Chrome's Concrete Architecture - Derived by uBlock Origin

Nicolas Merz, Zach Yale, Heather Geiger, Darrien Park, Kurt Blair, Dan Kailly

November 9, 2018

Abstract

Software source code is often vast and complex making the extraction of concrete architectures difficult. In this report, we derive the concrete architecture for the Google Chrome web browser to illustrate an understanding of the architecture of a system. Since there is no specific concrete architecture for Chrome published, we had to manually derive a representation from the provided subset of source code in Understand. We propose a combination of object-oriented and implicit invocation architectural styles and explain the key subsystems involved. Our chosen subsystems are the UI, Browser, Renderer, Network, Plugins, Base. Starting with an initial conceptual architecture, we detail the iterative steps involved in the Reflexion Analysis to derive a concrete architecture and a revised conceptual architecture. Based on our analysis of the source code in the Understand tool, we briefly discuss an interesting design pattern. We then provide an in-depth breakdown of the UI and the Network sub-subsystems to demonstrate dependencies within the module and interactions with other subsystems in the overall architecture. To illustrate our derived subsystem dependencies, we present a Sequence Diagram for two Use Cases in the browser; we show how a user logs into a webpage and Chrome saves the password and show how Chrome renders a page with JavaScript. We also discuss the Chrome browser's evolution. Finally, we touch on the limitations and lessons we learned while deriving Chrome's concrete architecture.

Contents

1 Introduction

In assignment 1, we derived a conceptual architecture was derived for the Chrome browser. The conceptual architecture was derived through conducting research into existing Chrome and Chromium documentation, developer notes, and blog posts. For Assignment 2, the team was tasked with using the Understand application to interpret a concrete architecture for Chrome, and re-evaluate the conceptual architecture derived in Assignment 1, while also iterating on previously derived deliverables given the new concrete architecture. This report will examine the team's interpretation of Chrome's concrete architecture, how the team derived it from the source code, and how it differs from our conceptual architecture. It will also examine a new conceptual architecture (that was created after the concrete architecture was derived), evaluating the differences between the new concrete and conceptual architectures. A Reflexion Analysis was conducted to derive the new conceptual architecture. An interesting design pattern (singleton) is evaluated and discussed, and an example from the source code is given. The team will examine the UI and Network subsystems in detail. For each subsystem, a revised conceptual diagram was considered, and a breakdown of the team's concrete implementation will be analyzed. Finally, UI and Network's revised conceptual subsystem will be compared to the concrete revision. Two use cases - user login and save password and user renders page with javascript - are evaluated with sequence diagrams. A comparison is made for both sequence diagrams between the concrete and conceptual diagrams. Finally, an analysis of Chrome's system evolution is provided, alongside some of the team's limitations and lessons learned.

2 Concrete Architecture

2.1 Derivation Process

Our derivation process for Chrome's concrete architecture first began in Assignment 1 when we derived a conceptual architecture. The information we researched in Assignment 1 along with our derived conceptual architecture helped guide the derivation of our concrete architecture. Keeping our conceptual derivation in mind, each group member began independently learning to use Understand. We then met as a group and decided to divide the group such that each group member was responsible for one subsystem. Seeing as the Chrome code is vast, we decided the best way to group files was by intended function from our conceptual architecture regardless of the new dependencies that arose. With consistent communication, we each grouped source code into our subsystems. We then compiled each subsystem into one master Understand file and ensured that no files existed within more than one subsystem module. The resulting architecture is our concrete architecture for Chrome. We then re-iterated over our conceptual architecture by removing all unexpected dependencies that we assumed to be hacks in the concrete architecture.

2.2 Proposed Concrete Architecture

We propose the architecture of Chrome to be a combination of Object-Oriented, and Implicit Invocation styles. Contrarily to Assignment 1, we no longer think it is layered because of the two-way dependencies between the UI, Browser and Renderer.

We propose the architecture is object-oriented because modules are encapsulated and invisible to other modules. Modules can make explicit calls to others. For example, the Browser Engine calls the Network module to access a web page's server to obtain confirmation of a user's login credentials. Furthermore, we think Chrome also uses Implicit Invocation in at least one situation. In Chrome, there are two primary types of messages sent from the Browser to the Renderer: *routed* and *control*. Control messages are sent from the Browser and do not specify a certain rendered process (not frame specific). Routed messages specify a process'unique ID. Typically just this one specific render process receives the routed messages. However, any class/process can receive routed messages by registering itself with the route using listeners (Inter-Process Communication - Chromium Design Documents). This newly registered process now receives the *routed* message that via their unique ID. A process registering itself with the message is advantageous for on the fly

