues

The World Wide Web is an exceptionally powerful medium for sharing and distribution. Basically, anything that is made available on the Web is instantly accessible by anybody anywhere in the world. We believe that this "instant deployment" capability, when applied to software applications, will fundamentally transform the software industry in the next 5-10 years, causing a major disruption and eventually removing most of the middlemen from software distribution.

This disruption will have interesting implications. For instance, there will no longer be any need to do "shrink-wrapped" software releases. Even more importantly, the need for manual application installation or upgrades will go away. Ideally, the user will simply point the web browser to a site containing an application, and the latest version of the application will start running automatically. Release cycles will become considerably shorter. All these changes will be significant improvements compared to the traditional way of deploying desktop software. There are still some issues associated with such a deployment model, though.

Applications are "always on". With the instant deployment model, applications are downloaded directly from the Web. The applications released on the Web are "always on" in the sense that all the changes made to them will be immediately visible to all those users who subsequently download the application. Since many of the users may still be using an earlier version, any updates to the application will have to be made carefully. For instance, if the application's internal data formats on a web server database change, backwards compatibility must be taken into account, since there may still be thousands of users who are using an older version of the application.

Towards "nano-releases". A software release is the distribution of an initial or new and upgraded version of a computer software product. Traditionally, new software releases have occurred relatively infrequently, perhaps a few times per year for a major software application such as a word processor or spreadsheet application, or a few times per month for some business-critical applications in early stages of their deployment cycle. The instant deployment model will change all this, allowing new releases to be made much more frequently. In the ultimate scenario, a new release occurs each time changes are made to the system, perhaps even several times a minute. The possibility of such "*nano-releases*" has not been investigated much so far, but is bound to have significant long-term impacts in the software industry.

Perpetual beta syndrome. The transition towards web applications will make releases deceptively simple. When combined with the use of dynamic languages that allow incomplete software to be run (see Section 4.4), it becomes dramatically easier to release software in early stages of its development. This will lead to what we call the "*perpetual beta syndrome*": many software applications will never reach a point when they are actually ready for prime-time use. Only those applications and web sites that will become adequately popular will ever reach maturity. To paraphrase one of Troutman's Laws of Computer Programming, "If a program is useful, it will have to be changed." In contrast, those web applications that will not become popular enough, will simply wither away and stay in beta form perpetually.

Fragmentation problems. The instant deployment model is closely related to the compatibility issues discussed earlier in Section 4.3. The instant deployment, "zero-installation" model works smoothly only as long as the target platform – in this case the web browser – is identical for all the users. If

different browser versions or additional plug-in components are required, application distribution becomes considerably more challenging. From the application developer's viewpoint, this results in *fragmentation*: the need to build multiple versions of the same application for different platform variants. Such fragmentation problems are familiar from the mobile software industry, in which there are hundreds or thousands of different target devices (mobile phones), each with its own characteristics and peculiarities.

4.6 Performance Issues

Until recently, the performance of web pages was not much of a problem. If there were any performance problems, they were more commonly associated with network latency and other connectivity issues, rather than with the performance of any particular components inside the web browser or on the web page itself. However, now that people have started running real applications on the Web, performance problems have become increasingly apparent.

Inadequate JavaScript performance. Current JavaScript virtual machines are unnecessarily slow. Even though JavaScript is a significantly more dynamic language than, for instance, the Java programming language, there is no fundamental reason for JavaScript programs to run 10-100 times slower than comparable Java applications on the Java virtual machine. At the very minimum, JavaScript Virtual Machine (VM) performance should be comparable to optimized Smalltalk VM implementations. This is not yet the case. Fortunately, a number of higher-performance JavaScript VMs are on their way, including Mozilla's *Tamarin* virtual machine (http://www.mozilla.org/projects/tamarin/).

Inadequate memory management capabilities. Current JavaScript virtual machines have simple, 1970's style garbage collectors and memory management algorithms that are poorly suited to large, long-running applications. For instance, with large applications that allocate tens of megabytes of memory, garbage collection pauses in Mozilla's SpiderMonkey JavaScript VM can be excessively long. Occasionally such pauses can take tens of seconds even on a fast machine. As in the VM performance area, with modern virtual machine implementation techniques memory management behavior could be improved substantially.

Inadequate graphics library performance. Application performance is typically a combination of many factors. The performance of the underlying execution engine, such as a JavaScript virtual machine, is in itself insufficient to guarantee the optimal performance of the application. Based on our experience, a major performance bottleneck in today's web browsers is graphics library performance. Graphics engines, such as the engines available for Scalable Vector Graphics (SVG), can be surprisingly slow. For highly interactive environments such as the Lively Kernel, this can have a significant negative impact on performance.

