JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2006; **00**:1–7 *Prepared using* smrauth.cls *[Version: 2003/05/07 v1.1]*

Research

A reference architecture for web browsers

Alan Grosskurth*, † and Michael W. Godfrey‡

David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, ON N2L 3G1, Canada

SUMMARY

A reference architecture for a domain captures the fundamental subsystems common to systems of that domain, as well as the relationships between these subsystems. The presence of a reference architecture can aid both during maintenance and at design time: it can improve understanding of a given system, it can aid in analyzing trade-offs between different design options, and it can serve as a template for designing new systems and reengineering existing ones. In this paper, we examine the history of the web browser domain and identify several underlying forces that have contributed to its evolution. We develop a reference architecture for web browsers based on two well known open source implementations, and we validate it against five additional implementations. We analyze the maintenance implications of different strategies for code reuse, and we identify several underlying evolutionary phenomena in the web browser domain; namely, *emergent domain boundaries*, *convergent evolution*, and *tension between open and closed source development approaches*.

KEY WORDS: Software architecture, reference architecture, software evolution, component reuse, web browser.

INTRODUCTION

A *reference architecture*[20] for a domain captures the fundamental subsystems and relationships between them that are common to existing systems in the domain. It aids in the understanding of these systems, some of which may not have their own specific architectural documentation. It also serves as a template for creating new systems by identifying areas in which reuse can occur, both at the design level and the implementation level. While reference architectures exist for many mature software domains such as compilers and operating systems, we are not aware of any reference architectures proposed for web browsers.

*Correspondence to: Alan Grosskurth, David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, ON N2L 3G1, Canada
†E-mail: agrossku@uwaterloo.ca
‡E-mail: migod@uwaterloo.ca

The web browser is perhaps the most widely used software application in history. It has evolved significantly over the past fifteen years; today, web browsers run on diverse types of hardware, from cell phones and tablet PCs to regular desktop computers. Web browsers are used to conduct billions of dollars worth of Internet-enabled commerce each year. A reference architecture for web browsers can help implementors to understand trade-offs when designing new systems, and can assist maintainers in understanding legacy code. Comparing the architecture of older systems with the reference architecture can provide insight into evolutionary trends occurring in the domain.

In this paper, we present a reference architecture for web browsers that has been derived from the source code of two existing open source systems and validate our findings against five additional systems. We explain how the evolutionary history of the web browser domain has influenced this reference architecture, and we identify underlying phenomena that can help to explain current trends. Although we present these observations in the context of web browsers, we believe many of our findings may represent more general evolutionary patterns which apply to other domains. This paper is organized as follows: the next section provides an overview of the web browser domain, outlining its history and evolution. We then describe the process and tools we used to develop a reference architecture for web browsers based on the source code of two existing open source systems. Next, we present this reference architecture and explain how it represents the commonalities of the two systems from which it was derived. We then provide validation for our reference architecture by showing how it maps onto the conceptual architectures of five additional systems. Finally, we summarize our observations about the web browser domain, discuss related work, and present conclusions.

THE WEB BROWSER DOMAIN

Overview

The World Wide Web (WWW) is a universal information space operating on top of the Internet. Each resource on the web is identified by a unique Uniform Resource Identifier[16] (URI). Resources can take many different forms, including documents, images, sound clips, or video clips. Documents are typically written using HyperText Markup Language[14] (HTML), which allows the author to embed hypertext links to other documents or different places in the same document. Data is typically transmitted via HyperText Transfer Protocol[15] (HTTP), a stateless and anonymous means of information exchange. A *web browser* is a program that retrieves documents from remote servers and displays them on screen, either within the browser window itself or by passing the document to an external helper application. It allows particular resources to be requested explicitly by URI, or implicitly by following embedded hyperlinks.

Although HTML itself is a relatively simple language for encoding web pages, other technologies may be used to improve the visual appearance and user experience. Cascading Style Sheets[3] (CSS) allow authors to add layout and style information to web pages without complicating the original structural markup. JavaScript, now standardized as ECMAScript[4], is a host environment for performing client-side computations. Scripting code is embedded within HTML documents, and the corresponding displayed page is the result of evaluating

the JavaScript code and applying it to the static HTML constructs. Examples of JavaScript applications include changing element focus, altering page and image loading behaviour, and interpreting mouse actions. Finally, there are some types of content that the web browser cannot display directly, such as Macromedia Flash animations and Java applets. *Plugins*, small programs that connect with the browser, are used to embed these types of content in web pages.

In addition to retrieving and displaying documents, web browsers typically provide the user with other useful features. For example, most browsers keep track of recently visited web pages and provide a mechanism for "bookmarking" pages of interest. They may also store commonly entered form values as well as usernames and passwords. Finally, browsers often provide accessibility features to accommodate users with disabilities such as blindness and low vision, hearing loss, and motor impairments.

History and evolution

Although key concepts can be traced back to systems envisioned by Vannevar Bush in the 1940s and Ted Nelson in the 1960s, the WWW was first described in a proposal written by Tim Berners-Lee in 1990 at the European Nuclear Research Center (CERN)[13]. By 1991, he had written the first web browser, which was graphical and also served as an HTML editor. Around the same time, researchers at the University of Kansas had independently begun work on a text-only hypertext browser called Lynx; they adapted Lynx to support the web in 1993. In the same year, the National Center for Supercomputing Applications (NCSA) released a graphical web browser called Mosaic, which allowed users to view images directly interspersed with text. As the commercial potential of the web began to grow, NCSA founded an offshoot company called Spyglass to commercialize its technologies and Mosaic's co-author left to co-found his own company, Netscape. In 1994, Berners-Lee founded the World Wide Web Consortium (W3C) to guide the evolution of the web and promote interoperability among web technologies. In 1995, Microsoft released Internet Explorer (IE), based on code licensed from Spyglass, igniting a period of intense competition with Netscape known as the "browser wars." Microsoft eventually dominated the market, and Netscape released its browser as open source under the name Mozilla in 1998. Figure 1 shows a timeline of the various releases of several prominent web browsers.