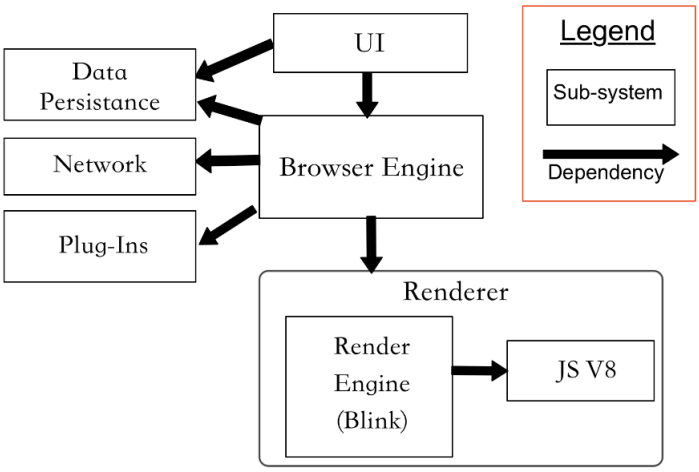

Figure 1: Initial Proposed Conceptual Architecture

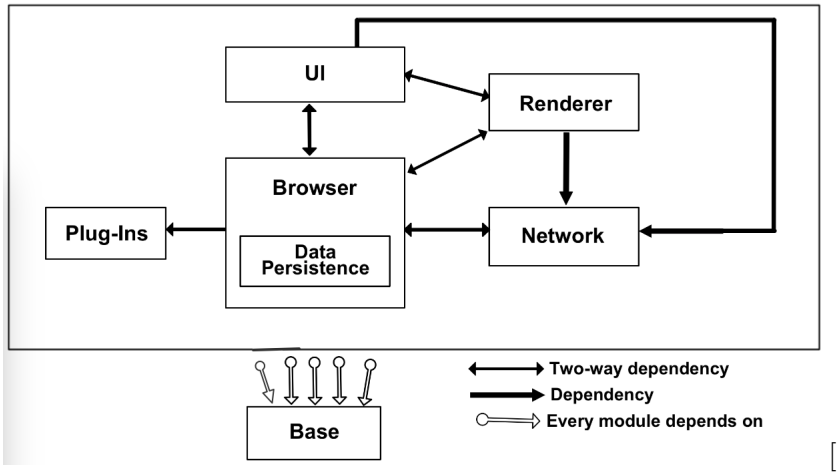

Figure 2: Proposed Concrete Architecture

configuration and reconfiguration of web pages. The Browser Engine acts as the broadcasting system and invokes all the processes that have registered themselves with the *event*, or message in this case (Garland and Shaw).

2.2.1 Subsystem Descriptions

- **UI:** The User Interface is the front end of the Google Chrome browser. It displays rendered HTML with CSS and any applicable JavaScript and Graphics content. The UI also provides access to inspect the HTML and JavaScript elements (Rahman, Rigby, Shihab) in the browsing window. Within the UI module is the UI backend that controls the makeup of the UI module, including tabs, the toolbar, and the bookmarks bar. It represents the views in the DOM tree. The backend depends on the Browser Engine to collect rendered content from the Renderer.

- **Browser:** The Browser Engine is the core of the Chrome browser. It connects to UI to the Renderer by acting as a communicator between the two. Two-way depedencies exists between these three aforementioned modules to allow the Browser to undertake its connective functionality. The Browser initiates all web page events like loading a URL, form submission, re-loading and navigating (Rahman, Rigby, Shihab). For this reason, the Browser also depends on the Network.

- **Data Persistence:** Data persistence is the module to store data, like passwords, browsing history, and

user preferences. We grouped Data Persistence in the Browser as we uncovered a lot of code overlap between the two modules.

- **Base:** The Base subsystem contains miscellaneous utility files important for Chrome's functionality. This module is a new addition to the concrete architecture. We attempted to split the code up into the other modules, but many strange dependencies arose, so we decided the best way to proceed was to create a new Base module. All other modules depend on Base.

- **Renderer:** The Renderer is the module that parses information contained in the URL. The Blink webkit acts as an interpreter for HTML and XML and formats according to CSS and XSL (Rahman, Rigby, Shihab). The Renderer contains the JavaScript V8 Engine to process JS contents. The Renderer interacts with both the Browser and the UI to parse content in the URL and transmit what to display on the webpage.

- **Network:** The Network module interfaces with the hardware to allow network connections and connections to other third party tools hosted on the network (Rahman, Rigby, Shihab). A network connection is required to access a web page's server. The Browser depends on the Network to find an internet connection in order to load a web page. The Browser Engine also calls the network to validate a user's login credentials, for example.

- **Plug-Ins:** Plugins are apps hosted on the user's machine like Adobe Flash, for example. The Browser depends on plugins as it will call it when a site instance requires access to a plugin. We kept the module even though the source code for Plug-Ins was not included in the Understand file.

3 Reflexion Analysis and Concrete vs. Conceptual Comparison

After we compiled each group members'allocation of files in Understand, and ensured there were no overlaps, we derived the concrete architecture shown in Figure 3 using Understand.

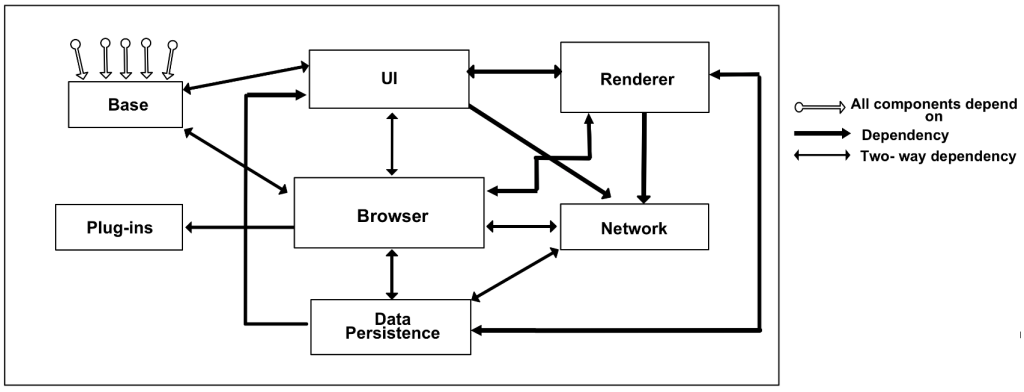

Figure 3: First Iteration of Concrete Architecture

This file had the unexpected dependencies shown in red in Figure 4 which we iterated over to investigate using the Reflexion Model technique. There was a good amount of unexpected dependencies and since we were still early on in the derivation process, we decided to start by double-checking the file allocations. We went through the file directories and found a number of folders that really did not belong in the current subsystem. For example, there was a Network services folder in Browser which we subsequently moved into Network. In this iteration we also decided to merge Data Persistence with Browser as there was a large amount of code overlap and file folders which we could not justify splitting up to keep independent modules.

