Return of the Great Spaghetti Monster
Learnings from a Twelve-Year Adventure in Web Software Development

Antero Taivalsaari[1] and Tommi Mikkonen[2]

[1]Nokia Technologies, Tampere, Finland
[2]University of Helsinki, Helsinki, Finland
antero.taivalsaari@nokia.com, tommi.mikkonen@helsinki.fi

Abstract. The widespread adoption of the World Wide Web has fundamentally changed the landscape of software development. Only ten years ago, very few developers would write software for the Web, let alone consider using JavaScript or other web technologies for writing any serious software applications. In this paper, we reflect upon a twelve-year adventure in web development that began with the development of the *Lively Kernel* system at Sun Microsystems Labs in 2006. Back then, we also published some papers that identified important challenges in web-based software development based on established software engineering principles. We will revisit our earlier findings and compare the state of the art in web development today to our earlier learnings, followed by some reflections and suggestions for the road forward.

Keywords: Web Programming, Web Applications, Web Engineering, Software Engineering, HTML5, JavaScript, the Internet of Things, IoT, Programmable World

1 Introduction

The widespread adoption of the World Wide Web has fundamentally changed the landscape of software development. In the past years, the Web has become the *de facto* deployment environment for new software systems and applications. Office productivity applications and corporate tools such as invoicing, purchasing and expense reporting systems have migrated to the Web. Banking, insurance and retail industries – to name a few – have been transformed profoundly by the emergence of web-based applications and internet services. Academic papers such as this one are now commonly written using collaborative, browser-based environments instead of traditional, installed office suites. Even software development is nowadays often performed using interactive, web-based tools.

Over ten years ago, we published a number of papers on the emergence of the Web as a software development platform and associated challenges [1, 2]. At that point, the world looked very different still. Back in 2006, very few developers would write software for the Web, let alone consider using JavaScript or other web technologies for writing any serious software applications [3]. Today, the

Software as a Service (SaaS) model [4] is prevalent, and interactive, dynamic software development for the Web has become commonplace. In fact, traditional installed applications now maintain a stronghold only in the mobile realm, where the number of mobile apps (especially for iOS and Android devices) has exploded in recent years [5]. In contrast, the number of applications that people install on their personal computers has been in steady decline over the past years. The majority of activities on personal computers are now performed using a web browser, leveraging the Software as a Service model [6].

A key technical manifestation of the early years of our twelve-year adventure in web development was the *Lively Kernel* system (`http://lively-kernel.org/`), originally created at Sun Microsystems Labs in 2006-2008. The Lively Kernel was one of the first fully interactive, self-sustaining, web-based software development environment that was built on the assumption that the web browser would become a credible, full-fledged software platform [7]. While the Lively Kernel is not very widely known or used today, it did pave the way – for its part – for today's Software as a Service based software development systems and truly interactive, live web programming. A recently published ten-year anniversary paper summarizes the roots, design thinking and the evolution of the Lively Kernel from the technical perspective over the past ten years [8].

The broader software development challenges that we faced in the early years were summarized in another paper that was provocatively called "Spaghetti Code for the 21st Century" [1, 9]. In that paper, we argued that web development had reintroduced many of the spaghetti code problems that had already largely been eliminated in the software industry some ten years earlier. We listed issues that plagued web application development at the time, reminiscing us of the fabled "spaghetti code wars" in the early 1970s. We argued that web development was effectively giving rebirth to many of the same issues that were identified two decades earlier as the main culprits for unreadable, unmaintainable code.

Since then, we have been involved in the development of various other projects related to web development. In this paper, a revisited version of an earlier conference paper [10], we reflect upon our twelve-year adventure in web development, focusing especially on our learnings on software development challenges associated with web-based software development. We will revisit various topics that we identified as central challenges in web development over ten years ago. Although things have generally been moving in a better direction, we argue that the "organic", rather uncontrolled evolution of the Web and the dramatic increase in popularity of web-based software development in general have exacerbated the problems and the "impedance mismatch" between web development and software engineering [1, 11]. We will also present some interesting research opportunities and directions for the next ten years.

The structure of this paper is as follows. We start the paper with a review of the software engineering principles in the context of the Web, revisiting the central challenges that we identified over ten years ago (Section 2). We then take a look at the state of web programming today, highlighting significant changes in web development since we started our journey many years ago (Section 3).

In Section 4, we compare the state of the art in web development today to our earlier findings. In Section 5, we present some additional observations and technical challenges, followed by some reflections and forward looking predictions in Section 6. Finally, Section 7 concludes the paper.

2 Software Engineering Principles in the Context of Web Programming

This section provides a condensed summary of our "Spaghetti Code for the 21st Century" paper, published as a Sun Labs Technical Report in June 2007 [9] and as a conference paper (in somewhat shorter form) in 2008 [1]. The challenges identified in those publications serve as the backdrop for the evaluation and discussion later in this paper.

Back in 1968, Edsger Dijkstra started his crusade against spaghetti code [12]. 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 twisted and tangled bowl of spaghetti. The term is commonly used in negative sense to imply that a given piece of work is difficult to understand.

As underlined by the spaghetti code controversy, software engineering remained an undeveloped, unestablished practice until the late 1970s [13, 14]. 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 [15–30]. MacLennan has summarized many of these principles in his book that focuses on the principles of programming languages [31].

An important milestone in the codification of software engineering is the 1968 NATO Software Engineering Conference [32]. Besides introducing the idea of reusable software and software components [33], 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), and *generality* (required for extensibility). In that conference, the first formal definition of software engineering was also specified. As summarized by Bauer in [32], 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.*" Today, some fifty years later, that definition is still as valid as ever.

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, was and still is well-suited to displaying documents and supporting simple navigation from page to page.

However, over the years web pages have increasingly taken the form of desktop-style applications, with richer user interface and direct manipulation capabilities, as well as more advanced asynchronous communication between the clients and servers. The size and complexity of web applications and pages have also grown dramatically[1]. This has increased the need to treat web development in the same fashion as software development. An *ad hoc*, document-oriented and tool-driven approach to web development – as was common especially in the early days of web development – is insufficient in this regard.

In many ways, web application development in its early days was reminiscent of software development in the 1970s before software engineering principles were defined and systematically applied to software development. The nascent field of *Web Engineering* has emerged in response to the need to introduce sound engineering principles to web development; the first international conferences in this area (such as the ICWE conferences) were arranged in the early 2000s.