Since 1998, a large number of Mozilla variations have appeared, reusing the browser core but offering alternative design decisions for user-level features. Firefox is a standalone browser with a streamlined user interface, eliminating Mozilla's integrated mail, news, and chat clients. Galeon is a browser for the GNOME desktop environment[5] that integrates with other GNOME applications and technologies. The open source Konqueror browser has also been reused: Apple has integrated its core subsystems into its OS X web browser, Safari, and Apple's modifications have in turn been reused by other browsers. Internet Explorer's closed source engine has also seen reuse: Maxthon, Avant, and NetCaptor each provide additional features to IE such as tabbed browsing and ad-blocking. Although each browser engine typically produces a similar result, there can be differences as to how web pages look and behave; Netscape 8, based on Firefox, allows the user to switch between IE-based rendering and Mozilla-based rendering on the fly.

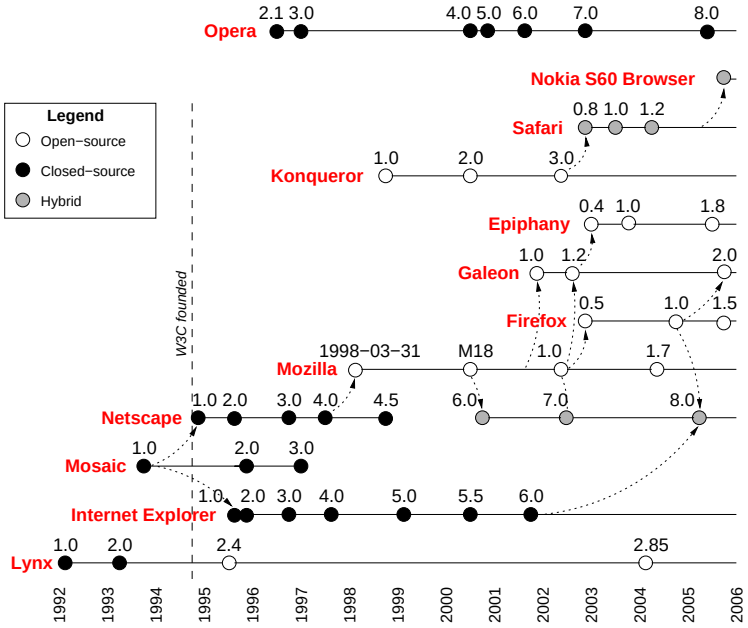


Figure 1. Web browser timeline

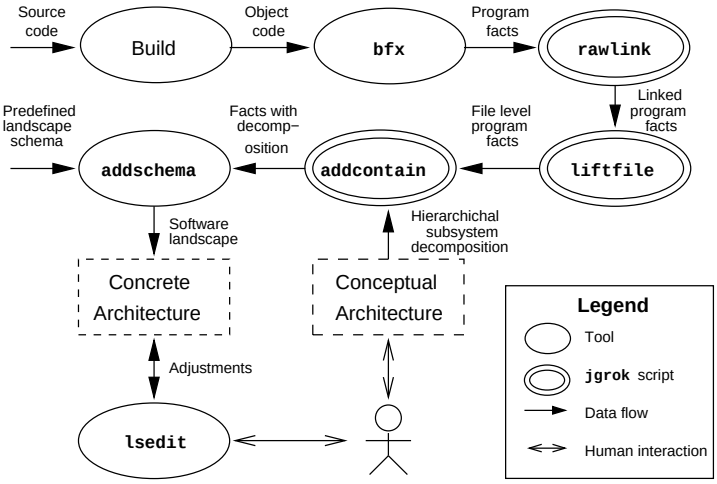


Figure 2. Extraction process for concrete architecture

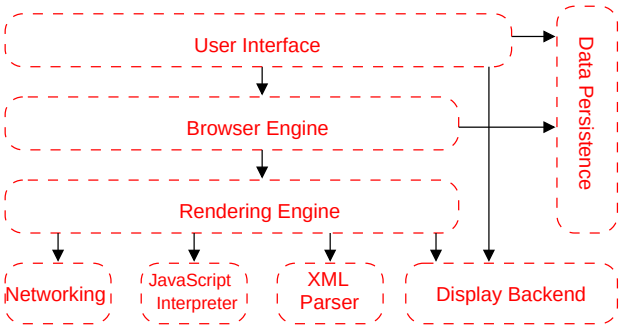


Figure 3. Reference architecture for web browsers

DERIVING A REFERENCE ARCHITECTURE

Using the source code and available documentation for two mature web browser implementations, we derived a reference architecture for the web browser domain. We used a process similar to that described by Hassan and Holt[26]. For each browser, a conceptual architecture was proposed based on domain knowledge and available documentation. The concrete architecture of each system was then extracted from its source code using the QLDX[8] reverse engineering toolkit, and this concrete architecture was then used to refine the conceptual architecture. A reference architecture was then proposed based on the common structure of these refined architectures, and it was validated against five other browser implementations.

A REFERENCE ARCHITECTURE FOR WEB BROWSERS

The reference architecture we derived is shown in Figure 3. It comprises eight major subsystems plus the dependencies between them:

1. The *User Interface* subsystem is the layer between the user and the *Browser Engine*. It provides features such as toolbars, visual page-load progress, smart download handling, preferences, and printing. It may be integrated with the desktop environment to provide browser session management or communication with other desktop applications.
2. The *Browser Engine* subsystem is an embeddable component that provides a high-level interface to the *Rendering Engine*. It loads a given URI and supports primitive browsing actions such as forward, back, and reload. It provides hooks for viewing various aspects of the browsing session such as current page load progress and JavaScript alerts. It also allows the querying and manipulation of *Rendering Engine* settings.
3. The *Rendering Engine* subsystem produces a visual representation for a given URI. It is capable of displaying HTML and Extensible Markup Language (XML) documents,

optionally styled with CSS, as well as embedded content such as images. It calculates the exact page layout and may use "reflow" algorithms to incrementally adjust the position of elements on the page. This subsystem also includes the HTML parser.