After this iteration, we derived the architecture shown in Figure 2. The architecture in Figure 2 is our proposed concrete architecture for Chrome.

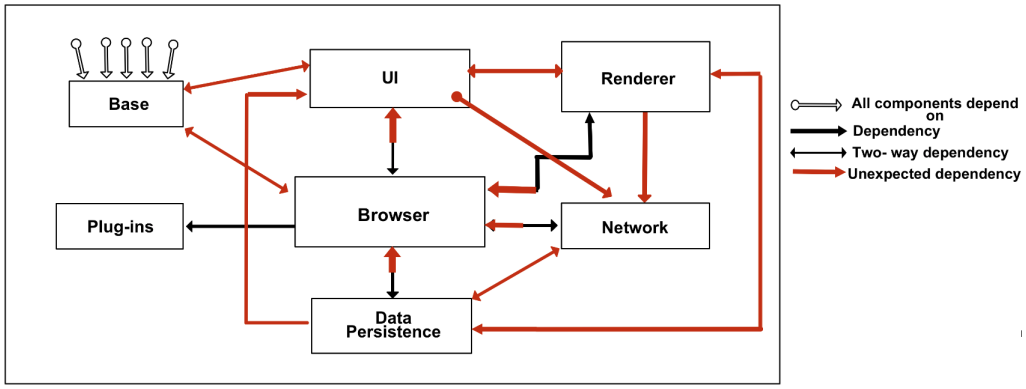

Figure 4: Unexpected Dependencies on First Iteration

We then decided to perform the Reflexion analysis once more to investigate unexpected dependencies in the concrete Architecture in order to derive a revised Conceptual Architecture shown in Figure 5.

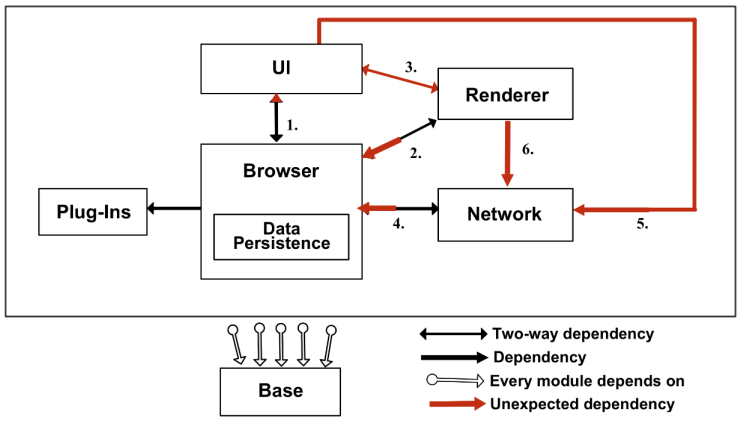

Figure 5: Unexpected Dependencies in Concrete

We analyzed each of the unexpected dependencies and concluded the following:

- The two-way dependencies between the UI and Browser (1) and between the Browser and Renderer (2) were there from the beginning, so they remain in our revised conceptual.
- The two way dependency of the UI and the Renderer (3) always existed to communicate between compositors and displayed rendered content.
- The unexpected dependency of Network on the Browser (4) existed from the beginning, so it remains in our revised conceptual.
- The UI dependency on Network (5) was deemed a hack because it consisted of 12 calls to an auxiliary OS-specific code to parse URLs.
- The Renderer's dependency on the Network (6) was deemed a hack because it consisted of URL loaders. We investigated and deemed that this was an unnecessary work around because the Browser is responsible for loading a URL and transmits this information between the Network and Renderer.

Figure 6 is our revised Conceptual Architecture and the result of the final iteration of our Reflexion Analysis after the hacks were identified and removed.

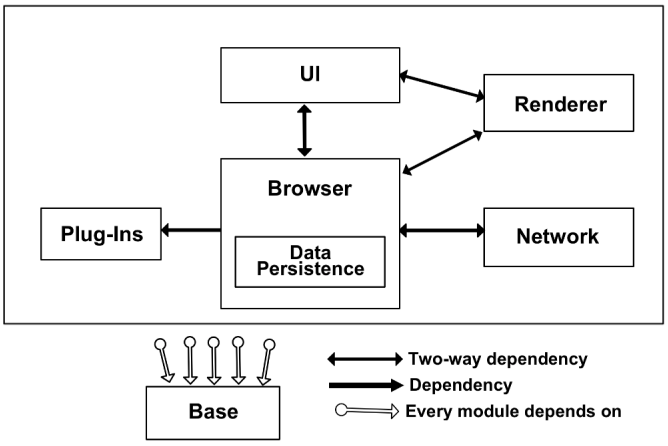

Figure 6: Revised Conceptual Architecture

3.1 Compare Concrete and Revised Conceptual

This section will briefly formalize what was mentioned earlier about verifying unexpected dependencies in the concrete architecture. We identified 6 unexpected dependencies and investigated each of them. The concrete architecture has 2 dependencies that aren't found in our revised conceptual architecture. These are dependencies that we deemed to be *hacks*. The three dependencies are UI to Network and Renderer to Network. The other 4 unexpected dependencies were also investigated, but we found that they are key to Chrome's functioning and that they actually existed from the beginning (we had overlooked these when deriving our conceptual architecture in Assignment 1).