In our "Spaghetti Code for the 21st Century" paper, we grouped the challenges in web application development into three main categories based on well-documented software engineering principles. These categories were: (1) *modularity and interfaces*; (2) *consistency, simplicity and elegance*; and (4) *reusability and portability.* Furthermore, identified two additional categories of important challenges related to (4) *usability* and (5) *development style.*

Below we provide a condensed summary of the key challenges that we identified back in 2007. These challenges will serve as a backdrop for the updated discussion in the rest of this paper. The abbreviations and numbers in parentheses below (e.g., SC#1, WI#2) will be used later in this paper to refer to the earlier identified issues.

- **Modularity and interface issues**
  - *Separation of Concerns (SC#1)*: Declarative and procedural development style are mixed up.
  - *Separation of Concerns (SC#2)*: User interface component placement, user interface style elements, event declarations and application logic are mixed up.
  - *Separation of Concerns (SC#3)*: Dependence on tool support.
  - *Well-Defined Interfaces (WI#1)*: No well-defined interfaces exist between the browser and other components, apart from the Document Object Model (DOM).
  - *Well-Defined Interfaces (WI#2)*: Hard-coded references and other implementation details are used openly.
  - *Information Hiding (IH#1)*: The DOM tree is exposed and manipulated through side effects.
  - *Information Hiding (IH#2)*: Source code of applications is exposed.
  - *Information Hiding (IH#3)*: No privacy mechanisms available in JavaScript.

[1] In 2016, Wired Magazine reported that the size of the average web page had exceeded the size of the original (year 1993) Doom multiplayer 3D computer game (`https://www.wired.com/2016/04/average-webpage-now-size-original-doom/`.)

- **Consistency, simplicity and elegance issues**
  - *Consistency (C#1)*: There are several ways to perform the same functions.
  - *Consistency (C#2)*: Things should happen explicitly rather than through side effects.
  - *Simplicity and Elegance (SE#1)*: Web applications are unstructured and hard to read.
  - *Simplicity and Elegance (SE#2)*: Different types of technologies (e.g., HTML, JavaScript, CSS, XML) are mixed up.
- **Reusability and portability issues**
  - *Reusability (R#1)*: Elements of reuse are scattered and mixed with the rest of the application.
  - *Reusability (R#2)*: Hard-coded references and other implementation details are exposed.
  - *Portability (P#1)*: There are still significant differences between browsers and browser versions.
  - *Portability (P#2)*: Portability of (developer) experience is poor.
- **Usability issues**
  - *Usability (U#1)*: The browser I/O model is poorly suited to desktop-style applications.
  - *Usability (U#2)*: The semantics of many browser features are unsuitable for applications.
- **Development style issues**
  - *Development Style (DS#1)*: No transitive closure of program structures is available statically.
  - *Development Style (DS#2)*: There is no support for static verification or static type checking.

Each of the issues was discussed in detail in the original technical report [9] and conference paper [1]. For a detailed description of the issues listed above, refer to those papers. In general, our earlier studies pointed out an *impedance mismatch* between software engineering and web development – the former was already an established discipline with well-defined, mature methodologies and commonly understood engineering practices, while the latter was based on a combination of *ad hoc* practices and tools. Let us next examine the state of the art in web development some twelve years later.

3 State of Web Programming Twelve Years Later – General Observations

In this section we will take a look at the state of the art in web programming today, approximately twelve years later since our original analysis. We will begin with a general overview of the changes that have occurred in the past several years. Later in the paper we will then reflect and map the present state in the industry to our original findings.

3.1 The Web and the Software as a Service (SaaS) model have redefined personal computing

Today, the use of the Web as a software platform and the benefits of the Software as a Service model are widely understood [4, 34]. For better or worse, the web browser has become the most commonly used desktop application; often the users no longer open any other applications on their PCs than just the browser. Effectively, for many average computer users today, the browser *is* the computer. A recent VisionMobile developer survey report strongly confirmed this observation, proposing the following key trends [6]:

- The browser has become the default interface for desktop applications.
- If the browser isn't used to run the desktop app, it is being used to distribute it.
- ChromeOS is gaining a foothold in Southern Asia.

Based on the points above, it is fair to say that the Web and the Software as a Service model have redefined the notion of personal computing. Although conventional desktop applications do still exist and are still widely used, desktop applications and their deployment model are now primarily web-based. Perhaps the most representative example of this ongoing paradigm shift is Microsoft's web-based Office 365 productivity suite (`https://www.office.com/`) that replaces Microsoft's earlier (native) Office suite – the most iconic and prevalent software product of the earlier PC era. This trend has also sparked the introduction of totally new computing device categories, such as Google's purely browser-based Chromebook personal computers (`https://www.google.com/chromebook/`) running the web-based ChromeOS operating system.

3.2 JavaScript has become a very popular programming language

Due to the central role of the web browser, JavaScript has become one of the most popular programming languages in the world, just as we anticipated ten years ago. While JavaScript language standardization work was stalled for many years, there is now major progress on the standards front. The ECMAScript 6 Specification was finally published in June 2015 [35], followed by ECMAScript 7 Specification a year later [36]. Although the suitability of the JavaScript language for large masses of software developers can still be debated, ECMAScript 6 – also known as ECMAScript 2015 – is actually a decent and expressive programming language, providing support for features such as modules, class declarations, lexical block scoping, iterators and generators, promises for asynchronous programming, and proper tail calls. Furthermore, libraries and tools have come to rescue for numerous other problematic characteristics. For instance, Flow (`https://flow.org/`) is a static type checker for JavaScript programs, which can also determine the completeness of applications.

3.3 Interactive, visual development on the Web has become commonplace

From the viewpoint of the original Lively Kernel vision (see [7]), it is interesting to note that interactive, visual development for the Web has become commonplace. There are numerous interactive HTML5 programming environments such as *Cloud9* (`https://c9.io/`), *Codepen.io* (`http://codepen.io/`), *Dabblet* (`http://dabblet.com/`), *JSBin* (`https://jsbin.com/`), *JSFiddle* (`https://jsfiddle.net/`), and *Plunker* (`https://plnkr.co/`) that capture many of the original qualities of the Lively vision – such as the ability to perform software development entirely within the confines of the web browser. In the research front, CoRED has investigated the possibilities of collaborative coding [37].

In addition, there are *web curation systems* (see [38]) and JavaScript visualization libraries such as *Chart.js* (`http://www.chartjs.org/`), *Cola.js* (`https://github.com/tgdwyer/WebCola`), *D3* (`https://d3js.org/`), and *Vis.js* (`http://visjs.org/`) that provide rich, interactive, animated 2D and 3D visualizations for the Web, very much in the same fashion as we envisioned when we started the work on the Lively Kernel back in 2006. A central difference, though, is that these new libraries are intended primarily for data visualization rather than for general-purpose application development.

