CISC 322

Assignment 2: Report

Google Chrome's Browser: Concrete Architecture

Friday, November 9, 2018

Group: Bits...Please!

Emma Ritcey *15er21@queensu.ca*

Kate MacDonald *14km90@queensu.ca*

Brent Lommen *brent.lommen@queensu.ca*

Bronwyn Gemmill *14bvg1@queensu.ca*

Chantal Montgomery *15clm1@queensu.ca*

Samantha Katz *12sk93@queensu.ca*

Abstract

Previously, the Google Chrome browser was investigated to determine its conceptual architecture. Using the derived conceptual architecture and analyzing the source code using Understand, the concrete architecture of the browser was proposed. After investigating the source code, two new subsystems were added: Utilities and Communication. As well, new dependencies were discovered and some predicted dependencies in the conceptual architecture were changed. The Render and UI subsystems were investigated in further detail to understand their dependencies. Using the proposed concrete architecture, the process of a user logging in and Chrome saving the password, as well as Chrome rendering a web page using JavaScript were explored in more detail. As well, team issues within Chrome and our own team were reported to support our derivation process and proposed architecture.

Table of Contents

Introduction

Google Chrome is an open-sourced web browser that was designed completely from scratch to evolve with the internet and meet the increasing needs of internet users. Throughout development, the Chrome team focused on simplicity, speed, security, and stability, which they referred to as the 4 S's. Their goal was to improve the internet as a whole which was why they decided to make Chrome an open-sourced project. They have made design documents, architecture overviews, source code and more available to the public so that they can share their development process with the community and help other developers improve their programs.

The goal of this report is to determine the concrete architecture of Chrome and perform a reflexion analysis between it and the conceptual architecture we previously derived in Assignment 1. After agreeing to stay with our original layered, 8 subsystem conceptual architecture, we used the program Understand to help us build the concrete architecture. During this process, we decided to add two new subsystems: Communications and Utilities, to take care of tasks that our original subsystems were not designed to perform. Once the architecture was finished, we determined it to now be Object-Oriented, as opposed to our Layered conceptual architecture. Looking at lower levels within subsystems, the object-oriented architecture style is also used.

Along with sharing our derivation process and analyzing the concrete architecture and the discrepancies between it and the conceptual architecture, we will provide further detail into the concrete architecture within the User Interface and Render Process. Following that, we will describe two Use Cases and how they interact with the various parts of the architecture to perform tasks successfully. Then, we will discuss various limitations we experienced as a team and lessons we have learned from this project.

Conceptual Architecture

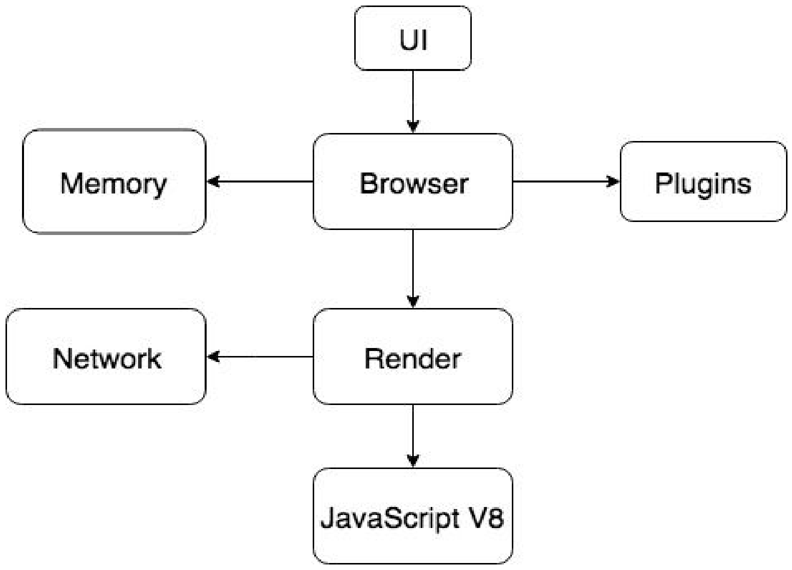

Figure 1. Conceptual architecture of Chrome proposed in the previous assignment.