4. The *Networking* subsystem implements file transfer protocols such as HTTP and FTP. It translates between different character sets, and resolves MIME media types for files. It may implement a cache of recently retrieved resources.
5. The *JavaScript Interpreter* evaluates JavaScript (also known as ECMAScript) code, which may be embedded in web pages. JavaScript is an object-oriented scripting language developed by Netscape. Certain JavaScript functionality, such as the opening of pop-up windows, may be disabled by the *Browser Engine* or *Rendering Engine* for security purposes.
6. The *XML Parser* subsystem parses XML documents into a Document Object Model (DOM) tree. This is one of the most reusable subsystems in the architecture. In fact, almost all browser implementations leverage an existing *XML Parser* rather than rewriting their own from scratch.
7. The *Display Backend* subsystem provides drawing and windowing primitives, a set of user interface widgets, and a set of fonts. It may be tied closely with the operating system.
8. The *Data Persistence* subsystem stores various data associated with the browsing session on disk. This may be high-level data such as bookmarks or toolbar settings, or it may be low-level data such as cookies, security certificates, or cache.

The reader may wonder why we have placed the HTML parser within the rendering engine subsystem, while isolating the XML parser in a subsystem of its own. The answer: although arguably less important to the functionality of the system, the XML parser is a generic, reusable component with a standard, well-defined interface. In contrast, the HTML parser is often tightly integrated with the rendering engine for performance reasons and can provide varying levels of support for broken or nonstandard HTML. This tight integration is the result of a *design decision* and seems to be a common feature of web browser architectures.

Mozilla

The Mozilla Suite was released as open source by Netscape in 1998. Most of the system has since been redesigned or rewritten, and a large number of new features have been added. Mozilla's key design goals are strong support for web standards, support for multiple platforms, and fast page rendering. We examined version 1.7.3, which consists of approximately 2,400 kLOC. Most of the source code is written in C++, although large parts of the user interface are written in JavaScript and some legacy components are written in C. We built and extracted the Linux version of Mozilla, which uses the GTK toolkit.

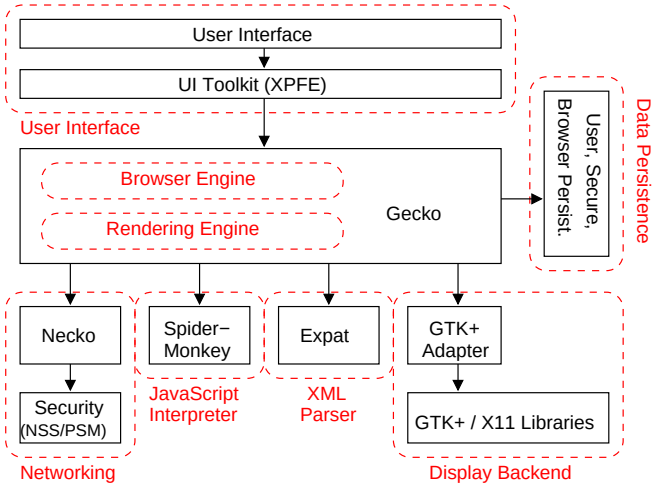


Figure 4. Architecture of Mozilla

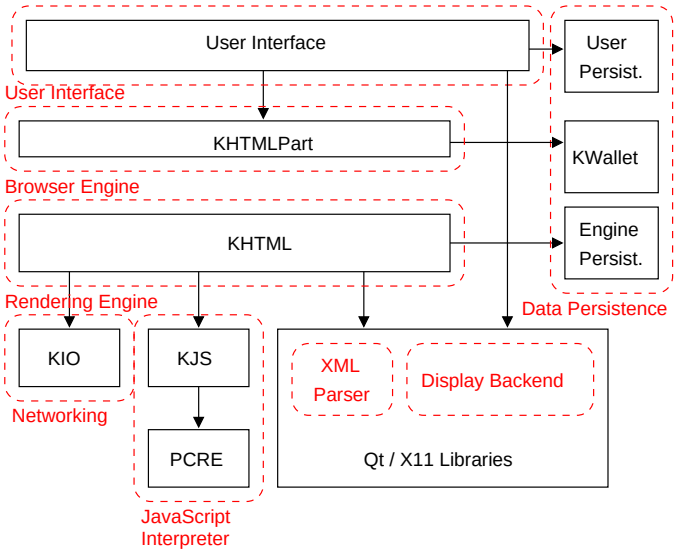


Figure 5. Architecture of Konqueror

The mapping of Mozilla's conceptual architecture onto the reference architecture is shown in Figure 4. The *User Interface* is split over two subsystems, allowing for parts of it to be reused in other applications in the Mozilla suite such as the mail/news client. All data persistence is provided by Mozilla's profile mechanism, which stores both high-level data such as bookmarks and low-level data such as a page cache. Mozilla's *Rendering Engine* is larger and more complex than that of other browsers. One reason for this is Mozilla's outstanding ability to parse and render malformed or broken HTML. Another reason is that the *Rendering Engine* also renders the application's cross-platform user interface. The UI is specified in platform-neutral Extensible User Interface Language (XUL), which in turn is mapped onto platform-specific libraries using specially written adapter components. This architecture distinguishes Mozilla from other browsers in which the platform-specific display and widget libraries are used directly, and it minimizes the maintenance effort required to support multiple, diverse platforms.