3.4 Web browser performance and JavaScript performance have improved dramatically

While the original versions of the Lively Kernel ran slowly, advances in web browsers and high-performance JavaScript engines soon changed the situation dramatically. The emergence of Google's Chrome web browser and the V8 JavaScript engine – created at Google by some of our former colleagues from Sun Microsystems – kick-started web browser performance wars. Raw JavaScript execution speed increased roughly by three orders of magnitude between years 2006 and 2013, effectively repeating the same dramatic performance advances that had occurred with Java virtual machines ten years earlier when those VMs evolved from simple interpreter-based systems to using advanced adaptive just-in-time compilation techniques. Although improvements in the UI rendering area have been less dramatic, from the end user's perspective today's web browsers are easily 10-20 times faster than ten years ago [39]. This has made it possible to run serious applications in the web browser. (Sadly, this has also enabled much richer use of interactive advertisements on web sites.)

3.5 HTML, CSS and the DOM turned out to be much more persistent than anticipated

The browser and JavaScript performance improvements – while definitely impressive – were not really unforeseen to us. We were convinced that the performance problems of the browser and JavaScript would ultimately get resolved. However, what was unforeseen to us how "sticky" the original core technologies in web development – HTML, CSS and JavaScript – as well as the use of the Document Object Model (DOM) would be. Our assumption was that software developers would prefer having a more uniform, conventional set of imperative graphics APIs – supporting direct, programmatic object manipulation much in the same fashion as in conventional desktop operating systems – instead of using features that were originally designed for document layout rather than for programming.

Furthermore, when we gave presentations in web developers conferences in the late 2000s, reminding web developers of traditional software engineering principles such as modularity, separation of concerns and the general importance of keeping specifications and public interfaces separate from implementation details [19], web developers shrugged and noted that the use of HTML, CSS and JavaScript already gave them the necessary separation. Likewise, the ability to manipulate graphics by poking the global DOM tree from anywhere in the application was seen as a perfectly acceptable way of doing things rather than as something that would raise any serious concerns.

In recent years, things have gone in a better direction given the earlier mentioned modularity mechanisms that have been added to the ECMAScript language, increasing use of RESTful APIs, as well as upcoming support for *Web Components* (`https://www.w3.org/TR/\#tr\_Web\_Components`). Web Components bring component-based software engineering principles to the World Wide Web, including the interoperability of higher-level HTML elements, encapsulation, information hiding and the general ability to create reusable, higher-level UI components that can be added flexibly to web applications.

3.6 Instant worldwide deployment and dramatically faster release cycles have become commonplace

When the Lively Kernel project was started, the majority of software deployments at Sun Microsystems were still done in a conventional fashion by distributing physical CDs/DVDs or by making new binary installers available on the Web. New software releases occurred relatively infrequently, perhaps a few times per year for major software products such as the Java SDK. In contrast, web-based systems allow changes to be published pretty much instantly worldwide.

Since the Lively Kernel was one of the first systems to boldly enter such an instant deployment model, we had no support from tools and techniques that have later been introduced in the context of continuous deployment [40]; this has given rise to an entirely new development process around associated automation and tools. In hindsight, it is amazing how quickly the traditional deployment model was replaced by instant worldwide deployment enabled by the Software as a Service model. This has resulted in dramatically faster release cycles as well as in the rise of entirely new continuous development and deployment practices methodologies across the industry, including DevOps [41]. These topics are now so widely studied and documented that we do not need to dive more deeply into them in this paper. For details the reader is referred to [42, 40, 43].

4 Comparing Then and Now – Reflections on Software Engineering Principles

In our original Spaghetti paper, we divided the challenges in web-based software development into three main categories and two additional areas based on established software engineering principles. We will now reflect on the present state of the art in web programming in light of those five categories. Instead of revisiting each earlier identified issue in detail, we will provide an overview of the key changes in the past decade in tabular format, complemented with some discussion.

4.1 Revisiting the Modularity and Interface Issues

Originally, the most essential modularity issues that we identified were related to the mixture of procedural and declarative programming style, the mixing of HTML/CSS/JS code, and the common assumption that development tools would ultimately solve the problems (instead of somebody actually addressing the underlying core issues) [1]. We also lamented the lack of well-defined interfaces and inadequate information hiding capabilities, as well as the openly exposed nature of the DOM tree and the source code comprising web pages.

Today, most of the previously identified challenges are still present at the browser level. For instance, the global nature of the DOM tree has not changed much, and web development is generally still a mixture of procedural and declarative development styles. However, improvements in the JavaScript/ECMAScript language as well as the availability of richer, better-designed libraries and frameworks has changed web development significantly. While in the earlier days the number of JavaScript libraries was very limited, today there is an extremely rich library ecosystem available for web development, including the currently dominant Angular.js and React.js developer ecosystems.

Table 1 presents a condensed evaluation of the main changes related to the modularity and interface issues that we identified back in 2007 [9]. For each previously identified issue we provide a short evaluation and comments on whether the issue has improved or worsened over the years.

4.2 Revisiting the Consistency, Simplicity and Elegance Issues

The second big bucket of challenges in our original Spaghetti paper was related to consistency, simplicity and elegance issues. One of the most significant observations back then was that the standards-compatible web browser offered too many ways to perform the same functions. We also lamented the unstructured nature of web applications, as well as the general tendency for web developers to mix different types of technologies, paradigms and development styles.

Table 2 presents a condensed evaluation of the main changes related to the consistency, simplicity and elegance issues that we identified back in 2007 [9]. For each previously identified issue we provide a short evaluation and comments on whether the issue has improved or worsened over the years.

Table 1. Modularity and Interface Issues Revisited

Issue	**Evaluation**	**Comments**
SC#1	Improved	Today, developers rarely perform low-level DHTML programming anymore. Thus, accidental mixing of programming paradigms is less common. The programming paradigm (declarative vs. procedural) is determined largely by the choice of libraries/frameworks on top of the web browser.
SC#2	Improved	Today, developers rarely use the low-level DHTML mechanisms for component placement or event declarations directly. Thus, accidental mixing of different types of declarations is less common. Today, these aspects are driven largely by the choice of libraries/frameworks on top of the browser.
SC#3	Neutral	In many ways, dependence on tool and library support in web development has increased considerably over the years. Whether this is a good or bad trend is subject to debate and personal preferences. The availability of richer and more mature development frameworks has definitely alleviated many of the issues identified earlier.
WD#1	Mostly neutral	The number of APIs provided by a standards-compatible web has increased over the years. However, the DOM is still the primary communication interface inside the browser.
WD#2	Improved	Today, developers rarely perform low-level DHTML programming anymore. Thus, hard-coded references or other implementation details are not as exposed as earlier. The actual improvements are dependent on the choice of libraries/frameworks on top of the browser.
IH#1	Slightly improved	The DOM is still an exposed, global data structure. Web Components ameliorate the issues by providing support for encapsulation, information hiding and the general ability to create encapsulated DOM elements. However, Web Components are not in widespread use yet.
IH#2	Neutral	The source code of web applications is still as exposed as earlier. Obfuscation techniques are used commonly by developers to hide source code.
IH#3	Improved	JavaScript/ECMAScript language improvements (e.g., modules and proper lexical block scoping) that have improved the situation and reduced the danger of accidental name clashes.