Inefficient bindings between the browser and other components. A great deal of the performance problems in the web application area can be attributed to inefficient communication between the browser and various other components, e.g., native graphics libraries and networking libraries. For instance, when the coordinates of a graphical object are passed from a JavaScript application to the browser (DOM) and ultimately to a native graphics library that actually draws the object, it is very common to convert the numeric parameters into strings and then back to numbers again, possibly several times during the process. Such conversions can easily slow down graphics performance by an

order of magnitude or more. In general, a lot of room for optimization remains in the area of improving the native communication interfaces between the web browser, JavaScript engine and graphics libraries.

5. Solutions and Recommendations

Compared to how dramatically web usage has increased since the early 1990s, it is remarkable how little the web browser has changed since it was originally introduced. In general, web browsers are already so widely established that it may seem rather difficult to try to make any significant changes in the design or the behavior of the browser. However, given how quickly the use of web applications is increasing, it is quite possible that web browsers will have to adapt to accommodate a more application-oriented approach, in addition to the document-oriented approach that dominates the Web today. In this section we provide a number of possible solutions and recommendations in this area.

Solving the usability and user interaction issues. The usability issues of the web browser seem relatively easy to fix. Basically, in order to support applications with direct manipulation and desktop-style user interaction, the I/O model of the web browser needs to be enhanced and complemented with capabilities familiar from the world of desktop applications. As we summarized earlier in Section 4.1, the problems in this area boil down to two basic issues: (1) the cumbersome I/O model of the web browser, and (2) the presence of some browser features that are semantically problematic in the context of real applications. We have already presented possible solutions to these issues in our earlier paper [MiT07a]. Please refer to that article for details related to this topic.

Solving the networking and security issues. The networking and security issues arise from the combination of the current "one size fits all" browser security model and the general document-oriented nature of the web browser. Decisions about security are determined primarily by the site (origin) from which the web document is loaded, not by the specific needs of the document (in this case: the application). Such problems could be alleviated by introducing a more fine-grained security model, e.g., a model similar to the comprehensive security model of the Java SE platform [GED03] or the more lightweight, permission-based, certificate-based security model introduced by the MIDP 2.0 Specification for the Java™ Platform, Micro Edition (Java ME) [RTV03].

The biggest challenges in this area are related to standardization, as it is difficult to define a security solution that would be satisfactory to everybody while retaining backwards compatibility with the existing solutions. Also, any security model that depends on application signing and/or security certificates involves complicated business issues, e.g., related to who has the authority to issue security certificates. Therefore, it is likely that any resolutions in this area will still take years. Meanwhile, a large number of security groups and communities – such as the Open Web Application Security Project (OWASP), the Web Application Security Consortium (WASC), and the W3C Web Security Context Working Group – are working on the problem.

Solving the interoperability and compatibility issues. As in the security area, the issues in the browser compatibility area are heavily dependent on standardization. In order to improve compatibility, an independently developed browser compatibility test suite, similar to the test suites available for the Java platform, would be very valuable. For each new browser feature, a reference implementation should also be made available. Having an independent third-party compatibility test organization might

also help. If such an organization were available, new browser versions could be subjected to third-party compatibility testing before the new versions of the browser will be released to the public.

In general, improved communication and collaboration between the browser vendors are key to any improvements in this area. Additional standardization work is needed especially in the area of JavaScript library specification. As we mentioned earlier in Section 4.3, standard JavaScript APIs are still missing from important areas such as advanced networking, 2D/3D/vector graphics, audio, video and other advanced media capabilities. This will slow down the development of more comprehensive, cross-browser web applications.

Solving the development and testing issues. As we discussed earlier, the transition from conventional applications to web applications will result in a shift away from static programming languages such as C, C++ or C# towards dynamic programming languages such as JavaScript, PHP or Python. Since mainstream software developers are often unaware of the fundamental development style differences between static and dynamic programming languages, there is a need for education in this area. Developers need to be educated about the evolutionary, exploratory programming style associated with dynamic languages, as well as the agile development methods and techniques that are available for facilitating such development.

In the testing area, there is an increased need for code coverage testing to ensure that all the parts of the applications are tested appropriately in the absence of static checking. Some of the problems can also be solved by tool support. For instance, static verification tools, such as *jslint* for JavaScript (http://www.jslint.com/), can be valuable in checking the integrity of an application before its actual execution.