Review of Subsystems

User Interface (UI): The UI is the subsystem responsible for the interaction between the user and the Chrome browser.
Browser: The main intention of the Browser Engine is to control the actions between the UI and Render. The browser engine is never responsible for rendering content from the web, this way the browser process is not affected by a crash during rendering. Instead, the browser is responsible for managing tabs, all interactions with the disk, user input and display.
Memory: The Memory component is used to manage and save data for a user on the user's disk. This includes saving bookmarks, history, cookies, cache of frequently/recently accessed pages, and all other data required to persist after the user has closed the browser.
Network: The Network component is used to connect to remote servers and fulfill URL requests given by the Browser Engine.
Plugin: Plugins provide additional features to the web page that the browser cannot handle itself, such as Adobe Flash and Java applets. They exist in their own subsystem for security and maintaining compatibility.
Render: The Render is responsible for controlling the contents of the tab or window displaying a website. The Render is specific to each tab or window, so if one tab crashes the other tabs will remain functional.
JavaScript V8: V8 was developed by The Chromium Project to interpret JavaScript. The JavaScript code embedded in the website to be displayed will be interpreted and executed by V8.

Derivation Process

To derive the concrete architecture of Google Chrome we used a software tool called Understand. Understand is a code analysis tool that allows you to visually analyze the dependencies of a code base. After importing the source code into Understand, we then created a new architecture where we created a component for each subsystem of our conceptual architecture. In addition to the components of our conceptual architecture we also added two new subsystems, Utilities and Communication.

Once we established the subsystems of our concrete architecture we had to analyze the source code and map the code to which subsystem it belonged to. For some of the larger subsystems, this mapping was done simply by looking at the source directories. For example the directory called “browser” was mapped to the Browser subsystem.

However, some of the code wasn’t as easily mapped. In these cases we would open individual classes and try to understand it’s functionality by reading developer comments and looking at the names of methods and which methods are being called. This analysis gave us a better understanding of the functionality and allowed us to map the code to the correct subsystem. After all the source code was mapped to the desired subsystem, Understand is able to analyze the code and create a graph where we can visualize and further analyze the dependencies between subsystems.

Concrete Architecture

Throughout our derivation process, we made sure that Chrome’s multi-process architecture and main principles were present in our Concrete Architecture shown in Figure 1. The Concrete architecture follows a similar high-level design as the Conceptual Architecture, whereby each Browser has a separate process and is isolated from any tab failures that may occur. Tabs are also run on separate processes and are managed by the Render Process, which operates within a sandbox, to insure overall security.

After adding all of our new dependencies, we concluded that Chrome has an Object Oriented design and is no longer Layered, as we previously determined in our Conceptual Architecture design. This is due to the high level of coupling that was produced when we mapped our new dependencies between subsystems, along with subsystems that connect bidirectionally, which are elements that are not present in a layered system. We kept all of the same subsystems from our Conceptual Architecture, while adding two new subsystems, Communication and Utilities, to handle additional requirements identified during our Understand analysis. We did not change any of the subsystems in our Conceptual

Architecture, as we determined these new subsystems were only an important aspect at a lower level after looking at the source code.

