CISC 322 Assignment 2 Report

Inspect Element:

Roberto Ruiz, Matthew Pollock, Maxwell Keleher, Joseph Gravenor,
Jack Guinane, Jonathon Gallucci

Abstract

This report is a presentation of our analysis of the source code Google Chrome web browser, which was developed as part of the Chromium open-source project. Our mission was to develop a concrete architecture of Google Chrome using the source code. It is an object oriented architecture style. In this report we will investigate the discrepancies between our initial conceptual architecture and the derived concrete architecture. In the previous report we described the Blink subsystem. We did not have access to the Blink source code so we opted to describe the Concurrency subsystem instead. This report will also contain 2 sequence diagrams, team issues and our lessons learned.

Concrete Architecture

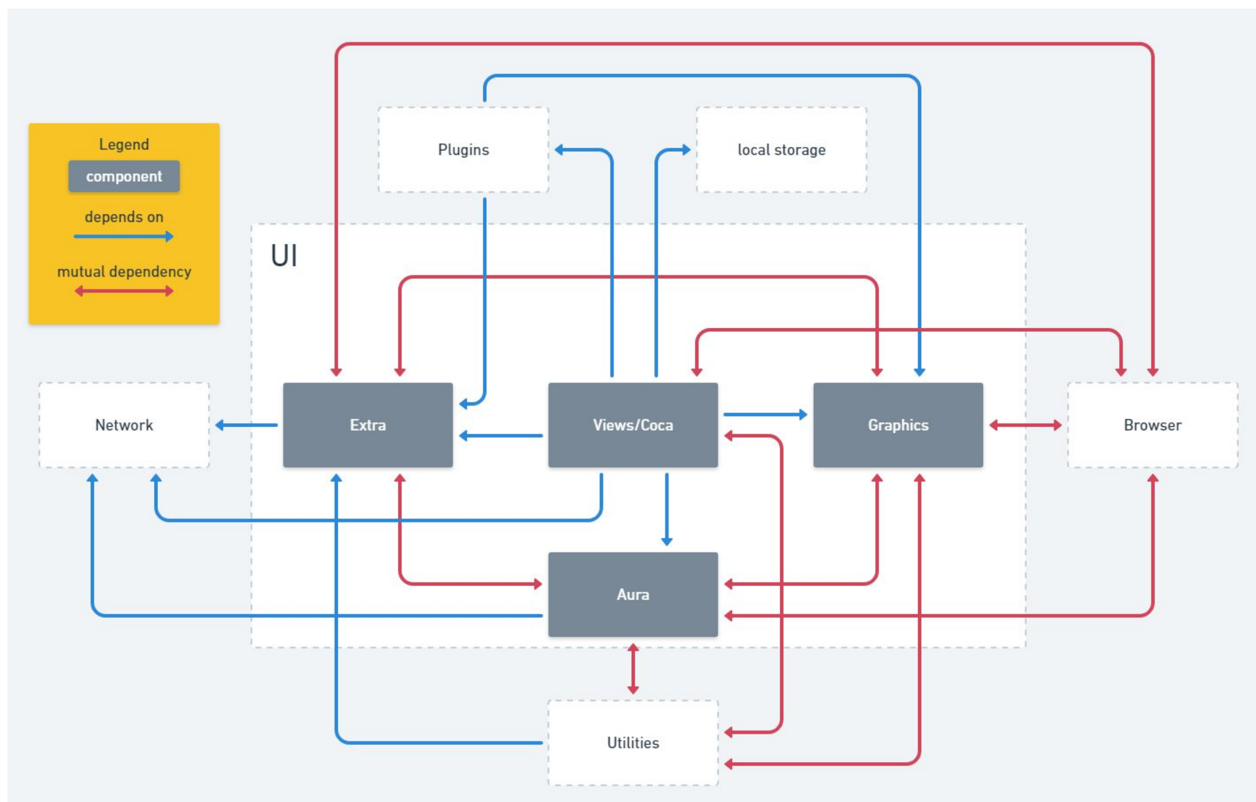
This is the concrete architecture that we created using the files given to us as a barebones version of Chrome. The directories' files were sorted based on the relation and services they provided. This methodology of categorizing proved to increase some cohesion of the sorted files, but mostly increased the coupling between the components they have been sorted into. From our conceptual architecture, we had low coupling and high cohesion between the components, and now in our concrete, we have high coupling and low cohesion. Furthermore, we initially proposed a loosely-layered architecture but the mutual dependencies break the rules of a layered style. Browser, for example, has co-dependencies with every single component in this system. In our conceptual architecture, Browser only depended on two

components, and had two others depending on it. Another thing to note between our conceptual and our concrete architectures, is the fact that our concrete now has two more components (Concurrency and Utilities) which are discussed further in detail in this report. We believe this concrete architecture best reflects the actual architecture of chrome with the given file structure.