Konqueror

Konqueror[7] is the official web browser of the K Desktop Environment[6] (KDE). It can also serve as a graphical file manager and a general-purpose file viewer. Development on the browser began in 1999, and its main design goals are speed, standards compliance, and integration with KDE. We examined release 3.3.2, which consists of approximately 613 kLOC, including the required KDE libraries. Konqueror is written entirely in C++, as is most of the code in KDE. We found Konqueror's codebase to be very well organized. Modules are split up cleanly into subdirectories and often contain a concise design document explaining the main abstractions and design decisions.

The mapping of Konqueror's conceptual architecture onto the reference architecture is shown in Figure 5. Konqueror makes extensive use of various KDE libraries: KHTML performs parsing, layout, and rendering of web pages; KJS interprets embedded JavaScript code; KWallet stores sensitive data, such as passwords, with strong encryption and error detection; and KIO is an asynchronous virtual file system that automatically provides encoding and decoding over common protocols. The *XML Parser* and *Display Backend* subsystems are both provided by the Qt toolkit[9], which serves as the basis for all KDE applications; these subsystems are external to the browser itself. The Perl Compatible Regular Expressions (PCRE) library is used as a backend for the regular expression functionality of the *JavaScript Interpreter*. PCRE is a mature and well-tested component used in many other high-profile open source projects including Python and Apache. *Data Persistence* is provided at three levels. First, some high-level data such as bookmarks and history is stored by Konqueror itself. Second, other high-level data such as form completions is stored by KHTML. Third, secure data such as passwords is stored by KWallet, which allows this data to be shared with other KDE applications.

We found that Konqueror makes extensive use of existing libraries that handle difficult tasks. In contrast, Mozilla has developed almost all of these libraries in-house, delegating to other libraries only when necessary. A consequence of this is that Konqueror is closely tied to UNIX-like operating systems and the Qt toolkit, while Mozilla supports several different

operating systems and display toolkits. However, as we will see in the next section, Apple was able to adapt Konqueror to its own needs by removing the dependency on Qt.

VALIDATING THE REFERENCE ARCHITECTURE

Five additional implementations were chosen against which to validate the reference architecture: Epiphany, Safari, Lynx, Mosaic, and Firefox. Epiphany was chosen because it demonstrates reuse using exclusively open source components—it combines Mozilla's engine with GNOME desktop components. Safari was chosen because it represents an interesting mix of open and closed source technology—Apple has adapted Konqueror's core subsystems to use OS X libraries and added a proprietary user interface. Lynx was chosen because it is the oldest web browser still regularly used and maintained. Mosaic was chosen because it was the first widely used graphical web browser. Firefox was chosen because of its exceptional extensibility.

Epiphany

Epiphany is the official browser of the GNOME desktop, and it embeds the Mozilla engine. The project was started in 2003 as a separate code branch of the Galeon browser (first released in 2000) because of disagreements over user interface design decisions. We examined the source of code release 1.4.6, which is approximately 70 kLOC, although it requires approximately 1,500 kLOC of Mozilla engine code to function. Apart from the Mozilla engine code, all of Epiphany is written in C.

The mapping of Epiphany's conceptual architecture onto the reference architecture is shown in Figure 6. Since Epiphany reuses Mozilla's entire engine, the only differing subsystems are the *User Interface* and the *Data Persistence* subsystems. Epiphany's architecture contains two XML parsers: Expat is used in the Mozilla engine for parsing web content (the typical use corresponding to the *XML Parser* subsystem), while `libxml2` is used by Epiphany to serialize high-level application data. Since the Expat parser is hidden as an implementation detail inside Mozilla, it is not possible for Epiphany to reuse it for this task.

Another component that Epiphany does not take full advantage of is Mozilla's XUL abstraction layer, which allows consistent user interfaces across multiple platforms. Instead, Epiphany directly uses the GTK+ toolkit, one of the supported XUL backends, along with several GNOME libraries. This is done in order to achieve closer integration with the GNOME desktop. It is interesting to note that data persistence is provided at three separate levels. User preferences are stored by the GConf configuration system, which stores preferences for all GNOME applications. High-level data such as toolbars, history, and bookmarks is stored by Epiphany itself in an XML format. Finally, low-level data such as cache, certificates, and cookies is stored by Mozilla's profile mechanism.