4 Design Pattern

An interesting example of a design pattern in Chrome is the use of Singleton Classes in the Chrome source code. The singleton pattern is a software design pattern that restricts the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system. The concept is sometimes generalized to systems that operate more efficiently when only one object exists, or that restrict the instantiation to a certain number of objects (Wikipedia). In Chrome, for example, a singleton class is used to represent the changing of sandbox states on the Linux platform. This can be found in the source code file named *SandboxLinux*.

5 Sub-system Breakdowns

5.1 UI

5.1.1 Sub-subsystem interactions

The UI module has a two way dependency between the Browser and the Renderer and UI also depends on the Base module for key utility functions. The UI's interactions with the Browser allow for saving browsing history in Data Persistence and for Network connection through the Browser to display a URL. The UI also interacts with the Renderer to gather rendered webapge content to display.

In the revised conceptual, the internal architecture for the UI is Object-Oriented. Once the user interacts with the Display, by either clicking a web page or requesting browser history, the user's action is sent to the Event Manager and connects to the Browser to access updated URL data or any stored passwords or browsing history. If Display has new graphics to be rendered or if Event Manager calls Window Manager to update the display, the Renderer is called on to provide updated rendered content for the page.

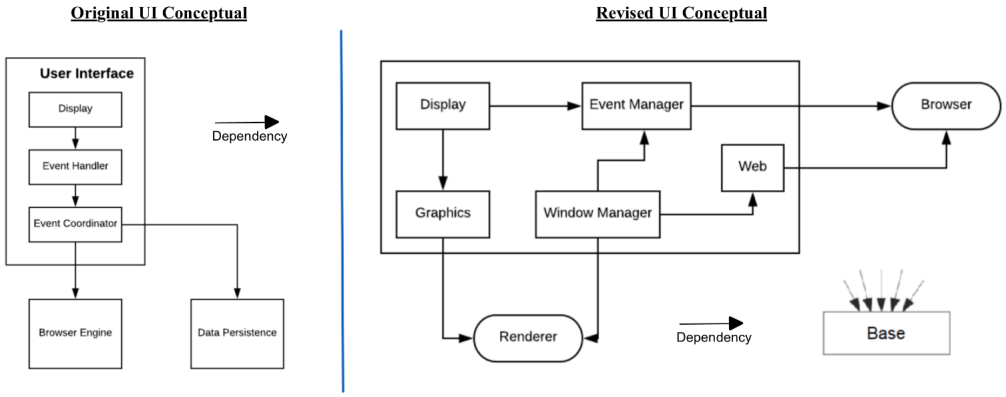

Figure 7: UI Conceptual Architectures: Original and Revised

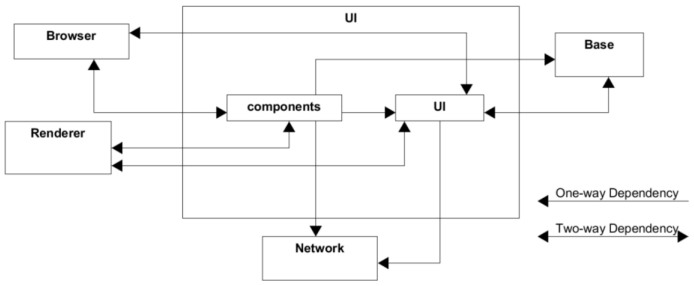

Figure 8: UI Concrete Architecture

5.1.2 UI subsystem architecture derivation

After deriving the UI's concrete architecture (Figure 8) from the source code, we iterated to revise the conceptual (Figure 7) as the concrete architecture uncovered various overlooked dependencies.

The concrete architecture (Figure 8) was created by examining the source code directories. In the revised conceptual, the UI folders were grouped into 8 directories and creating a new architecture with the updated dependencies. We found these directories to have the most essential UI functionality and allowed the most logical grouping of files to represent the subsystem. We made revisions to our UI conceptual architecture (Figure 7) to better represent the key dependencies revealed by analyzing Chrome source code. The updated conceptual now reflects the two-way dependencies between the UI and Browser and the UI and Renderer, and the UI's depedency on Base, all of which we uncovered in our analysis of the source code.

5.2 Network

5.2.1 Sub-subsystem interactions

The Network subsystem is a multi-platform library that uses mostly single-threaded processes when fetching resources for Browser. Instead of using an existing networking library, Chrome engineers developed the network stack to ensure greater control on overall performance, and to avoid bugs that could be present in existing libraries, such as WinHTTP. Along with general network utilities such as cookies and host resolution, Network also implements standard web protocols such as FTP and HTTP.

5.2.2 Network subsystem architecture derivation

After looking into design documents, we decided to go with an object-oriented style for our conceptual architecture. Considering that the code layout of the network stack shows a collection of different networking utilities, we thought it wouldn't make sense to classify it with other architectural styles. For example if we