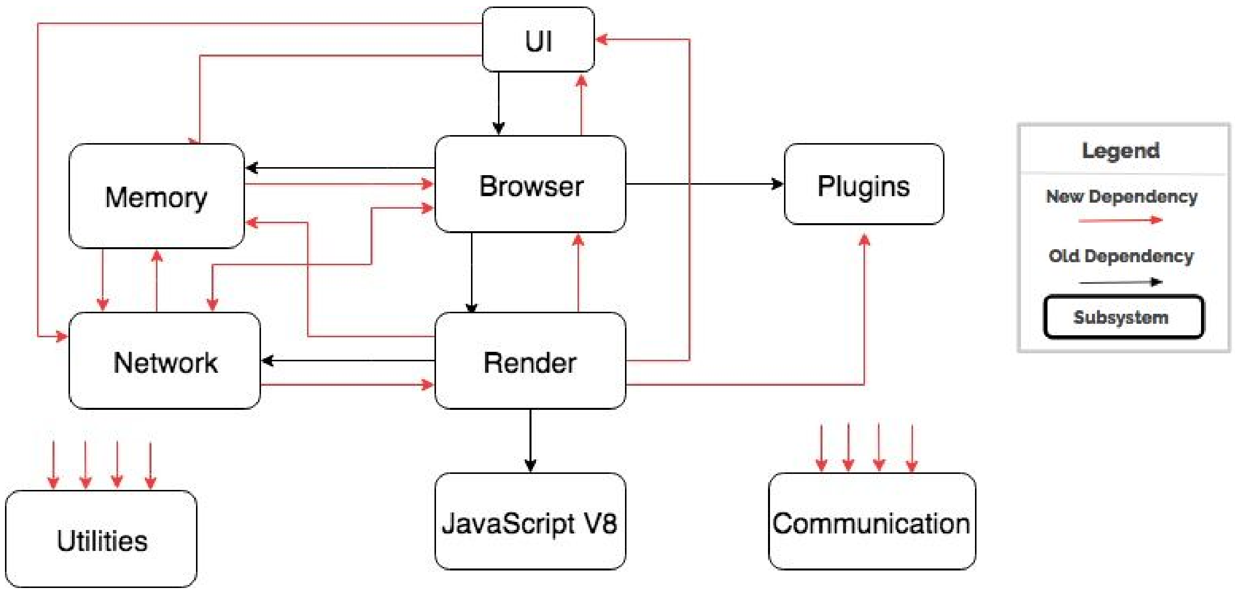

Figure 2. The proposed concrete architecture of the Google Chrome browser.

New Subsystems

Communication

This subsystem was created to encapsulate the concurrency control within the Chrome architecture. As Chrome is a multi-process architecture, it is important to identify a subsystem that can handle inter and intra process communication. The subsystem holds the source code for all elements related to communication between subsystems, processes and threads as well as the Mojo packages. Mojo is a system that allows cross-process and cross-thread communication by message passing using message pipes. This is a necessary subsystem in our concrete architecture, as it allows the concurrency module to reside independently outside of the browser, meaning it can facilitate any necessary communication between different Browser or Render processes. All subsystems are dependent on the Communication subsystem.

Utilities

While looking at the source code of Chrome, we noticed a package called “base”. This package contains all the common code shared between all of the sub-projects. It also consists of all the specific code for each operating system, including functions such as string manipulation and generic utilities. Because of this, we decided to add a Utilities subsystem to our concrete architecture, in which we placed the “base” package when doing the mapping of our architecture using Understand. In our architecture, we have every other subsystem depending on Utilities, which intuitively makes sense as it contains code for each of the other subsystems, and therefore they rely on the Utilities subsystem in order to function properly.

New Dependencies

Memory -> UI

When the User types a URL in the search bar, a feature of Chrome is to automatically begin auto-completing the URL following anything that has been previously accessed. In order for this to be possible, Memory must be able to determine what is happening in the search bar of the UI. From there, Memory can return a list of possible URLs that can be listed in the UI.

Memory -> Browser

Memory can store a snapshot file of the current Browser process along with the Browser's general memory consumption at that time. In order to save an accurate snapshot file of the process in Memory, it depends on what is currently happening in the Browser.

Network -> Memory

The Network allows the IO buffer to be streamed to Memory, but it relies on the current status of Memory. The Network depends on whether the Memory component is ready to accept a stream or not. This method allows for more efficient and optimal read operations for various chunks of data.

Network -> Render

The Render process stores a Network Interface List within the source code, which must be accessed by the Network. This list enables the Network to determine which configuration to take so that the Render will be able to properly access the internet according to its rendering specifications.

Browser -> UI

When there is a conflict or error in the Browser process, the user must be notified. In order for the Browser to display the error message, it depends on the current status of the User Interface. Once it determines the status of the User Interface, it can display the appropriate message. Generally, this is a fatal error, as it occurs in the Browser, not just the general Render process.