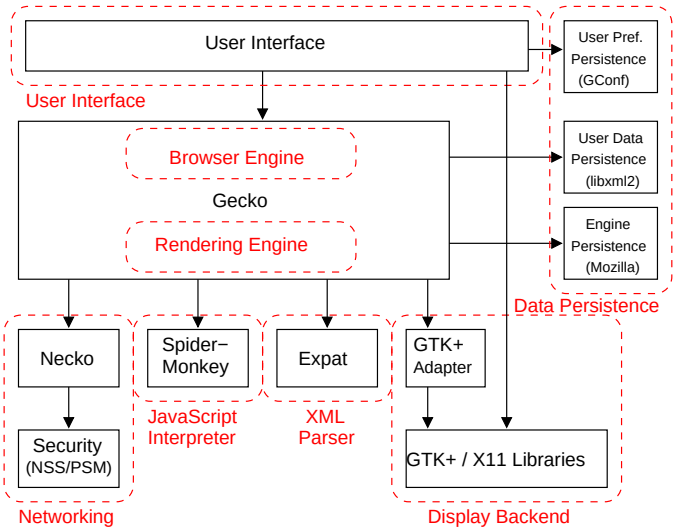


Figure 6. Architecture of Epiphany

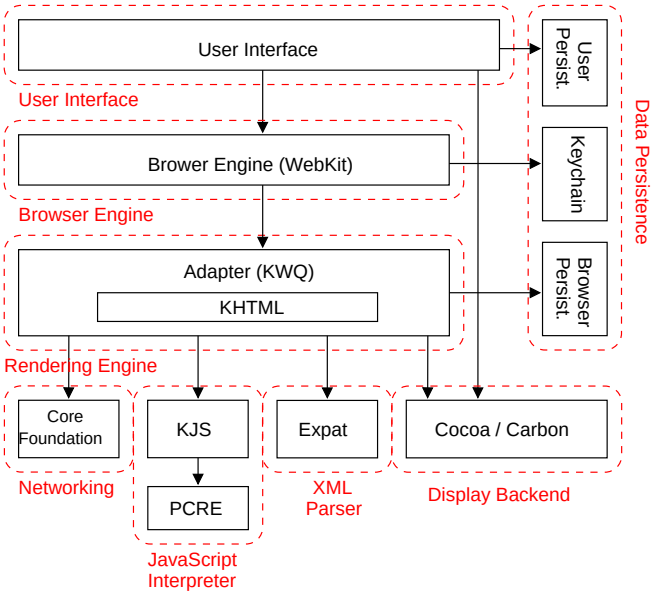


Figure 7. Architecture of Safari

Epiphany's approach to code reuse has affected the way maintenance must be performed. Epiphany treats the Mozilla code as a blackbox—it is reused without modification and all communication takes place through a small number of well-defined interfaces. Epiphany's code is stored in a repository for the GNOME desktop; Mozilla's code is stored in a repository for all Mozilla products. The two projects use separate bug-tracking systems and mailing lists. This approach avoids duplication of effort; bug fixes and feature enhancements to the Mozilla engine are realized immediately in Epiphany, as long as the relevant interfaces are not changed. If they are, Epiphany developers need to modify Epiphany to make use of the updated interfaces. Consequently, for each release of Epiphany, there is typically one suggested version of Mozilla with which it is compatible and stable.

Safari

Safari[10] is a web browser developed by Apple Computer for its Mac OS X operating system. The first version was released in January 2003. The main design goals for Safari are usability, speed, standards compliance, and integration with OS X. Safari reuses the KHTML rendering engine and the KJS JavaScript interpreter from the KDE project. The modified versions are called WebCore and JavaScriptCore, and are released under the GNU Lesser General Public License (LGPL). However, the rest of Safari's code is proprietary, including the user interface.[†] We examined the source code of release 125 of WebCore and JavaScriptCore, which consists of 114 kLOC of C++ code and 22 kLOC of Objective C++. Since we could not extract the proprietary parts, their structure was inferred from Apple's developer documentation[2].

The mapping of Safari's conceptual architecture onto the reference architecture is shown in Figure 7. The *Rendering Engine* is composed of the KHTML core engine wrapped in the KWQ adapter. KWQ is written in Objective C++, allowing it to present an Objective C API to KHTML, which is written in C++. This was needed for integrating Safari into OS X. *Networking* functionality is provided by OS X's Core Foundation networking library, used in place of KIO. The *XML Parser* subsystem is provided by the Expat XML parser, used in place of the *XML Parser* provided by the Qt toolkit. The *Display Backend* subsystem is composed of two complementary libraries: Carbon and Cocoa. Carbon provides a lower-level C API for display routines, while Cocoa provides a higher-level Objective C API. Persistent data is handled by three separate system-wide services that are built into OS X: Preferences, Keychains, and Caches. The use of these services allows Safari to to integrate smoothly with other OS X applications.

Overall, Safari's conceptual architecture corresponds well to our reference architecture. Safari reuses the core engine from Konqueror, substitutes a Mac OS X look-and-feel, and makes use of other components and libraries native to OS X in place of the Linux- and KDE-specific components of Konqueror. Apple maintains its own parallel version of Konqueror's engine in order to achieve total control over its development and adapt the code to fit the needs of OS X. A consequence of this decision is that bug fixes and new features contributed to Safari

[†]In June 2005, Apple released another component of Safari, the WebKit framework, as open source.

must be propagated manually to Konqueror (and vice versa), or else rewritten from scratch. Since the initial fork, Safari's engine has been developed at a faster pace than Konqueror's with various OS X-specific changes, causing the two codebases to diverge. Recently, however, updates to Safari were reused in Konqueror, helping it pass the Acid2 browser test[1]. Also, several technologies from Konqueror have been ported to Safari, including the scalable vector graphics engine (KSVG2), the new Document Object Model architecture (KDOM), and the render tree library (KCanvas).

Lynx

Lynx is a one of the most popular text-only browsers in use today. It predates the WWW, first serving as an interface for an "organization-wide information system." Custom hypertext capabilities were then added, followed by support for the Gopher protocol. Finally, support for WWW protocols was grafted on, making Lynx into a true web browser. This incremental development process has resulted in a system composed of small fragments of code with no coherent overall structure. Furthermore, much of the code is low-level and platform-specific, increasing its complexity.

The mapping of Lynx's conceptual architecture onto the reference architecture is shown in Figure 8. The `libwww` library provides a wide variety of functionality such as HTML parsing and support for both the HTTP and FTP protocols. The `libgnutls` library provides optional support for secure protocols. The `curses` library is used to display and navigate information on character-cell terminals. Lynx's conceptual architecture shows a clear separation between three main subsystems: browser core, networking, and display backend; however, there is no clear separation between the *User Interface*, *Browser Engine*, *Rendering Engine*, and *Data Persistence* subsystems. This is likely because they are less complex due to Lynx's text-only nature—the rendering engine outputs web pages in linear form rather than attempting to lay out elements at appropriate coordinates, and the user interface relies solely on keyboard input rather than dealing with menus, widgets, and mouse events. Lynx does not contain a *JavaScript Interpreter* or an *XML Parser* because these are relatively modern features that are not yet supported.

The lack of modularity and the text-only nature of Lynx make its conceptual architecture much simpler than our reference architecture. The missing components draw attention to changes that have occurred in the web browser domain since the era when Lynx was actively being developed. Due to the distributed and open nature of the web, new technologies are constantly being employed by browsers to enhance the user experience. Once browser support for a particular technology reaches a critical threshold, authors may begin using the technology in their web pages. This effectively makes browsers that do not keep pace, such as Lynx, less useful than browsers that do. However, Lynx still suffices for browsing many types of sites, hence its continued popularity with some users.