Table 2. Consistency, Simplicity and Elegance Issues Revisited

Issue	**Evaluation**	**Comments**
C#1	Worsened	The number of programming models offered by a generic web browser has actually increased over the years. For instance, the introduction of the WebGL API has made it possible to perform web rendering using yet another built-in API. Also, the number of libraries and frameworks has increased dramatically over the years. Thus, developers are faced with even more choices and even more ways to perform the same functions.
C#2	Improved	Today, developers rarely use the low-level DOM manipulation operations directly. This has reduced the use of programming styles that rely on side effects. The actual improvements in this area are dependent on the choice of libraries/frameworks on top of the web browser.
SE#1	Improved	The availability of mature, higher-level libraries and frameworks has improved the overall quality and structure of web applications considerably. Again, the actual improvements in this area are driven mainly by the choice of libraries/frameworks.
SE#2	Improved	Today, developers rarely perform low-level DHTML programming anymore. Thus, accidental mixing of different types of technologies and paradigms is less common. The actual improvements in this area are dependent on the choice of libraries/frameworks on top of the browser.

Discussion. In the consistency, simplicity and elegance area, things have generally been moving in a better direction. That said, today's web application developers are faced with an even more overwhelming cornucopia of choices in almost all aspects of web development. A great example are the rendering mechanisms inside the web browser that we have studied recently in [44]. The basic observation in that article is that the generic, standards compatible web browser offers five overlapping rendering models: DHTML, Canvas API, WebGL, SVG and Web Components. The developers are confronted with various choices also in choosing communication models, e.g., whether to use Ajax [45], Comet [46], Server-Sent Events [47], WebSockets [48], WebRTC [49]), or Web Workers [50].

In the broader picture, the deficiencies of the web browser as a software platform have been tackled with an abundance of libraries. As of this writing, there are more than 1,300 officially listed JavaScript libraries in `javascripting.com`, with new ones being introduced nearly on a daily basis. Although many of the libraries are domain-specific, a lot of them are aimed squarely at solving the architectural limitations of the web browser, e.g., to provide a consistent set of manifest interfaces to perform all the programming tasks. Over the years, JavaScript libraries have evolved from mere convenience function libraries to full-fledged Model-View-Controller (MVC) frameworks providing extensive UI component sets, application state management, network communication and database interfaces, and so on.

4.3 Revisiting the Reusability and Portability Issues

The third bucket of challenges in our original Spaghetti paper was related to reusability and portability. In our earlier paper we criticized the generally unstructured nature of web applications that made it difficult to isolate and package components for reuse. We also lamented the incompatibilities between different browsers and browser versions that made it burdensome to write portable code that would work across different browsers and browser versions. Furthermore, we noted that the abundance of different libraries and frameworks reduced the overall portability of developer experience, since development guidelines and recommended practices for one library were typically different from other libraries. Table 3 presents a condensed evaluation of the main changes related to the reusability and portability issues that we identified back in 2007 [9].

Table 3. Reusability and Portability Issues Revisited

Issue	**Evaluation**	**Comments**
R#1	Improved	Today, developers rarely perform low-level DHTML programming anymore. Thus, a programming style that spreads elements of reuse in a spaghetti-like fashion is less common. Web Components make it possible write web UI components with proper encapsulation and information hiding. The actual improvements in this area are dependent on the choice of libraries/frameworks on top of the web browser (including the option to use Web Components).
R#2	Improved	Today, developers rarely perform low-level DHTML programming anymore. Thus, the use of hard-coded references is significantly less common. The actual improvements in this area are dependent on the choice of libraries/frameworks and/or Web Components.
P#1	Improved	Browser compatibility has improved significantly over the years. For instance, the event handling capabilities of Microsoft browsers are now compatible with the other major web browsers. There are also much better compatibility test suites available nowadays. Then again, the rapid introduction of new browser features and APIs has a tendency to keep browsers somewhat incompatible with each other, as the browser vendors struggle to implement all the latest features.
P#2	Neutral or somewhat worsened	Regarding the portability of developer experience, the abundance of web technologies has made things even more challenging for developers, as it is not necessarily very easy to migrate skills learned with one library ecosystem to another. For instance, the transition from Angular.js development to React.js can be demanding. However, given that the majority of today's software developers grew up with web technologies, they are less burdened by patterns and conventions learned during the earlier desktop software era.

Discussion. In the reusability and portability area, things have also been moving in a better direction, primarily because the richer development frameworks, the introduction of Web Components and new JavaScript language mechanisms have encouraged the developers to write considerably more structured code and components specifically intended for reuse.

However, just like ten years ago, a central problem in web application development is browser incompatibility. While browser compatibility has generally improved significantly over the years, the rapid pace of innovation and constant introduction of new features has kept browsers pacing each other as browser vendors have prioritized their implementation roadmaps differently. Over the years, there have also been business and legal reasons for some of the incompatibilities, such as the intellectual property rights issues in the media codec area. Because of incompatibilities, developers still often depend on compatibility bridge libraries such as Modernizr (`https://modernizr.com/`) that detect missing features in the underlying browser and fill in the gaps automatically.

4.4 Revisiting the Usability and User Experience Issues

The fourth bucket of challenges in our original Spaghetti paper was related to usability. In our earlier paper we focused the usability analysis on the impedance mismatch between traditional desktop applications and the document-oriented, page-oriented application model introduced by the web browser. We noted that in web applications user interaction was based primarily on pages and hyperlinks, as opposed to PC applications that supported modern (or at least modern back then) user interaction features such as direct manipulation, menu-oriented navigation, and a rich set of interactive graphical widgets.

Table 4. Usability Issues Revisited

Issue	**Evaluation**	**Comments**
U#1	Neutral	The page-oriented user interaction model can still be considered suboptimal for desktop applications. The introduction of more advanced communication capabilities in the web browser, such as asynchronous HTTP requests [45] and Server-Sent Events [47], have alleviated issues considerably, though. More broadly, the web browser has become such a dominant environment for applications that our earlier observations are mostly irrelevant nowadays.
U#2	Neutral	The semantics of many of the browser buttons and other features (e.g., the context menus that open up when right-clicking objects in a desktop browser) are still poorly suited to applications. However, the web browser has become such a dominant environment for applications that our earlier observations are mostly irrelevant nowadays.