Render -> Memory

One reason this dependency exists in our concrete architecture is for security reasons. The render uses a security filter to allow for safe browsing for the user. If a website poses a threat, it will display a specific message that is stored in memory, based on the type of threat. Therefore, the render must communicate with the memory to gain access to the correct message it wants to display. For example, dangerous or deceptive sites are often called "phishing" or "malware" sites. When phishing and malware detection is on, the user may get a message such as, “The site ahead contains malware: The site you're trying to visit might try to install bad software, called malware, on your computer.”

Render -> Network
Looking at the source code, it was clear that the Render communicates with the Network subsystem to determine if the network connection is up and running.

Render -> Browser:
The Render communicates with the Browser as this allows safe browsing based on the type of browser. It also allows for message passing, specifically for Render-to-Browser messages which are called FrameHost messages. In the frame_messgaes.h package, hundreds of possible messages can be found that are sent from the Render to Browser. One example is FrameHostMsg_DidFinishDocumentLoad, which is a message used to notify the browser that a document as been loaded.

Render -> UI
The render extension uses the UI layout in order to render web pages as quickly as possible. A render tree is created to compute the layout of a webpage, which is then used as input for the paint process that the Browser uses to render pixels to the screen. This is just one of the steps in the rendering path, but ensuring it is optimized allows for content to be rendered to the screen as fast as possible.

UI -> Render
According to the Chrome source code, the UI must connect to the Render in order to play and display cc animations to the user.

UI -> Memory
This dependency exists as the UI communicates with Memory to make window resizing appear smooth. When resizing windows, it is possible for the UI to lag. However, if past window resizes are kept in Memory, and the UI is able to access them, it is able to load much faster for the user.

UI -> Network
The source code showed that the UI depends on the Network subsystem because the UI receives the URL and format of a website from the network.

Subsystems

User Interface (UI)

The UI subsystem was investigated to further understand its dependencies. This subsystem was not investigated in the previous report.

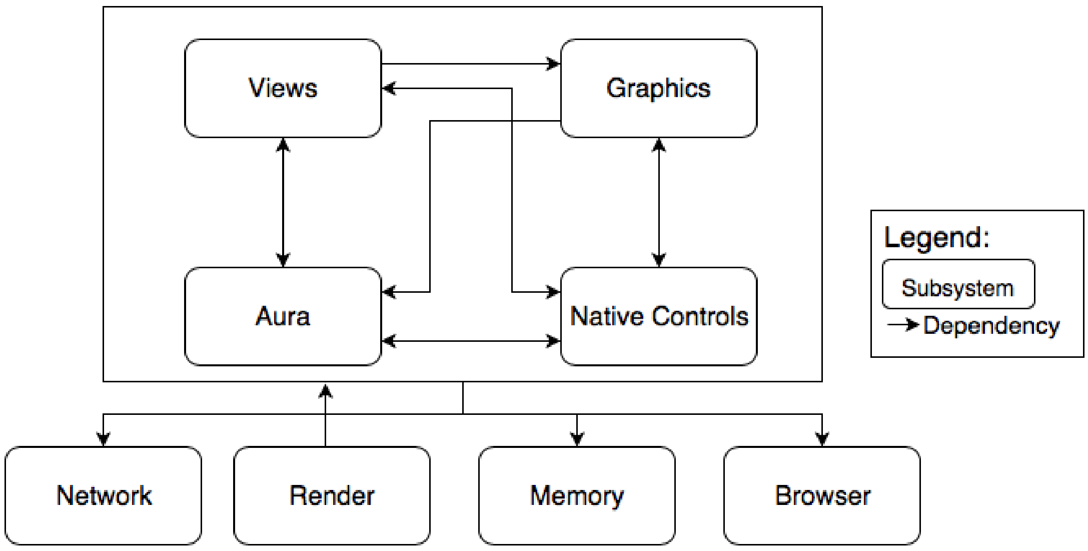

Figure 3. The proposed subsystem of the UI within the concrete architecture.