Solving the deployment issues. One of the main benefits of the Web is instant worldwide deployment: Any artifact that is posted on the Web is immediately accessible to anybody in the world who has a web browser. This “instant gratification” dimension will revolutionize the deployment and distribution of software applications, and will imply various changes in the business model of almost everyone in the software industry.

One of the main challenges in the deployment area is to define a model that addresses the fundamental changes in the nature of applications that we discussed in Section 4.5: applications that are always on, the ever-shortening release cycles (towards “nano-releases”), and the perpetual beta syndrome. We plan to focus on these topics in more detail in another research paper.

Solving the performance issues. As already mentioned, the JavaScript virtual machines, graphics library implementations, and native function bindings in today's web browsers are surprisingly slow. For instance, JavaScript virtual machines are still one to two orders of magnitude slower than high-performance virtual machines for other, comparable dynamic programming languages. Now that people have started running significant desktop-style applications on the Web, these performance problems are becoming increasingly apparent. Fortunately, solutions in this area are relatively straightforward. Techniques for high-performance virtual machine implementation have been investigated for decades. Plenty of existing expertise exists in this area, both to support faster execution and more efficient memory management. To improve graphics performance and native function bindings, various techniques are also available, including closer integration with hardware-accelerated graphics engines.

6. Conclusions

Whether we like it or not, the World Wide Web is increasingly the platform of choice for advanced software applications. Web-based applications have major benefits: in particular, they require no installation or manual upgrades, and they can be deployed instantly worldwide. 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 or hardware architecture.

Web-based software development will dramatically change the way people develop, deploy and use software. This will cause a paradigm shift in the software industry – in the same fashion as the Web has already transformed the sharing and distribution of documents, books, photos, music, videos and many other artifacts. 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.

In this paper, we have summarized our experiences with the web browser as an application platform. In Section 2, we provided a historical summary of the evolution of the Web, focusing especially on the ongoing transition from web pages towards web applications. We also provided a quick summary of the systems that are most commonly used for web application development today. In Section 3, we gave an overview of the Sun™ Labs Lively Kernel – an exceptionally flexible web programming environment that we have developed at Sun Labs. In Section 4, we summarized our experiences in using the web browser as an application platform, taking a look at the various issues that we have discovered based on our work on the Lively Kernel. Finally, we provided suggestions and recommendations for future improvement.

In conclusion, we are excited by the rapid emergence of the World Wide Web as a platform for real applications. Web-based applications will open up entirely new possibilities for software development, and will ideally combine the best of both worlds: the excellent usability of conventional desktop applications and the enormous worldwide deployment potential of the World Wide Web. While the web browser is not an ideal platform for desktop-style applications, the instant deployment aspect makes web applications inherently superior to conventional desktop-style applications. With our own work on the Sun Labs Lively Kernel, we have demonstrated that there is a better way to build browser-based web applications that support rich user interaction, advanced graphics, integrated development and deployment, and online collaboration. We hope that such features will become commonplace in web application development in the near future.

Finally, we encourage readers to take a look at the Lively Kernel system. The Lively Kernel is available at the following web site:

http://research.sun.com/projects/lively

Acknowledgements

The Lively Kernel is the result of a lively collaboration at Sun Labs. The authors would like to thank the following people for their valuable contributions (in alphabetical order): Mikko Kuusipalo, Kristen McIntyre, Richard Ortiz, Pekka Reijula, Stephen Uhler, Mario Wolczko.

References

CPJ05 Crane, D., Pascarello, E, James, D., *Ajax in Action*. Manning Publications, 2005.

Dus03 Dussealt, L., *WebDAV: Next-Generation Collaborative Web Authoring*. Prentice-Hall Series in Computer Networking and Security, 2003.

ECM99 ECMA Standard 262: ECMAScript Language Specification, 3rd Edition, December 1999. Web link: http://www.ecma-international.org/publications/standards/Ecma-262.htm

Fla06 Flanagan, D., *JavaScript: The Definitive Guide*, 5th Edition. O'Reilly Media, 2006.

FlM08 Flanagan, D., Matsumoto, Y., *The Ruby Programming Language*. O'Reilly Media, 2008.

FPM08 Freedman, C., Peters, K., Modien, C., Lucyk, B., Manning, R., *Professional Adobe AIR: Application Development for the Adobe Integrated Runtime*. Wrox Publishing, 2008.

GoR83 Goldberg, A., Robson, D., *Smalltalk-80: the Language and its Implementation*. Addison-Wesley, 1983.