Derivation

Our derivation process begun with an investigation into the source materials provided along with the visualization of the code's dependencies using Understand. We looked into each directory, sub-directory, and sub-sub-directory, viewing the files contained within each. Comparing the method headers, comments, README files, and online resources we determined the purpose of each file and sub-directory. We then moved these into the corresponding subsystem of our conceptual architecture. This optimization for cohesion came with a price in the form of coupling. Our first attempt had each element fully connected within the system. To reduce coupling, we changed our approach. Instead of dividing up each directory into its smallest pieces, we used the knowledge we had gained from our first attempt and looked at what the purpose of each directory was as a whole. We moved the directories as units, prioritizing the structure the Chromium team had developed over higher cohesion. Although lower cohesion, we believe that these changes better reflect the actuality of Chrome's concrete architecture. The result of our second attempt was an architecture that had much less coupling and better represented the way in which Chrome was built. We also decided that we would require two new subsystems (concurrency and utilities). The reasoning behind this decision is provided in our break-down of each subsystem, later in the report.

UI subsystem



New UI subsystem:

Differences between original subsystem:

I had an implicit invocation arrow going into the ui architecture in the first assignment to explain that it listens for os messages, but not in the second assignment. I did still say aura listens to OS messages, but I found no documentation saying how it did this, so I didn't include the implicit invocation arrow. I decided that widget os should be changed to aura because it was more accurate to how the concrete architecture actually is. In the conceptual architecture, I had views pulling double duty as both the UI framework and the simpler window managing that happens in aura. For cohesion sake, the more aura specific aspects were moved to aura. Before I had much of the graphics component split up into skia and GDI, with the compositor being in views, but there were more components dedicated to the compositor than I had originally thought, so it made sense from a cohesion standpoint to put them into A new component called graphics. Skia and GDI are now in this Graphics component, because chrome treats the 2 engines as external libraries and only includes their function calls in the chrome source code.(1 & 2) There is a new component called extra. Extra holds base, which according to the documentation “provides random utilities for building the UI”. This isn’t very cohesive, but it is for functionality.(2)

Violations of the architecture:

The file_maneger does not belong in UI, because no other component in UI depends on it. In fact, no component depends on it and it seems to be a hack to get google drive to work.(1) Snapshot is taken

out because it is just for debugging.(2) Aura should not depend on views, however it does. This is because of src/ui/views/widget/desktop_aura, which was aura before they decided to make aura its own subsystem. There are only 2 files within aura that depend on this directory. These files are device_list_cache_x11.cc and x11_event_source.cc.(2) Both depend on this directory for getting the display device, meaning they never got around to implementing this feature in aura, and is a hack as it relies on a different implementation of aura.

What everything does:

Aura:

Aura hosts the native platform view which hosts the view hierarchy(does this by having a DesktopHost that wraps an HWND and one that wraps an X window)(5 & 4). Handles the window delegate tells the window how to look when it is created and responds to I/O performed on the window(s)(4). Aura handles specific window manager functions such as: constraint-based moving and sizing, shell features such as the persistent launcher at the bottom of the screen, status areas, etc(4). Handles OS messages(3). Aura is responsible for the Window hierarchy, event cracking and propagation, and other basic window functionality (like focus, activation, etc).(4)

Views/Coca:

With the messages views gets from Aura, views constructs relevant views:events to propagate to the view hierarchy(4). “interface framework built on a type called, confusingly, View. Responsible for providing the content of our Aura windows”(6). The view hierarchy provides content for the UI(buttons menus etc), and event dispatch that happens on those elements(6)