Views

The Chrome team wanted to create a unique UI which compelled them to develop a custom framework called views. There is a codependency between Views and Native Controls when the user interacts with the page like, touching a button. Views is listening to create background to show touch feedback.

Native Controls

Native Controls exists in the UI for componentes where Chrome's distinct and custom look is not necessary. It uses the host operating system's native buttons to reduce the need for updates every time a new version of the operating system is released. Along with its codependency with Views, as mentioned above, it also has a a codependency with Aura. This connection is necessary to register the changes in the boundaries of the page. Aura is needed to focus and ensure no content is lost.

Aura

Aura handles basic window functionality like focus and activation. It is very similar to Views but is much simpler. It handles low level events and passes them to their delegates. As described above it has a codependency with Native Controls. It also has a codependency with

Views. Aura handles a state change like a user clicking a button which then results in Views responding to the change.

Graphics

Graphics has co-dependencies with all subsystems in the UI expcet for Views and Aura. Views and Aura only depend on Graphics while Graphics does not depend on them. It depends on Graphics to get insets for the border, like size, colour, thickness, etc. It is needed by other subsystems to display Chrome's custom content, emojis, colours, etc.

Render

After the new concrete architecture was proposed the render was investigated again. Dependencies predicted in the conceptual architecture were further understood and new dependencies were discovered. The Blink source code was excluded from the provided files for simplicity.

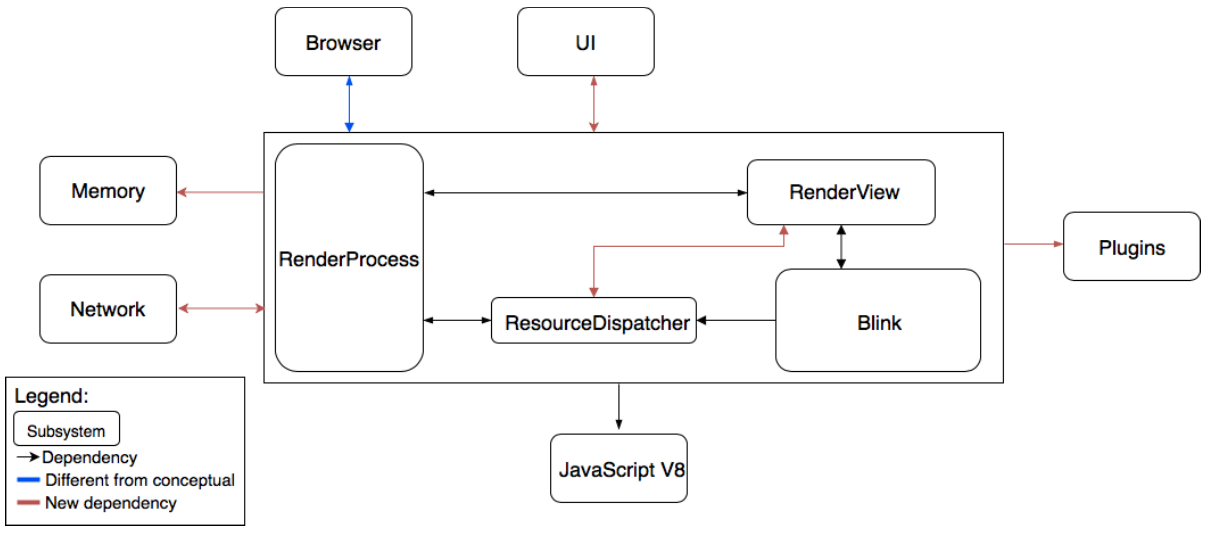

Figure 4. The proposed subsystem of the render within the concrete architecture.

RenderProcess <-> RenderView

This dependency was predicted during the conceptual architecture phase. The RenderView corresponds to the content of the tabs. It is managed by the RenderProcess which is why this dependency was initially proposed. After searching through the source code, it was discovered that this connection is used when updating screen display if the user were to zoom in or zoom out.