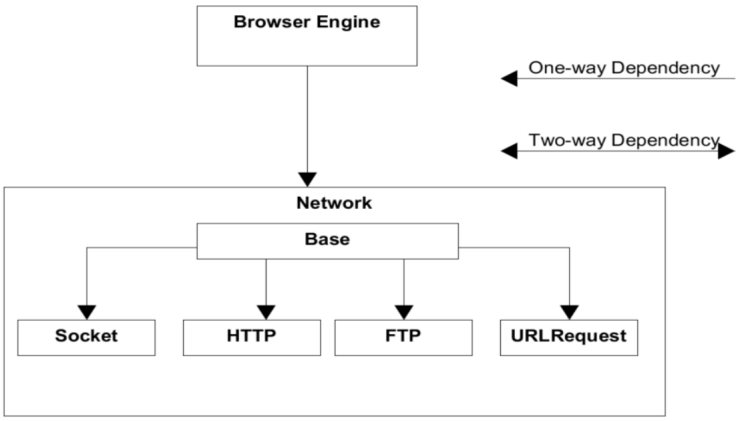

Figure 9: Network Conceptual Architecture

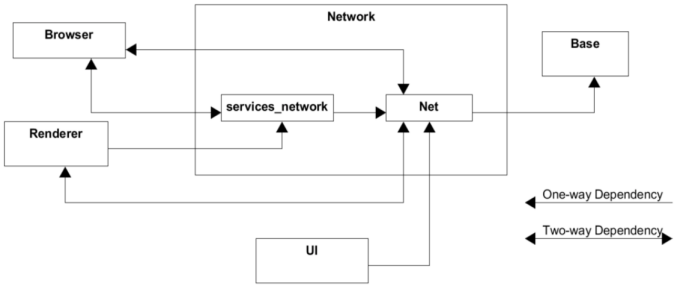

Figure 10: Network Concrete Architecture

chose a repository style, a brief inspection showed that the network stack is a library of single-threaded processes, and not a central data-structure with connecting procedure calls, so object-oriented is more applicable.

After looking through the Chromium source code in Understand, there were several revisions made when deriving the concrete architecture. We decided to place all of the subsystem components into the /net component, and introduced a /services component to be consistent with what we found in Understand. There also are several one-way and two-way dependencies between Network's components, and other Chrome subsystems (a notable example being files in /services having a two-way dependency with Linux sandboxing management in Browser). Because of the similar code layout found in our conceptual architecture, and the fact that Chrome's networking utilities can be found in the /net directory, our group didnt see much value-add in making the architecture more granular, or re-classifying the architecture style

6 External Interfaces

- **Cache:** Website information can be stored locally in a cache to reduce the amount of information that needs to be fetched remotely. The external interfaces used for this are the Backend (used for enumeration of resources) and Entry (used for operations for a given resource) interfaces. All cache information is stored in a folder called cache, and this folder contains an index file and four data files. The index file contains the hash table which allows Chrome to find the required data stored in those four data files. Chrome usually uses the memory cache first due to its high-speed and responsiveness, but will also store web page information into non-volatile storage (ie. the disk cache) in case of crashes or shutdowns.

- **Graphics Rendering:** Pages can be rendered using a GPU if it is available. The compositor (a part of

the Renderer) can interface with Chrome's GPU process to render pages. The compositor places its processed graphics into some shared memory (such as a bitmap), which can then be accessed by the GPU process and be supplied to the required graphics calls.

- **Operating System:** Features such as sandboxing and plugins leverage resources provided by the operating system to perform their required functionality. The architecture of sandboxing and the assurances that it provides is specific to each OS, however it utilizes the on-board security protocols to allow code execution that cannot permanently make changes to the computer or access confidential information. Plugins, on the other hand, utilize packages and frameworks installed on the OS to enhance or expand the capability of the browser. These resources need to be OS-bound because they are usually developed by third-parties and are often the cause of browser instability.

7 Use Cases

7.1 User Login, Chrome saves password

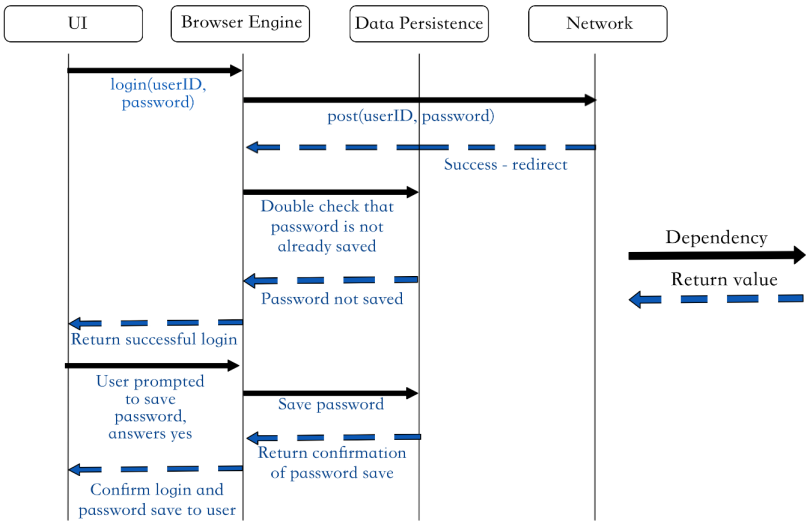

Figure 11: Use Case: Original User Logs In and Saves Password

The sequence diagram (Figure 12) demonstrates the control flow for a user logging into a website. The browser will handle the credential authentication, and upon success will save the password for future use to allow for features such as auto-fill and password management. However, a critical assumption is that this use case assumes the web page has already been rendered, and as a result, only the authentication and saving functionalities have been traced. It is assumed that the user has not previously entered a password to the site and that the user inputs their credentials correctly on the first attempt. Therefore, functions relating to updating previously saved passwords or handling incorrect credentials were not included.

The sequence begins with I/O events that are triggered as the user inputs their credentials into the fields of a login form. Upon the submit click event, the FormDataParser encapsulates the UTF-8 form data as a custom PasswordForm struct. This structure is a name-value pair, usually of type HTML, which is to be used for storage and to be interpreted by the rest of the browser. The renderer then submits the data to the browser module as a WebContent class, which holds all HTML render information for the web page.