Mosaic

Mosaic is the first graphical web browser designed for widespread use. First released in 1993, it introduced numerous innovations such as rendering images interspersed with text

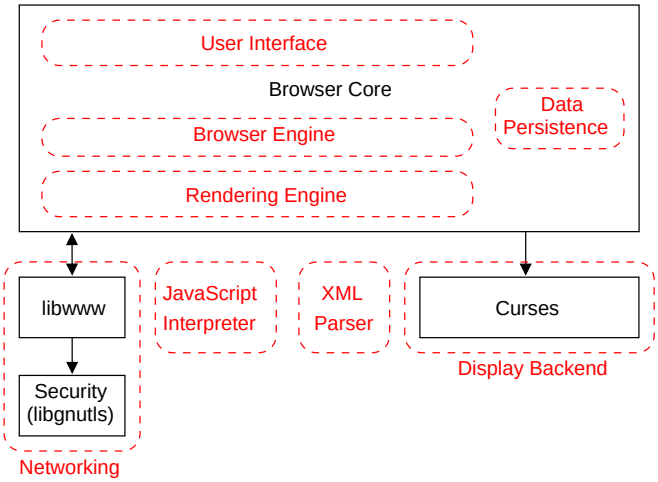


Figure 8. Architecture of Lynx

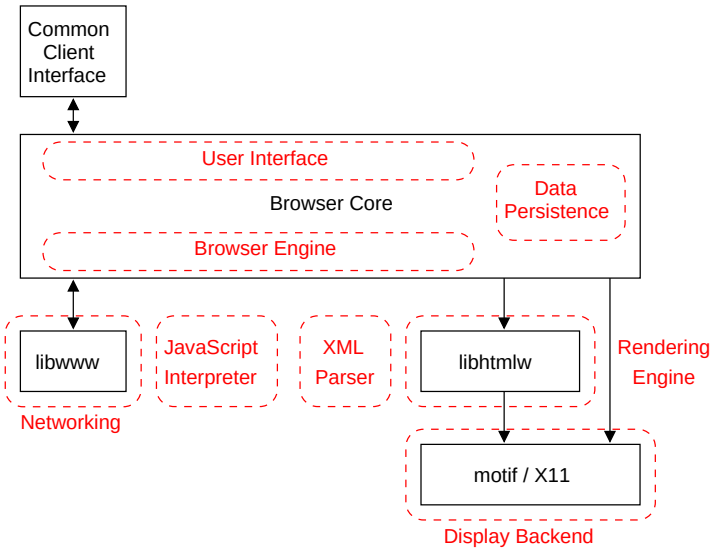


Figure 9. Architecture of Mosaic

and multimedia capabilities. It was developed at NCSA by a small team of programmers. Separate versions were maintained for each of the Windows, Mac, and UNIX platforms, although some common code was shared among them. Mosaic was surpassed by Netscape and Internet Explorer, and development stopped officially in 1997. Although source code for the UNIX version was publicly available, the license for it is not technically open source. It is still possible to obtain the source code and compile and run it on current UNIX systems. We examined the source code of release 2.7b6, which is approximately 88 kLOC and is written entirely in C.

The mapping of Mosaic's conceptual architecture onto the reference architecture is shown in Figure 9. As with Lynx, Mosaic is too old to contain modern subsystems such as a *JavaScript Interpreter* or *XML Parser*. The *Rendering Engine* subsystem is realized by `libhtmlw`, a generic display widget used to render document data. Although it performs the same task as modern rendering engines, it is much smaller in size because the layout options for early web pages were vastly simpler than they are for modern web pages. The Common Client Interface (CCI) was an experimental subsystem that allowed external applications to communicate with currently running sessions of Mosaic via TCP/IP. Using the CCI, a client could perform HTTP actions including GET and POST, arrange to have particular types of rendering output forwarded to it, or register to be notified each time Mosaic loaded a new URI. The CCI is closely coupled with the browser core and is an interesting example of a subsystem not typically found in modern web browsers.

Mosaic reuses Tim Berners-Lee's `libwww` library, albeit heavily modified, as does Lynx. Many of the changes made by Mosaic were later incorporated back into the original version of the library. This early instance of open source code reuse among browsers marks the beginning of a crucial trend in the web browser domain: the willingness of implementors to build on existing, mature code rather than attempting to develop equivalent functionality from scratch. At the time, `libwww` was the only library available providing HTTP protocol and HTML parsing functionality, and its reuse dramatically reduced the amount of effort required to develop a new web browser.

Firefox

Firefox is a variation of Mozilla, providing a streamlined interface and removing many of the integrated components such as clients for mail, news, and chat. We examined release 1.0 of Firefox, which is approximately 2,400 kLOC (it shares most of its code with the Mozilla Suite). In most respects, Firefox's conceptual architecture is similar to Mozilla's; hence we will not show a mapping of it onto the reference architecture. However, Firefox does contain one notable feature not found in the reference architecture: a powerful extension facility. While standard browser plugins are used to display content that the browser is unable to display directly, extensions can "hook in" and alter the browser at various levels in the architecture. For example, extensions can modify user interface elements such as toolbars and menus, capture user interface events such as mouse clicks, alter the way web pages are rendered, and provide support for additional network protocols.

Extensions have implications with respect to software maintenance. When a new version of Firefox is released, the interfaces used by extensions to communicate with the browser are often

modified. This means extensions are only compatible with certain versions of the browser, and must be updated frequently to maintain compatibility with newer versions. Extensions are typically developed in different source code repositories and by different engineers. This can result in overlapping functionality, and in some cases conflicting behavior among extensions. Finally, extensions have the potential to create security vulnerabilities in Firefox. For example, Greasemonkey is an extension that lets users add scripts to any web page to change its behavior, and some early versions contained a security hole that allowed malicious web sites to read any file on the user's hard drive. Fortunately, the extension developer was able to quickly release a temporary fix. However, had this not been the case, the Firefox developers might taken advantage of the fact that Greasemonkey is open source and created their own security patch.