RenderProcess <-> ResourceDispatcher

The ResourceDispatcher is employed when a request is made to the server to fetch content. The request is sent via the RenderProcess to the browser. This dependency exists when ensuring that page content is displayed. It determines if the page resource data is complete along with ensuring the URL is loaded to completion. This dependency ensures all the content is displayed and to the correct sizing.

ResourceDispatcher <-> RenderView

This connection was not proposed in the original conceptual architecture. After using Understand and analyzing the source code it was determined that these subsystems are dependent on each other. This dependency was apparent when the Render wants to ensure there is a connection. For example, if navigation service does not return a suggestion or the loading time exceeds the limit then the local error page is displayed.

Use Cases

The following sequence diagrams will demonstrate how the Chrome browser works during two use cases. For simplicity, the calls to the Utilities and Communication subsystem have not been included. All subsystems (besides JavaScript v8) would need to access these subsystems during their execution in order to run shared internal functionality (Utilities) and communicate with each other (Communication).

Use Case 1: User successfully logs into a website, Chrome saves the password

The first use case starts with the user entering a URL into the Chrome UI to the website which they want to log into. The UI sends a message to the Browser to create a SiteInstance which represents a group of tabs from the same site which are running in the same process. The Site Instance is contained within the BrowserInstance which is created when Chrome is opened. The Browser will then send a URLRequest object to the network to tell it to make an HTTP GET request to to the server where the URL is pointing. When the resources are returned from the network, the Browser creates a new process for these resources, since a new SiteInstance has been created. To do this, the Browser will call the RenderProcess in Render to spawn a new RenderView. Render paints a window on the screen and receives user input and this is what makes up the WebContent that is returned to the UI.

When the user enters their information and submits the page, the UI makes a call to the Render through the function ForwardMouseEventToRenderer. Render registers this as the user submitting the form. When the user submits the form, PasswordStoreSigninNotifier registers the sign in event and passes it to the Browser to save. The password save and page request happen asynchronously.

Once the password event is sent to the Browser, the Browser uses the PasswordStore interface to save the password in memory. This is then returned to be OK and PasswordSaveConfimationView is used to return to the UI to alert the user that their password is saved in memory.

Once the mouse event has been sent to Render, the Render creates a unique request ID and forwards the request to the browser. The Browser then receives the request and converts it into a URLRequest. Since it is submitting data, the type of the request will be a POST request. Again, the returned resources are passed to the associated Render process and the WebContent is returned to the UI.

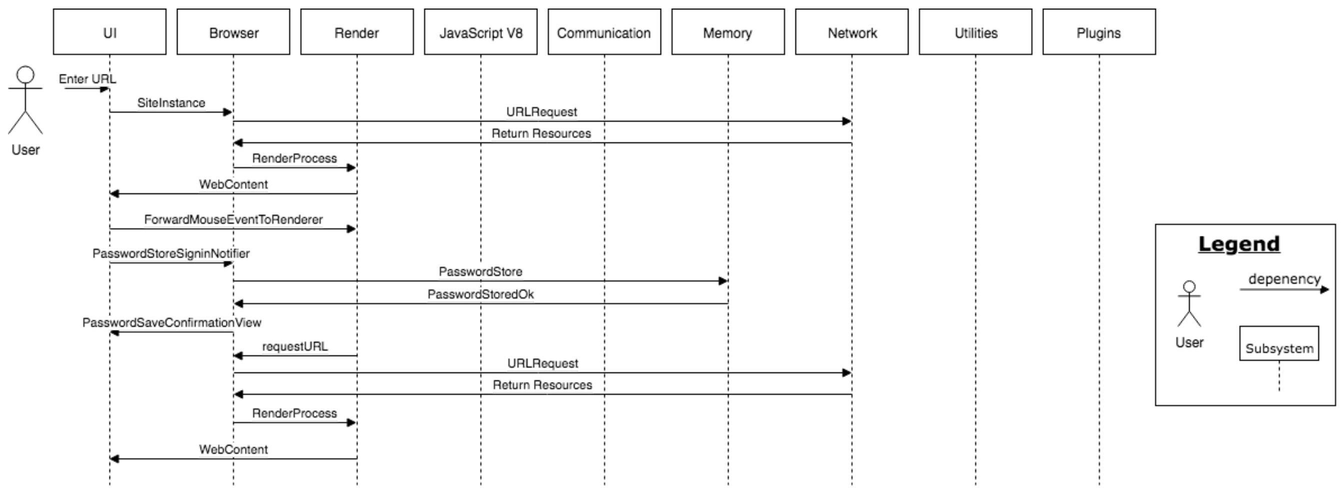

Figure 5. Sequence diagram for a user logging in then Chrome saving the password.