The browser module extracts credential data from the WebContent class by using the GetWebContents-ForLogin() method. This creates a new object, called AuthCredentials, to be submitted to the network stack. The network stack contains a getter function to communicate with the web server and retrieve the desired AuthCredentials by using the username as a primary key. An equals method is used to make the authentication comparison and will return true if the credentials are a match. This prompts the browser to

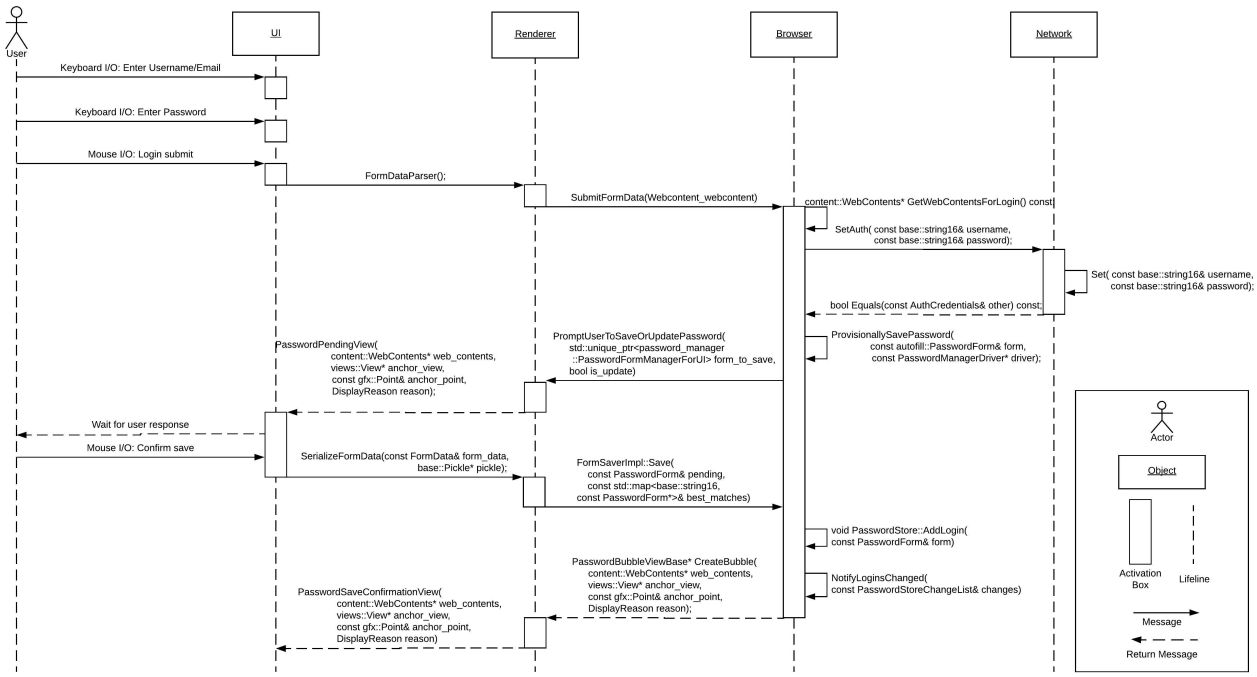

Figure 12: Use Case: Revised User Logs In and Saves Password

provisionally save the password in case the user would like to use autofill in the future. A save password prompt will signal the renderer to display a pending password view to the user and will wait for I/O.

Upon initiating a click event to confirm the save, the password manager will serialize the form data to allow it to be transferred to storage and the save method will relay the information from the UI back to the main browser. The browser will add the password form to the secure password store in the disk and the rest of the system will be notified of the changes once data modifications have occurred. Upon successful completion, a new bubble is created to prompt Renderer to display the password save confirmation view to the UI.

Note that there are some discrepancies between this report copy of Use Case 1 (Figure 11), and the version in our presentation. The presentation displays an extra module for data persistence. However, our latest iteration of the conceptual architecture includes storage as a subsystem of Browser. As a result, the object and lifeline have been removed from the final sequence diagram. Any calls made to storage are included as self-loops. In addition, the file names have been replaced with proper function calls and the appropriate parameters have been listed.

7.1.1 Comparison between sequence diagram iterations

In comparing the conceptual (Figure 11) and concrete (Figure 12) sequence diagrams for this use case, there are two main differences that can be identified. The first divergence is that the concrete architecture incorporates a Renderer subsystem, whereas for the conceptual architecture there was no Renderer involvement. The second divergence is that because the source code was used to inform the design of the sequence diagram, the parameters/function calls differ between diagrams. Another result of the source code being used to inform the design of the concrete sequence diagram is that the more minute details regarding which calls and returns are made differ slightly, as a result of the team discovering the specific order in which these functions are called within the code. Aside from these two aspects- Renderer being added and the parameters/function calls changing- the overall logic of the sequence diagrams remains the same between the concrete and conceptual use case.