SUMMARY AND OBSERVATIONS

Our reference architecture for web browsers identifies the key subsystems found in modern implementations, as well as the relationships between them. In addition to serving as a guide for maintenance and reengineering, it serves as a valuable framework for comparing browser implementations, both past and present. We have validated the reference architecture with five additional open source browsers and examined the results. Overall, we have found these implementations correspond quite closely, although there are several reasons why their architectures differ from our reference architecture. Some of the subsystems in the reference architecture may be combined as a single subsystem for simplicity, while others may be spread across multiple subsystems in the web browser for greater flexibility. New subsystems may be added to provide additional capabilities, while others may be omitted to make the browser more lightweight.

Table I shows various statistics about the different web browsers studied.[‡] Konqueror achieves nearly the same breadth of functionality as Mozilla with approximately one-quarter of the amount of code. Lynx, while smaller than the other browsers, is still five times larger than Links, a more recent text-only browser with a comparable feature set. We are unable to obtain complete size information for Safari because of its closed source components; the numbers shown correspond only to the open source parts.

The different approaches to code reuse and maintenance demonstrated by Epiphany and Safari are driven by the number of resources available to each project. Epiphany is developed by a small team of engineers, who work on the project in their spare time as volunteers. The "blackbox" approach to reusing Mozilla's engine allows Epiphany to take advantage of the efforts of the large number of engineers who contribute to Mozilla, some of whom are employed full-time to work on the project. Safari, on the other hand, has its own team of engineers employed full-time by Apple. The "whitebox" approach to reusing Konqueror's engine allows Apple to tailor the browser to its own needs, while at the same time manually propagate any

[‡]Data obtained using David A. Wheeler's SLOCCount[11] package.

Table I. Approximate web browser statistics

Project	Version	Language	Files	kLOC	Start
Mozilla	1.7.3	C++, C	10,700	2,400	1998
Konqueror	3.3.2	C++	3,150	600	1996
Epiphany	1.4.6	C++, C	7,230	1,540	2000
Safari	1.2	C++, Obj C	> 1,550	>230	2003
Lynx	2.8.5	C	200	120	1989
Mosaic	2.7b6	C	295	88	1993
Firefox	1.0	C++, C	10,700	2,400	2002

important changes from Konqueror when desired. Conversely, Konqueror is sometimes able to propagate changes made by Apple to its own engine.

The recent arrival of cell phone web browsers represents disruptive change in the evolutionary history of browsers. Previously, cell phones had monochrome, text-only screens and extremely small amounts of memory, providing a limited platform for browsing. The approach taken by wireless providers was to create WAP[22] gateways, which translated HTML into stripped-down markup that was displayed in a page-at-a-time fashion. However, as cell phone displays and interfaces became more advanced, cell phone browsers began to render the same HTML pages as regular browsers. Currently, many cell phones have high-resolution color displays and increased amounts of memory. This allows them to run stripped-down versions of desktop browsers, although the user interfaces are typically tuned to accommodate the smaller screen and limited input methods[19]. Opera has released a mobile version of its browser that includes many features traditionally found only in desktop browsers, such as bookmarks and password management; Nokia has released a browser based on WebCore that displays a thumbnail rendering of the whole web page and allows the user to zoom in on areas of interest.

RELATED WORK

Reference architectures have been proposed for other domains, including real-time train control systems[20], avionics[12], and web servers[26]. Product line architectures[18] are similar to reference architectures, although they generally represent a group of systems intended to be produced by a single organization, while reference architectures represent the entire spectrum of systems in a domain. Various aspects of Mozilla's architecture and development process have been studied[24][28][21].

Larrondo-Petrie et. al. have used object-oriented domain analysis to create a domain model, object model, and feature tree that describe the structure and functionality commonly provided by web browsers[27]. However, they do not extract the concrete architectures of browsers to refine their models. Chuang, et al. have conducted a quantitative energy-profiling study of Konqueror in order to examine the feasibility of their proposed profiling solution and aid in the possible design of an energy-efficient browser[17].

CONCLUSIONS

We have examined the history and evolution of the web browser domain, developed a reference architecture for web browsers based on two existing implementations, and validated this reference architecture by mapping it onto five additional implementations. We have analyzed the maintenance implications of the different code reuse strategies employed by these browsers. We have also observed several interesting evolutionary phenomena in the web browser domain; namely, *emergent domain boundaries*, *convergent evolution*, and *tension between open and closed source development approaches*.

As the web browser domain has evolved, its conceptual boundaries—both external and internal—have become increasingly more defined. However, there are still discrepancies as to the nature of these boundaries. Microsoft has claimed that Internet Explorer is a fundamental part of the Windows operating systems, providing rendering functionality to other applications. This has posed a problem for third-party browsers such as Netscape that sought to compete with IE. Similarly, mail, news, and ftp client functionality have been integrated into the Netscape and Mozilla browsers, discouraging competition from external clients. It will be interesting to observe how the web browser domain adapts to support embedded devices such as cell phones and PDAs, where limited memory makes it undesirable to deploy multiple competing applications.

The large amount of effort devoted to creating high-quality open source browser implementations has had a profound influence on the domain. During the "browser wars," proprietary extensions were added to core components in order to attract customers. Today, increased pressure to comply with standards has led to the general reuse of core components; the browsers are mainly differentiated by their user-level features. However, these features are often easily duplicated; for example, tabbed browsing and pop-up blocking were once innovative features but are now commonplace. These observations suggest that the web browser domain is exhibiting a form of *convergent evolution*[23] (*i.e.* species are independently evolving similar morphological features).