Table 4 presents a condensed evaluation of the main changes related to the usability and user experience issues that we identified back in 2007 [9].

Discussion. Overall, the user experience of the web browser has not changed very much from the page-oriented *back-forward-reload* metaphor introduced by the NCSA Mosaic browser in the early 1990s [51]. Ten years ago, the page-oriented interaction and navigation model of the web browser seemed like a throwback to an earlier era. We remarked that the interaction model of the web browser was reminiscent of the I/O model of the IBM 3270 series terminals of the 1970s – in both systems the entire display was updated in response to each successful user-initiated network request. We also lamented the poorly defined semantics of many of the browser functions and buttons. For instance, semantics of the 'back', 'stop' and 'reload' buttons were unclear when these features were used in desktop-style web applications.

Even today, it still is not entirely clear to the user what will happen if the user presses the 'back', 'stop' or 'reload' button during a financial transaction initiated from a browser-based banking or stock trading application. To avoid potentially harmful (and expensive) interactions, some web applications explicitly disable many of the browser's navigation buttons.

Interestingly, even though the observations that we presented earlier are still valid today, the usability issues have become mostly irrelevant because of the dominant role that the web browser and the Software as a Service (SaaS) model have today. Nowadays, the majority of computer users are so accustomed to the web browser and its user interface that they rarely miss the features or conventions of the earlier PC-era desktop applications. Browser-based interaction has simply become the norm also for desktop applications[2]. For the average computer user, the web browser effectively is *the* application platform now.

It should be noted that the adoption of Single-Page Application (SPA) development style [52, 53] and its support in popular frameworks such as Angular.js [54] have improved the overall user experience considerably for those web sites that wish to behave more like classic desktop applications. Nevertheless, we still think that a lot of room remains in improving the overall usability of web applications. Those discussions are beyond the scope of this paper, however.

4.5 Revisiting the Development Style Issues

The fifth and last bucket of challenges in our original Spaghetti paper was related to development style. We pointed out that the field of web programming bears the imprint of the document-oriented – as opposed to application-oriented – roots of the Web. The programming capabilities of the Web have largely been an afterthought – designed originally for relatively simple scripting tasks. For instance, the JavaScript language was originally created by Brendan Eich in ten days in May 1995.

[2] Even this paper was written using the online tool *Overleaf* instead of a conventional, PC-based word processing application such as Microsoft Word.

In our earlier paper the analysis of development style issues focused primarily on the differences that arise from the use of dynamic vs. static programming languages. Ten years ago, the software development landscape was dominated by statically compiled programming languages such as C, C++ and Java. The use of interpreted, dynamic programming languages such as Lisp, Scheme, Python or JavaScript was limited. Today, dynamic languages (especially JavaScript and Python) have a central role especially in client-side development, driven by the expectation and need to have much shorter release cycles and the ability to perform changes in near real time [55].

Table 5 presents a condensed evaluation of the main changes related to the development style issues that we identified back in 2007 [9].

Table 5. Development Style Issues Revisited

Issue	**Evaluation**	**Comments**
DS#1	Improved	An abundance of tools is available nowadays to ensure the completeness of web applications. For instance, the earlier mentioned Flow tool (`https://flow.org/`) is a static type checker for JavaScript programs that can also determine the completeness of applications. Tools such as Webpack module bundler can also help in packaging applications and make sure they contain all the necessary components.
DS#2	Improved	An abundance of tools is available nowadays to support static verification and static type checking of web applications. There are also JavaScript language extensions and variants such as TypeScript that make optional type checking available with the latest ECMAScript features (`https://www.typescriptlang.org/`).

5 Comparing Then and Now – Additional Observations

In our WEBIST conference paper published in 2017, we listed a number of additional topic areas and web application development issues that were not covered by our original Spaghetti evaluation framework presented over ten years earlier. Below we summarize these additional observations and challenges. Some of these topics arise from the security limitations of the web browser, while other topics arise from the dramatically faster development cycles that have become commonplace in the past 5-10 years.

Limited access to local resources or host platform capabilities. Web documents and applications are run in a sandbox that places significant restrictions on the resources and host platform capabilities that 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 cookies and using the `localStorage` mechanism. While these security restrictions prevent malicious access, they make it difficult to build web applications that utilize local resources or host platform capabilities. Consequently, the functionality that can be offered by web applications is still significantly more limited than that of native applications. In this area things have not changed very much in the past ten years.

Mobile computing is still dominated by apps – for now. During the original development of the Lively Kernel system in 2006-2008, we were aiming at making the system run well also on mobile web browsers. Although the feasibility of running the system on mobile devices was demonstrated [56], in practice mobile devices and browsers were still so slow those days that no serious mobile Lively applications could be built. Furthermore, the considerably smaller screen sizes, different input modalities and limited mobile OS API access made it difficult to develop and run applications on mobile devices.

The technical reasons for the desktop and mobile app divergence are well understood nowadays [57, 58]. A major contributor to the divergence is the limited access to the underlying platform capabilities mentioned above. One approach for tackling the shortcomings of the Web as a mobile platform is to use cross-platform or hybrid app designs [59, 3]. In the late 2000s, so called *Rich Internet Application* (RIA) platforms such as Adobe AIR, Apache Cordova [60] (formerly PhoneGap) and Microsoft Silverlight [61] were very popular. RIA systems were an attempt to bring alternative programming languages and libraries to the Web in the form of browser plug-in components that each provided a complete, more efficient platform runtime (see [3]). However, just as it was predicted in [62], the RIA phenomenon turned out to be rather short-lived.

More broadly, it is interesting to note that in the past ten years desktop computing and mobile computing have evolved in entirely different directions. While personal computers are now driven mostly by the Software as a Service model, mobile devices are still dominated by native or hybrid apps. This divergence is unlikely to continue indefinitely. There are already indications that desktop and mobile operating systems will ultimately converge. For instance, Microsoft's latest Windows 10 Mobile operating system represents an attempt to unify Windows application platform across multiple device classes.

Fine-grained security model is still missing. Compared to traditional desktop applications, web applications can still be viewed as second-class citizens that are at 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. The situation is further complicated by opportunistic designs and mashware paradigm, where applications are composed out of data and code available from various web sites [11, 63].

Testing of web applications is still challenging. Related to testing, 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 runtime problems that are impossible to detect before execution. Furthermore, with scripting languages such as JavaScript applications can even modify themselves on the fly, and there is no way to statically detect the possible errors resulting from such modifications. Consequently, web applications require significantly more testing to make sure that all the possible application behaviors and paths of execution are covered.