GED03 Gong, L., Ellison, G., Dageforde, M., *Inside Java™ 2 Platform Security: Architecture, API Design, and Implementation*, 2nd Edition. Addison-Wesley (Java Series), 2003.

Goo06 Goodman, D., *Dynamic HTML: The Definitive Reference*. O'Reilly Media, 2006.

HaT07 Hanson, R., Tacy, A., *GWT in Action: Easy Ajax with the Google Web Toolkit*. Manning Publications, 2007.

IKM97 Ingalls, D., Kaehler, T., Maloney, J.H., Wallace, S., Kay, A., Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself. Presented at the OOPSLA'97 Conference. Web link: http://ftp.squeak.org/docs/OOPSLA.Squeak.html

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.

Mal95 Maloney, J.H., *Morphic: The Self User Interface Framework*. Self 4.0 Release Documentation, Sun Microsystems Laboratories, 1995.

MaS95 Maloney, J.H., Smith, R.B., Directness and Liveness in the Morphic User Interface Construction Environment. Proceedings of the 8th annual ACM Symposium on User Interface and Software Technology (UIST), Pittsburgh, Pennsylvania, 1995, pp. 21-28.

McC62 McCarthy, J., LISP 1.5 Programmer's Manual (with Abrahams, Edwards, Hart, and Levin). MIT Press, Cambridge, Massachusetts, 1962.

MiT07a Mikkonen, T., Taivalsaari, A., Web Applications: Spaghetti code for the 21th century. Technical Report TR-2007-166, Sun Microsystems Laboratories, 2007.

MiT07b Mikkonen, T., Taivalsaari, A., Using JavaScript as a Real Programming Language. Technical Report TR-2007-168, Sun Microsystems Laboratories, 2007.

Moo07 Moock, C., *Essential ActionScript 3.0*. O'Reilly Media, 2007.

Pau07 Paulson, L.D., Developers shift to dynamic programming languages, *IEEE Computer*, vol 40, nr 2, February 2007, pp. 12-15.

RTV03 Riggs, R., Taivalsaari, A., Van Peursem, J., Huopaniemi, J., Patel, M., Uotila, A., *Programming Wireless Devices with the Java(TM) 2 Platform, Micro Edition* (2nd Edition). Addison-Wesley (Java Series), 2003.

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.

Wea07 Weaver, J.L., *JavaFX Script: Dynamic Java Scripting for Rich Internet/Client-Side Applications*. Apress, 2007.

Wir71 Wirth, N., Program development by stepwise refinement. Communications of the ACM vol 14, nr 4 (Apr) 1971, pp.221-227.

YUM07 Yoshihama, S., Uramoto, N., Makino, S., Ishida, A., Kawanaka, S., De Keukelaere, F., Security Model for the Client-Side Web Application Environments. IBM Tokyo Research Laboratory presentation, May 24, 2007.

About the Authors

Dr. Antero Taivalsaari is a Principal Investigator and Senior Staff Engineer at Sun Labs. Antero is best known for his seminal role in the design of the Java™ Platform, Micro Edition (Java ME) – one of the most popular commercial software platforms in the world, with nearly two billion devices deployed so far. Antero has received Sun's Chairman's Award twice (in 2000 and 2003) for his work on Java ME technology. Since August 2006, Antero has been co-leading the Lively Kernel project with Dan Ingalls to bring desktop-style user experience to the world of web programming.

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. He recently published a book on mobile software development with Wiley & sons (Tommi Mikkonen, *Programming Mobile Devices: An Introduction for Practitioners*, John Wiley & sons, 2007).

Dan Ingalls is a Distinguished Engineer and Principal Investigator at Sun Labs. Dan is the principal architect of five generations of Smalltalk environments. He designed the byte-coded virtual machine that made Smalltalk practical in 1976. More recently, he conceived a Smalltalk written in itself and made portable and efficient by a Smalltalk-to-C translator, now known as the *Squeak* open-source Smalltalk. Dan also invented *BitBlt* – the general-purpose graphical operation that underlies most bitmap graphics systems today – as well as pop-up menus. Dan is interested in dynamic languages, graphics and kernel software. Since August 2006, Dan has been co-leading the Lively Kernel project, a project to rethink web programming and the Web itself.

Dr. Krzysztof Palacz is a Staff Engineer at Sun Labs, working in the Lively Kernel project. His research interests include distributed systems, dynamic programmming languages, Java, middleware, programming languages, virtual machines and web programming. Prior to joining the Lively Kernel project, Krzysztof made significant contributions to Sun's multitasking Java virtual machine (MVM) technology, as well as completed a Ph.D. degree in computer science at Purdue University.