The availability of mature browser components has also resulted in tension between open and closed source development approaches. Mozilla's open source engine has been reused in numerous applications, both open and closed source. Similarly, Konqueror's open source engine has been used as the basis for Safari. As per the terms of the license, Apple has contributed its changes to open source components back to the community. Conversely, Internet Explorer represents a closed source engine that can potentially be embedded in an otherwise open source product. Netscape 8 strikes a balance by embedding both the Mozilla and IE engines, allowing users to switch on the fly. While we have seen applications composed of both open and closed source components before, the interaction usually takes place on the perimeter, as is the case with closed source binary modules for the Linux kernel. We believe the heterogeneous combination of core open and closed source software components within individual systems makes the web browser domain unique in terms of software maintenance and evolution.

ACKNOWLEDGEMENTS

An earlier version of this paper[25] appeared in the *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*.

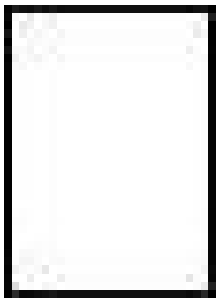
We thank Ali Echihabi for his contributions to an earlier project, as well as Ric Holt for his feedback and advice.

REFERENCES

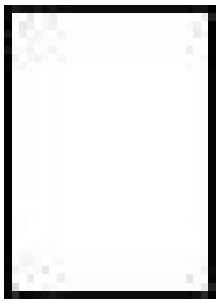
1. Acid2 browser test home page. `http://www.webstandards.org/act/acid2/`.
2. Apple developer documentation. `http://developer.apple.com/documentation`.
3. Cascading Style Sheets home page. `http://www.w3.org/Style/CSS`.
4. ECMAScript language specification. `http://www.ecma-international.org/publications/standards/Ecma-262.htm`.
5. Gnome desktop environment. `http://gnome.org`.
6. K Desktop Environment home page. `http://kde.org`.
7. Konqueror web browser home page. `http://konqueror.org`.
8. QLDX reverse engineering toolkit home page. `http://swag.uwaterloo.ca/qldx`.
9. Qt application development framework home page. `http://www.trolltech.com/products/qt`.
10. Safari web browser home page. `www.apple.com/safari`.
11. SLOCCount home page. `http://www.dwheeler.com/sloccount/`.
12. Don Batory, Lou Coglianese, Mark Goodwin, and Steve Shafer. Creating reference architectures: An example from avionics. In *Proceedings of the 1995 Symposium on Software Reusability (SSR'95)*, pages 27–37, 1995.
13. Tim Berners-Lee. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor*. Harper San Francisco, 1999.
14. Tim Berners-Lee and Dan Connolly. Hypertext markup language — 2.0, RFC 1866, November 1995.
15. Tim Berners-Lee, Roy Fielding, and H. Frystyk. Hypertext transfer protocol — HTTP/1.0, RFC 1945, May 1996.
16. Tim Berners-Lee, Roy Fielding, and Larry Masinter. Uniform resource identifier (uri): Generic syntax, RFC 3986, January 2005.
17. Chen-Ting Chuang, Chin-Fu Kuo, Tei-Wei Kuo, and Ai-Chun Pang. A multi-granularity energy profiling approach and a quantitative study of a web browser. In *Proceedings of the 10th International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'05)*, 2005.
18. Paul Clements and Linda M. Northrop. *Software product lines: practices and patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
19. O. de Bruijn, R. Spence, and M. Y. Chong. RSVP browser: Web browsing on small screen devices. *Personal Ubiquitous Computing*, 6(4):245–252, 2002.
20. Wolfgang Eixelsberger, Michaela Ogris, Harald Gall, and Berndt Bellay. Software architecture recovery of a program family. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, pages 508–511, 1998.
21. Michael Fischer and Harald Gall. Visualizing feature evolution of large-scale software based on problem and modification report data. *Journal of Software Maintenance and Evolution: Research and Practice*, 16:385–403, 2004.
22. WAP Forum. Wireless transaction protocol, version 10, July 2001.
23. Douglas J. Futuyma. *Evolutionary Biology*. Sinauer Associates, Sunderland, MA, USA, 3rd edition, 1998.
24. Michael Godfrey and Eric H. S. Lee. Secrets from the monster: Extracting Mozilla's software architecture. In *Second International Symposium on Constructing Software Engineering Tools (CoSET'00)*, June 2000.
25. Alan Grosskurth and Michael W. Godfrey. A reference architecture for web browsers. In *ICSM'05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 661–664, Washington, DC, USA, 2005. IEEE Computer Society.
26. Ahmed E. Hassan and Richard C. Holt. A reference architecture for web servers. In *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE'00)*, pages 150–160, 2000.

27. Maria M. Larrondo-Petrie, Krishnakumar R. Nair, and Gopal K. Raghavan. A domain analysis of web browser architectures, languages and features. In *Southcon/96 Conference Record*, pages 168–174, June 1996.
28. Audris Mockus, Roy T. Fielding, and James Herbsleb. Two case studies of open source software development: Apache and Mozilla. In *ACM Trans. Software Engineering and Methodology*, pages 11(3), 309–346, 2002.

AUTHORS' BIOGRAPHIES



Alan Grosskurth is currently pursuing an M.Math in Computer Science at the University of Waterloo under the supervision of Dr. Michael W. Godfrey. He graduated from the University of Toronto with a B.Sc. in Computer Science and Mathematics in 2004. His research interests include software evolution, software build systems, and computer security.



Michael W. Godfrey is an assistant professor in the David R. Cheriton School of Computer Science at the University of Waterloo. He is the associate chairholder of the Industrial Research Chair in Telecommunications Software Engineering sponsored by Nortel Networks, the National Science and Engineering Research Council (NSERC), and the University of Waterloo. He holds a Ph.D. in Computer Science from the University of Toronto (1997) and has also been a faculty member at Cornell University. His research interests include software architecture extraction and modelling, reverse engineering, software evolution, and program comprehension.