Forgiveness and error-tolerance. The web browser and the JavaScript virtual machine have been designed to be extremely permissive and error-tolerant. As a general principle, errors are not reported unless absolutely necessary. For instance, spelling errors in JavaScript variable names implicitly result in the creation of a new variable with the misspelled name. Likewise, minor accidental syntax errors, such as using square brackets "[]" instead of parentheses "()", e.g., in string indexing operation `"String.chatAt()"`, will go unreported and can lead to problems that are very difficult to trace. While such permissiveness enables the successful execution of source code that contains spelling errors, this usually results in other, significantly more difficult errors later in the execution. When an error is finally reported, the actual problems hides elsewhere in the program. Such problems multiply when creating web-based mashups that combine code from multiple sources and different authors [64, 11].

Challenges arising from the forgiving, permissive nature of the web browser and JavaScript are tackled primarily by tools. However, declarative applications are not easy to test with present-day tools; they rely on, e.g., test coverage that has little meaning in a declarative setup. Furthermore, coverage testing has very little meaning if the application relies extensively on external libraries and modules. For instance, in a typical Node.js application today, the amount of actual application code is often marginal compared to the thousands of NPM modules that the application uses. In such an environment, actual application code may only consist of a few hundred or thousand lines, while the NPM modules used by the application contain millions of lines of code from external sources.

Fashion-driven development. Over the past years there has been a notable trend in the library area towards fashion-driven development. By this we refer to the developers' tendency to surf on the wave of newest and most dominant "alpha" frameworks. For instance, the once hugely popular *Prototype.js* and *JQuery.js* libraries are nowadays mostly forgotten, replaced by *Knockout.js* and *Backbone.js* in 2012. Back in 2014, *Angular.js* was by far the most dominant alpha framework, while in 2016-2017 it is the *React.js + Redux.js* ecosystem that is capturing the majority of developer attention. As witnessed by the somewhat unfortunate recent evolution of the Angular ecosystem, the alpha frameworks have a tendency to evolve very quickly once they get developers' attention, leading into compatibility issues. To make the matters worse, once the next fashionable major framework emerges and hordes of developers start jumping ship onto the new one, it becomes questionable to what extent one can build long-lasting business-critical applications and services, e.g., for the medical industry in which products must commonly have a minimum lifetime of twenty years. With the present pace of upgrades, the browser and the web server as the runtime environment would be almost completely replaced by patches, upgrades, and updates; similarly, most of the libraries would be replaced several times by newer ones.

Opportunistic design and cargo cult programming. In web development there has historically been a strong tradition of *mashup-based development* [64, 11]: searching, selecting, pickling, mashing up and gluing together disparate libraries and pieces of software [63]. Often such development has the characteristics of *cargo cult programming*[3]: ritually including code and program structures that serve no real purpose or that the programmer has chosen to include because hundreds of other developers have done so – without really understanding why. While this approach can save a lot of work and open up interesting opportunities for large-scale code reuse [64, 65], this approach does not foster development of reliable, long-lasting applications, because even the smallest changes in the constituent components or subsystems – each of which evolves separately and independently – can break applications [66]. In Node.js development, opportunistic design is especially common, as the developers often include numerous NPM modules for convenience, or simply because many their colleagues or other developers have done so.

6 Reflections and the Road Forward

In striking contrast with the situation twelve years ago, there is now an incredible amount of innovation in the web development area. End user sofware has largely migrated to the Web, JavaScript has become one of the most popular programming languages in the world, and new libraries and tools have become available almost on a weekly (if not daily) basis (see, e.g., `http://www.javascripting.com/`). The rapid pace of innovation and rather uncontrolled, organic evolution of the Web have resulted in a situation in which there are numerous ways to build applications on the Web – many more than most people realize, and also arguably more than are really needed. This has put the developers in a complex position in which it is difficult to choose technologies that would be guaranteed to still be around and supported ten years from now.

While the overall complexity of the web application development scene has increased, there have been improvements in nearly all the problem areas that we identified earlier. For instance, as already mentioned, there has finally been tremendous progress in ECMAScript (JavaScript) language standardization [35, 36]. Furthermore, newer browsers – in particular Microsoft's Edge browser[4] that has replaced Internet Explorer – are significantly more compatible with each other than dominant browsers ten years ago. We are confident that similar compatibility improvements will find eventually their way also to mobile web browsers that still have more significant feature deviations today (there is a good overview available at `http://mobilehtml5.org/`).

[3] https://en.wikipedia.org/wiki/Cargo_cult_programming
[4] https://www.microsoft.com/en-us/windows/microsoft-edge

In the same vein, Web Components offer hope that well-known (but hitherto missing) software engineering principles and practices will eventually find their way into the web browser, including modularity and the ability to create higher-level, general-purpose UI components that can be flexibly added to web applications. Web components are still the "dark horse" in web development – they are little known to most developers, and it is difficult to place betting odds on their eventual success. Web components cater to nearly any imaginable use case but they are especially well-suited to the development of full-fledged web applications that require an extensible set of GUI widgets.

Looking forward, we predict that the current transition towards the Internet of Things (IoT) and the Web of Things (WoT) will drive the industry towards systems that have much better support for interactive development and programming. We are moving to the *Programmable World Era* in which literally all everyday objects will be connected to the Internet and will have enough computing, storage and networking capabilities to host a dynamic programming environment, thus turning everyday objects remotely programmable [67, 68]. Such a dynamic programming vision is actually very close to one of our central goals when we started the Lively Kernel system development back at Sun Microsystems Labs in 2006 [8].

For better or worse, everyday objects around us will have more computing power, storage capacity and network bandwidth than computers that were used for entire computing departments in the 1970s and 1980s. The availability and presence of such capabilities will open up tremendous possibilities for entirely new types of applications and services. Many of the platforms under development for the IoT domain leverage Node.js, which effectively means that JavaScript may well become the *de facto* programming language for IoT applications as well.

The Internet of Things area offers a natural playground for dynamic programming capabilities provided by live object systems such as the Lively Kernel. To this end, we plan to harness and leverage the Lively environment as a web-based graphical end-user programming environment for IoT devices, with the goal to realize the broader Programmable World vision by implementing the same kind of direct manipulation capabilities that demonstrated earlier. The key difference is that rather than just making the World Wide Web more lively, we now aim at making the entire world around us programmable in an effortless and lively fashion ("Lively Things") [8].

7 Conclusions

The World Wide Web is the most powerful medium for information sharing in the history of humankind. Somewhat accidentally, the success of the Web has turned the web browser also into a dominant platform for end-user software. Today, the Software as a Service (SaaS) model is prevalent on desktop computers, while traditional installed applications still maintain a stronghold in the mobile application area.