Use Case 2: Chrome renders a web page that has JavaScript

The second use case concerns a user requesting a web page which has JavaScript. This use case begins the same as the first with the user entering a URL into the Chrome UI. The URL is sent to the Browser and a new SiteInstance is created. Browser then sends the requested URL in a URLRequest object to Network. Once Network gets the resources from the external server, it returns them to Browser. Browser sends the resources to Render, and the RenderHost creates a new RenderView process for the tab. Since the page may take longer to load while waiting for the JavaScript to compile, the RenderViewProcess within Browser makes a call to the cache in Memory for a Backing Store. The Backing Store contains a bitmap of the last rendered version of the page. In this use case, we will assume that the user has not recently visited this page, so there is no backing store in memory.

Within Render, Blink has a ResourceLoader object to obtain the resources from the ResourceDispatcher. Blink creates the DOM tree and sends the JavaScript code to the JavaScript V8 subsystem to compile into machine code that the computer can understand. The compilation is returned to be OK so Render paints the window on the screen and returns this as the WebContent to the UI.

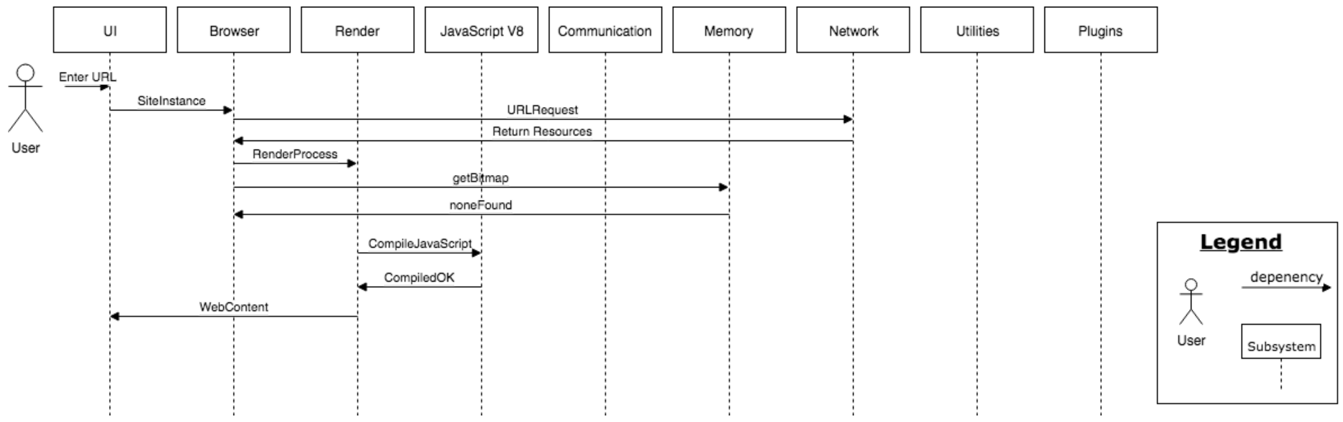

Figure 6. Sequence diagram for Chrome rendering a web page that uses JavaScript

Current Limitations and Lessons Learned

Due to the size and nature of a project like Chrome, it requires many different teams to work on different components. With so many different teams working on different components, our team found it difficult to understand much of the source code. Lack of commenting and consistent documentation made it difficult to trace dependencies. This demonstrated and reinforced for our team, the importance of documentation and commenting throughout coding projects for readability and understanding purposes.