7.2 Render a page with JavaScript

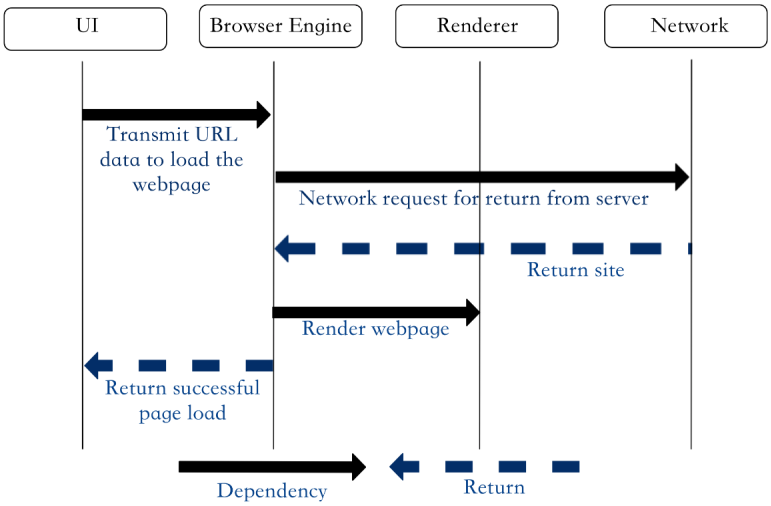

Figure 13: Use Case: Original Render Page with JavaScript

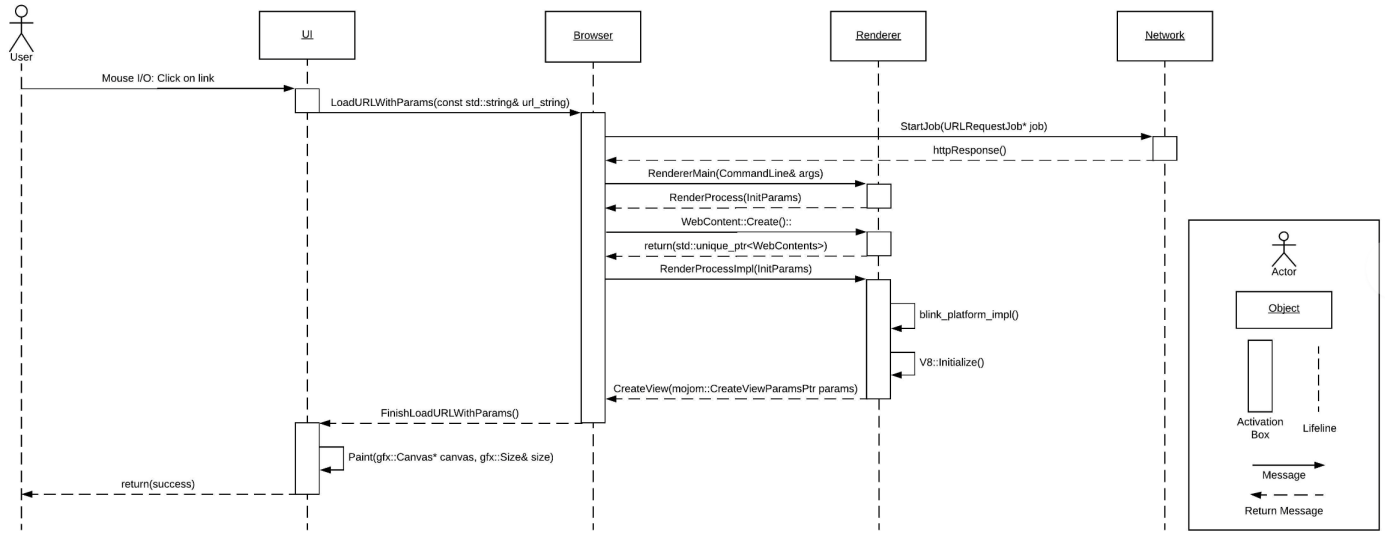

Figure 14: Use Case: Revised Render Page with JavaScript

This sequence diagram describes the control flow for a user navigating to a page on a new site that contains Javascript, and the subsequent rendering of this page. The only considerable assumptions being made are that there is only a single view being rendered and painted to, that single elements are not being refreshed using utilities such as AJAX, and the page being opened is on a different site than the one the clicked link was on.

The sequence begins by having a user click on a link to a new site. This prompts the UI to send the URL of the opened page to the Browser through LoadURLWithParams, which is handled by the navigationmanager, passing it as a string. Browser initiates the HTTP request through the network stack by creating a URLRequestJob through the StartJob method.

Chrome's network stack makes the request by interfacing with the operating systems network stack. It takes the response and returns it using httpResponse. With this data, Browser can send it to the Renderer. First, a new Renderer must be instantiated (as the assumption is that the clicked link is to a new site) so that the Site Isolation model, which states that only tabs on the same site can be located in the same process, can be upheld. This is done through the RendererMain call, which returns a RenderProcess.

Next, the WebContents::Create method is called. WebContents is the main object responsible for rendering web content (usually HTML). Once this has been instantiated, the RenderProcessImpl method can be called, which initiates the rendering process.

There are two self calls done in Renderer. Blink and V8 code was not supplied, so while the exact code could not be analyzed, some basic function calls were still included as these are two incredibly important subsystems of Renderer. Renderer then calls CreateView to return its created view to Browser. Finally, FinishLoadURLWithParams is called to complete the UI's initial Load URL request. Then, a paint call is made from the views subsystem of UI to the UI's compositor subsystem, rendering the page to the user.

7.2.1 Comparison between sequence diagram iterations

In the concrete sequence diagram (Figure 14) for rendering a page with JavaScript, there are only two major differences when comparing the diagram to the conceptual diagram (Figure 13). In the concrete use case, more interaction occurred between the browser and renderer than anticipated, and we did not account for the V8 and Blink self-calls within renderer. Similar to the first use case, access to the source code allowed for the team to inform the parameters and function calls used within the diagram to correspond with the actual ones within the code. While the same subsystems are used in the conceptual and concrete sequence diagrams, these changes made between the conceptual and concrete diagrams radically change the rendering flow for a page containing JavaScript.