Over ten years ago, we published a number of papers on the emergence of the Web as a software platform. We noted that the field of web programming bears the imprint of the document-oriented – as opposed to application-oriented – roots of the Web. We pointed out that the programming capabilities of the Web have largely been an afterthought – designed originally by non-programmers for relatively simple scripting tasks. We examined the state of the art in web software development in light of established software engineering principles, and enumerated issues that plagued web application development at the time. Those issues reminisced us of the fabled "spaghetti code wars" in the early 1970s.

In this paper, we have revisited our earlier findings and examined the state of the art in web software development today based on our experiences and learnings from various web development projects in the past twelve years. In almost all areas and issues that we identified a decade earlier, things have generally been moving in a better direction. However, at the same time the overall complexity of the web development landscape has increased considerably, reflecting the vast amount of innovation and interest in this space.

Furthermore, while there has recently been tremendous progress in JavaScript language evolution and in improving JavaScript performance, the majority of innovation has occurred in higher layers parts of the stack (e.g., in developing more comprehensive and powerful libraries and frameworks), leaving some of the core issues – such as the overall complexity of the web browser – still unaddressed.

Looking forward, we believe that interactive, web-based software development capabilities will become even more important in the future as the industry moves towards the *Programmable World Era* in which everyday objects around us will become connected and programmable.

References

1. Mikkonen, T., Taivalsaari, A.: Web Applications – Spaghetti Code for the 21st Century. In: Proc. Int'l Conf. Software Engineering Research, Management and Applications (SERA'2008, Prague, Czech Republic, August 20-22, 2008), IEEE Computer Society (2008) 319–328
2. Taivalsaari, A., Mikkonen, T., Ingalls, D., Palacz, K.: Web Browser as an Application Platform. In: 34th Euromicro Conference on Software Engineering and Advanced Applications (SEAA'2008, Parma, Italy, September 3-5, 2008), IEEE Computer Society (2008) 293–302
3. Casteleyn, S., Garrigós, I., Mazón, J.N.: Ten Years of Rich Internet Applications: A Systematic Mapping Study, and Beyond. ACM Trans. Web **8**(3) (July 2014) 18:1–18:46
4. Turner, M., Budgen, D., Brereton, P.: Turning Software into a Service. Computer **36**(10) (2003) 38–44
5. Petsas, T., Papadogiannakis, A., Polychronakis, M., Markatos, E.P., Karagiannis, T.: Rise of the Planet of the Apps: A Systematic Study of the Mobile App Ecosystem. In: Proceedings of the 2013 Internet Measurement Conference, ACM (2013) 277–290

6. VisionMobile: Cloud and Desktop Developer Landscape. `http://www.visionmobile.com/product/cloud-and-desktop-developer-landscape/` (2016) [Online; accessed 5-March-2016].
7. Taivalsaari, A., Mikkonen, T., Ingalls, D., Palacz, K.: Web Browser as an Application Platform: The Lively Kernel Experience. Technical report, TR-2008-175, Sun Microsystems Laboratories (2008)
8. Ingalls, D., Felgentreff, T., Hirschfeld, R., Krahn, R., Lincke, J., Röder, M., Taivalsaari, A., Mikkonen, T.: A World of Active Objects for Work and Play: The First Ten Years of Lively. In: Proceedings of SPLASH'2016 Onward! Track (Amsterdam, the Netherlands, October 30 - November 4, 2016). (2016) 238–249
9. Mikkonen, T., Taivalsaari, A.: Web Applications: Spaghetti Code for the 21st Century. Technical Report TR-2007-166, Sun Microsystems Labs, June 2007 (2007)
10. Taivalsaari, A., Mikkonen, T.: The Web as a Software Platform: Ten Years Later. In: Proceedings of the WEBIST'17 Conference, Porto, Portugal. (2017)
11. Mikkonen, T., Taivalsaari, A.: The Mashware Challenge: Bridging the Gap Between Web Development and Software Engineering. In: Proceedings of the FSE/SDP workshop on Future of software engineering research, ACM (2010) 245–250
12. Dijkstra, E.W.: Letters to the Editor: Go To Statement Considered Harmful. Communications of the ACM **11**(3) (1968) 147–148
13. Dijkstra, E.W.: Programming: From Craft to Scientific Discipline. In: International Computing Symposium. (1977) 23–30
14. Hoare, C.: Programming: Sorcery or Science? IEEE Software **1**(2) (1984) 5
15. Corbato, F.: Sensitive Issues in the Design of Multi-Use Systems. Technical report, DTIC Document (1968)
16. Parnas, D.L.: Information Distribution Aspects of Design Methodology. (1971)
17. Dahl, O.J., Dijkstra, E.W., Hoare, C.A.R.: Structured Programming. Academic Press Ltd. (1972)
18. Parnas, D.L.: A Technique for Software Module Specification with Examples. Communications of the ACM **15**(5) (1972) 330–336
19. Parnas, D.L.: On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM **15**(12) (1972) 1053–1058
20. Parnas, D.L.: On the Design and Development of Program Families. IEEE Transactions on software engineering (1) (1976) 1–9
21. Parnas, D.L.: Designing Software for Ease of Extension and Contraction. In: Proceedings of the 3rd international conference on Software engineering, IEEE Press (1978) 264–277
22. Parnas, D.L., Clements, P.C., Weiss, D.M.: Enhancing Reusability with Information Hiding. Tutorial: Software Reusability (1983) 83–90
23. Parnas, D.L., Clements, P.C.: A Rational Design Process: How and Why to Fake It. IEEE transactions on software engineering (2) (1986) 251–257
24. Morris Jr, J.H.: Protection in Programming Languages. Communications of the ACM **16**(1) (1973) 15–21
25. Morris Jr, J.H.: Types are Not Sets. In: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages, ACM (1973) 120–124
26. Liskov, B., Zilles, S.: Programming with Abstract Data Types. In: ACM Sigplan Notices. Volume 9., ACM (1974) 50–59
27. Liskov, B., Zilles, S.: Specification Techniques for Data Abstractions. In: ACM SIGPLAN Notices. Volume 10., ACM (1975) 72–87
28. Guttag, J.: Abstract Data Types and the Development of Data Structures. Communications of the ACM **20**(6) (1977) 396–404