During our investigation into the conceptual architecture of Chrome in Assignment 1, we learned about the importance of communication in every aspect of our project. This became even more important during Assignment 2. We had to communicate effectively to work together to trace the dependencies between components in our concrete architecture. As the Chrome project was so large, we found it most effective to work in smaller teams or pairs on different parts of the concrete architecture to create a final product that we were satisfied with as a team.

Conclusion

After a significant amount of time spent learning how to use Understand, our group worked through the various limitations we faced to construct Chrome's concrete architecture. Along the way we also added two new subsystems, Communications and Utilities, to take care of functions out of the realm of our original subsystems. There were many new dependencies in our architecture as well, which all had significant purposes. With these new dependencies, we determined the concrete architecture to be object-oriented, which differed from our layered conceptual architecture. After using Understand and examining parts of the source code of Chrome, we now have a much more solid understanding of how Chrome is constructed. This will prove to be beneficial for Assignment 3, where we determine a new feature to add to Chrome and how we would implement it.

Data Dictionary

Architecture: High level structure and organization of subsystems in a software.

Browser: Program used to display web content and allow for user interaction.

Cache: Allows faster access of recently/frequently accessed data by storing it in a smaller, faster memory.

Cookie: Data created by a website about a user to be temporarily or permanently stored in the user's device as a text file for personalization of web content.

JavaScript: Programming language used to create dynamic web pages.

Layered: Architecture style that organizes subsystems hierarchically.

Open-Source: Source code of the program is freely available to the public to be viewed and modified.

Subsystem: Component of a larger system.

Threads: Units of a program which are running simultaneously.

Network Interface: Point of interconnection between the system and public/private networks.

Naming Conventions

Document Object Model (DOM): Represents the web page as nodes and objects to modify the structure, style and content.

HyperText Markup Language (HTML): The markup language used to display contents of a web page. It controls the layout of the page.

HyperText Transfer Protocol (HTTP): The protocol used by the World Wide Web defining how messages are formatted and transmitted between the browser and a web server.

Interprocess Communication (IPC): A mechanism that allows processes and threads to communicate between one another. This can be unilateral or bilateral depending on the system.

Random-Access Memory (RAM): A type of computer data storage that stores data and machine code currently being used. Allows for quick read and write access to the storage device.

User Interface (UI): The subsystem in a software where the user interacts with the software.

Uniform Resource Locator (URL): Reference to the web address of a particular site.

References

S. (n.d.). Browser Market Share Worldwide. Retrieved October 12, 2018, from http://gs.statcounter.com/browser-market-share

Grosskurth and Godfrey. (n.d.). A Case Study in Architectural Analysis: The Evolution of the Modern Web Browser. Retrieved October 13, 2018, from http://research.cs.queensu.ca/~emads/teaching/readings/emse-browserRefArch.pdf

G. (n.d.). Design Documents. Retrieved October 12, from http://www.chromium.org/developers/design-documents

G. (n.d.). Google Chrome Comic Book. Retrieved October 11, from https://www.google.com/googlebooks/chrome/index.html

C. (n.d.). Threading and Tasks in Chrome. Retrieved October 13, from https://chromium.googlesource.com/chromium/src/+/lkgr/docs/threading_and_tasks.md

M. (n.d.). Mojo System. Retrieved October 13, from https://chromium.googlesource.com/chromium/src/+/master/mojo/README.md

Fisher, D. (2009, April 29). Retrieved October 19, 2018, from http://www.youtube.com/watch?v=A0Z0ybTCHKs

Plugin Architecture. (n.d.). Retrieved October 19, 2018, from https://www.chromium.org/developers/design-documents/plugin-architecture

Multi-process Architecture. (n.d.). Retrieved October 19, 2018, from https://www.chromium.org/developers/design-documents/multi-process-architecture

Core Principles. (n.d.). Retrieved October 19, 2018, from https://www.chromium.org/developers/core-principles

How Chromium Displays Web Pages. (n.d.). Retrieved November 9, 2018, from
https://www.chromium.org/developers/design-documents/displaying-a-web-page-in-chrome

Network Stack. (n.d.). Retrieved November 9, 2018, from
https://www.chromium.org/developers/design-documents/network-stack