8 Chrome Evolution

Chrome has gone through many iterations since its first beta release in 2008; moving from an unstable, barebones browser to an incredibly robust one, and becoming the most popular browser in the world. Chrome's first official, stable cross-platform release (Chrome 5.0) was in May of 2010. Since then, the Chrome team has taken user feedback to implement new features and perfect existing ones, improving the usability and performance of the system. The incredibly popular Extensions feature did not exist before the release of Chrome 5.0. The earlier stages of Chrome's development also saw the release of form auto-filling, the Chrome Web Store, and password syncing. Some large changes later on include encryption for all syncing in 2011, the release of Android and iOS clients in 2012, and the change from WebKit to Blink in 2013.

As more resources pour into Chrome, the number of files has increased drastically, representing its increased rate of development. This can be seen in Table 1 by the growing number of major releases year over year.

Year	Chrome Release	Number of Files
2010	5.0	7200
2011	13.0	10200
2012	22.0	17200
2014	34.0	28500

Table 1: Evolution of Chrome

Chrome has seen large changes in terms of modularity and cohesion. In earlier releases, many features spanned multiple modules, which lead to high coupling and low cohesion (some spanned more than 10 modules). However, in release 34.0, Chrome developers attempted to remove all dead code. This reduced how many modules certain features touched. In the dead code, there were a lot of unused feature toggles which were not even fully implemented, but since their code was in the codebase they created artificial dependencies between modules. In release of Chrome 34.0, features now span between 2 to 5 modules, which allows module changes to be less complicated (Rahman, 2018).

9 Our Limitations and Learnings

9.1 Limitations

The primary challenge encountered by our group for this assignment was the time constraint. Although we had originally developed a clear time line and allocated enough time for the completion of tasks, we quickly realized that the deriving the concrete architecture and delving into the source code to find specific function calls for the use case sequence diagrams required far more attention than we had originally anticipated. Even though we only examined a subset of Chrome source code to derive our concrete architecture, the Chromium code browser still included test cases, which increased the amount of code to search through. We found it challenging to group the file directories under our subsystem modules and tough find the correct specific function call for our use cases. The code was more vast and complex overall than we initially anticipated, and this, in turn, very clearly reflected the Funnel of Uncertainty for a project as our expected completion times vastly differed from the actual.

Furthermore, there was a very apparent lack of readability within the Chromium source code. In addition to the poorly organized directories, not all file folders contained proper READMEs or complete commenting. And while every team member had prior C++ knowledge, it was still quite difficult to understand the data flow and determine which functions were relevant to our specific use cases.

Finally, we also felt that a limiting factor was the Understand software. Although Understand is a tool commonly used in industry, we feel we were unable to use it to its full potential. We think it would have been helpful to have more tutorials demonstrating the use of the tool in order to fully grasp the functionality of the software. Also, the limited processing power of our machines and the fact that Understand is a very resource intensive software meant that loading architectures, saving changes, and opening files often resulted in a crash or a stall.

9.2 Lessons Learned

Despite the limitations we encountered, there are a few key takeaways from this assignment that would be greatly beneficial for future projects. Most importantly, we have learned that mature software systems are much more complicated than they seem and have many unexpected dependencies. Compared to our original conceptual architecture which was highly layered, we found it very difficult to conclude one-way dependencies and learned complexity only increases as updates are continuously applied. Even for simple tasks, there may be many unexpected calls between modules to properly function. Furthermore, we learned the significance of communication and delegating subsystem-specific tasks within a team for a project with such a large scope. Good team organization was especially important when compiling the final architecture and conducting comparison analyses.

10 Conclusion

We concluded that the concrete architecture for Google Chrome is a combination of object-oriented and implicit invocation architectural styles. After using the source code to develop the concrete architecture, the conceptual architecture was revised, to be more closely align with the concrete architecture. Such changes included adding relationships between subsystems that were not accounted for originally. A Reflexion Analysis was used to verify unexpected dependencies. The singleton design pattern was identified in Chrome's source code; an instance of it exists within Chromium's sandboxing process, for example. The UI and Network subsystem architectures were re-constructed using the source code to derive a concrete architecture. In these subsystems, several new two-way dependencies were recognized that were not expected in the first conceptual report. We also refined our use cases sequence diagrams to account use explicit function calls found in the source code. This allowed us to create more accurate use case flows for the concrete sequence diagrams in comparison to the conceptual architecture.

The team encountered a major time restraint in deriving the concrete architecture. We found the scale

and complexity of the Chromium project's code base to be daunting. Despite these challenges, the team managed to improve their communication and delegation skills throughout the time spent working on this assignment. As a result, these improved delegation skills allowing the team to develop a robust concrete architecture, detailed subsystems, and better-informed use case diagrams on time.

References

[1] A. Neupan, "The evolution of google chrome," 2013.

[2] T. Rahman, P. C. Rigby, and E. Shihab, "The modular and feature toggle architectures of google chrome," 2018.

[3] na, "The chromium projects - design documents," 2009-present.

[4] na, "Threading and tasks in chrome.," 2009-present.

[5] na, "Sandbox design in chrome.," 2009-present.

[6] A. E. Hassan, "Module 5: Reference architecture (web servers and web browsers)," 2018.

[7] D. Garlan and M. Shaw, "An introduction to software architecture," 1994.

[8] na, "The chromium projects - plugin architecture," 2009-present.

[9] A. Boodman, "How chromium works," 2015.

[10] na, "The chromium projects - teams," 2009-present.

[11] na, "The chromium projects - getting around the chromium source code directory structure," 2009-present.

[12] P. Barth, S. Felt, and A. Boodman, "Protecting browsers from extension vulnerabilities," 2009.

[13] Wikepedia, "Singleton patter," 2018.