29. Zilles, S.N.: Procedural Encapsulation: a Linguistic Protection Technique. In: ACM Sigplan Notices. Volume 8., ACM (1973) 142–146
30. Corbató, F.J.: On Building Systems that Will Fail. In: ACM Turing award lectures, ACM (2007) 1990
31. MacLennan, B.J.: Principles of Programming Languages: Design, Evaluation, and Implementation. Oxford University Press (1999)
32. Naur, P., Randell, B.: Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO. (1969)
33. McIlroy, M.D., Buxton, J., Naur, P., Randell, B.: Mass-Produced Software Components. In: Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany. (1968) 88–98
34. Bouzid, A., Rennyson, D.: The Art of SaaS: A Primer on the Fundamentals of Building and Running a Successful SaaS Business. Xlibris (2015)
35. ECMAInternational: ECMAScript 2015 Language Specification, Standard ECMA-262, 6th Edition, June 2015. `http://www.ecma-international.org/ecma-262/6.0/` (2015) [Online; accessed 22-Feb-2017].
36. ECMAInternational: ECMAScript 2016 Language Specification, Standard ECMA-262, 7th Edition, June 2016. `http://www.ecma-international.org/ecma-262/7.0/` (2016) [Online; accessed 22-Feb-2017].
37. Lautamäki, J., Nieminen, A., Koskinen, J., Aho, T., Mikkonen, T., Englund, M.: CoRED: Browser-Based Collaborative Real-time Editor for Java Web Applications. In: Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work, ACM (2012) 1307–1316
38. Lupfer, N., Kerne, A., Webb, A.M., Linder, R.: Patterns of Free-form Curation: Visual Thinking with Web Content. In: Proceedings of the 2016 ACM on Multimedia Conference (MM'16, Amsterdam, The Netherlands, October 15-19, 2016). (2016) 12–21
39. Wagner, J.L.: Web Performance in Action: Building Fast Web Pages. Manning (2016)
40. Leppänen, M., Mäkinen, S., Pagels, M., Eloranta, V.P., Itkonen, J., Mäntylä, M.V., Männistö, T.: The Highways and Country Roads to Continuous Deployment. IEEE Software **32**(2) (2015) 64–72
41. Debois, P.: Devops: A Software Revolution in the Making. Journal of Information Technology Management **24**(8) (2011) 3–39
42. Olsson, H.H., Alahyari, H., Bosch, J.: Climbing the "Stairway to Heaven" – A Multiple-Case Study Exploring Barriers in the Transition from Agile Development towards Continuous Deployment of Software. In: Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on, IEEE (2012) 392–399
43. Fitzgerald, B., Stol, K.J.: Continuous Software Engineering: A Roadmap and Agenda. Journal of Systems and Software **123** (2017) 176–189
44. Taivalsaari, A., Mikkonen, T., Pautasso, C., Systä, K.: Comparing the Built-In Application Architecture Models in the Web Browser. In: Software Architecture (ICSA), 2017 IEEE International Conference on, IEEE (2017) 51–54
45. Garrett, J.J.: Ajax: A New Approach to Web Applications http://adaptivepath.org/ideas/ajax-new-approach-web-applications/, 18 February 2005.
46. Crane, D., McCarthy, P.: What Are Comet and Reverse Ajax? Springer (2009)
47. Hickson, I.: Server-Sent Events. W3C Recommendation 03 February 2015, latest version available at http://www.w3.org/TR/eventsource/ (2015)

48. Pimentel, V., Nickerson, B.G.: Communicating and Displaying Real-time Data with WebSocket. IEEE Internet Computing **16**(4) (2012) 45–53
49. Bergkvist, A., Burnett, D.C., Jennings, C., Narayanan, A.: WebRTC 1.0: Real-time Communication Between Browsers. Working draft, W3C (2012)
50. W3C: W3C Schools – HTML Web Workers Example, `http://www.w3schools.com/html/html5\_webworkers.asp`.
51. Darken, R.: Breaking the Mosaic Mold. IEEE Internet Computing **2**(3) (1998) 97
52. Mesbah, A., Van Deursen, A.: Migrating Multi-Page Web Applications to Single-Page Ajax Interfaces. In: 11th European Conference on Software Maintenance and Reengineering CSMR'07, IEEE (2007) 181–190
53. Mikowski, M.S., Powell, J.C.: Single Page Web Applications: JavaScript End-to-End. Manning (2013)
54. Jadhav, M.A., Sawant, B.R., Deshmukh, A.: Single Page Application using AngularJS. International Journal of Computer Science and Information Technologies **6**(3) (2015)
55. Poulson, L.D.: Developers Shift to Dynamic Programming Languages. IEEE Computer **40**(2) (2007) 12–15
56. Mikkonen, T., Taivalsaari, A.: Creating a Mobile Web Application Platform: The Lively Kernel Experiences. In: Proceedings of the 24th ACM Symposium on Applied Computing (SAC'2009), proceedings vol 3. (2009) 177–184
57. Charland, A., Leroux, B.: Mobile Application Development: Web vs. Native. Communications of the ACM **54**(5) (2011) 49–53
58. Joorabchi, M.E., Mesbah, A., Kruchten, P.: Real Challenges in Mobile App Development. In: 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, IEEE (2013) 15–24
59. Dalmasso, I., Datta, S.K., Bonnet, C., Nikaein, N.: Survey, Comparison and Evaluation of Cross Platform Mobile Application Development Tools. In: Wireless Communications and Mobile Computing Conference (IWCMC), 2013 9th International, IEEE (2013) 323–328
60. Wargo, J.M.: Apache Cordova 4 Programming. Pearson Education (2015)
61. Moroney, L.: Microsoft Silverlight 4 Step by Step. Microsoft Press (2010)
62. Taivalsaari, A., Mikkonen, T.: The Web as an Application Platform: The Saga Continues. In: 37th Euromicro Conference on Software Engineering and Advanced Applications (SEAA'2011, Oulu, Finland, August 30 - September 2, 2011), IEEE Computer Society (2011) 170–174
63. Hartmann, B., Doorley, S., Klemmer, S.R.: Hacking, Mashing, Gluing: Understanding Opportunistic Design. Pervasive Computing, IEEE **7**(3) (2008) 46–54
64. Taivalsaari, A., Mikkonen, T.: Mashups and Modularity: Towards Secure and Reusable Web Applications. In: Automated Software Engineering-Workshops, 2008. ASE Workshops 2008. 23rd IEEE/ACM International Conference on, IEEE (2008) 25–33
65. Salminen, A., Mikkonen, T.: Mashups: Software Ecosystems for the Web Era. In: IWSECO@ICSOB (International Conference on Software Business). (2012) 18–32
66. Salminen, A., Mikkonen, T., Nyrhinen, F., Taivalsaari, A.: Developing Client-Side Mashups: Experiences, Guidelines and the Road Ahead. In: Proc. 14th Int'l Academic MindTrek Conference: Envisioning Future Media Environments, ACM (2010) 161–168
67. Wasik, B.: In the Programmable World, All Our Objects Will Act as One. Wired (May 2013) (2013) 462
68. Taivalsaari, A., Mikkonen, T.: Roadmap to the Programmable World: Software Challenges in the IoT Era. IEEE Software, Jan/Feb 2017 **34**(1) (2017) 72–80