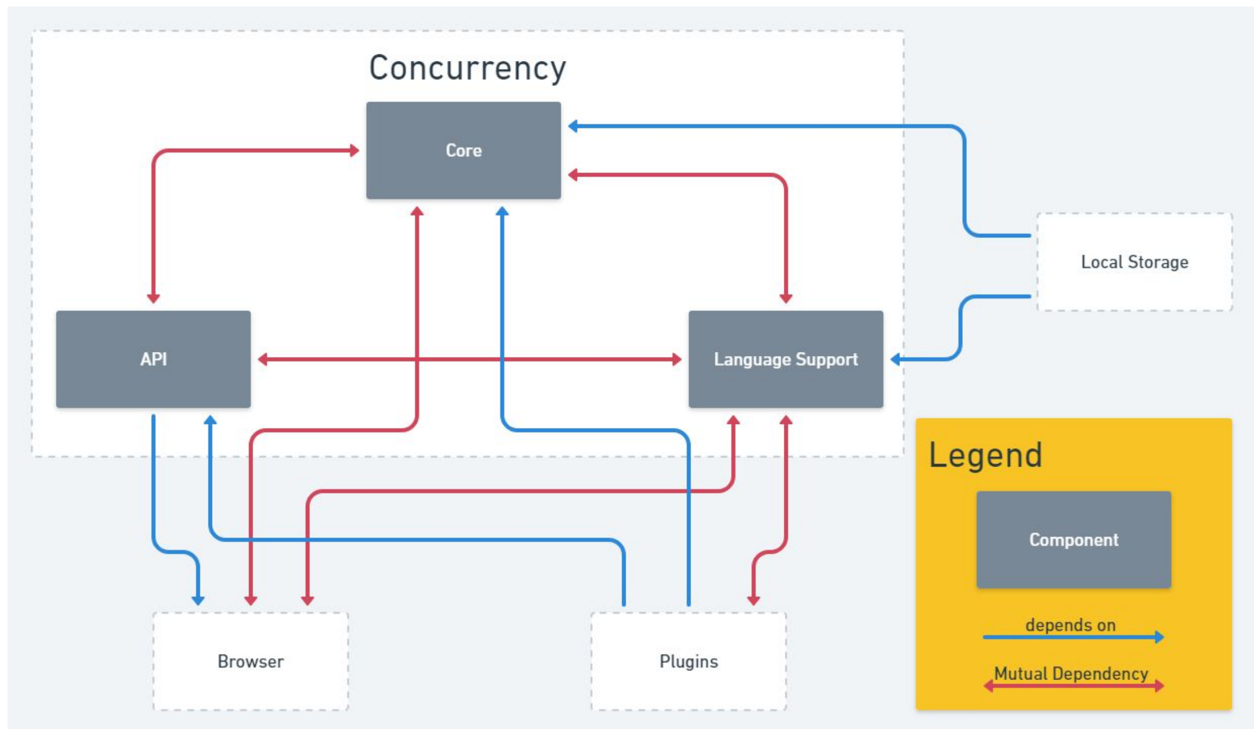
Base:

“Random utilities for building UIs”(2). Includes factories, templates, and other miscellaneous functions that don't fit anywhere else(2)

Graphics:

Graphics handles calls to skia and gdi, which are not present in chrome, but are third party components(2). Graphics also contains the compositor which is responsible for drawing to screen(1). Handles canvas, animation, geometry, etc.(1)

Concurrency



The Concurrency subsystem is the component that creates and manages concurrent operations, mainly through the Mojo framework. Our previous report noted that Chrome's model of concurrency required a "centralized controller," which we had assumed was part of the Browser subsystem. However, in our investigation into Chrome, we found this to be done within its own subsystem. The reasoning behind placing Concurrency in its own subsystem is so it can be accessed by all the components that require concurrent operations, instead of going through the Browser subsystem to do this. The subsystems that now depend on Concurrency are Browser, Plugins, and Local Storage. The processes in which Concurrency fulfills these subsystems' requests are still very similar to the ways we originally outlined in our first report's concurrency section.

We chose to investigate further into the Concurrency subsystem as one of our two subsystem architectures. Mojo Core contains the high-level support libraries such as System APIs and Bindings APIs. The Language Support component is contains helper classes for C++, JavaScript, and Java. The API component contains the foundations for any higher-level Mojo APIs. It allows for the create of primitives such as message pipes, data pipes, and shared buffers. It also contains APIs to bootstrap cross platform connections.

The Concurrency subsystem depends on Plugins and Browser. The dependency on Plugins is for its service manager subsystem, used for bindings, interfaces, and execution of code. The Browser dependency is for backend operations, such as threading and bindings. The Core component depends on Browser for memory, debugging, event tracing, and threading. The API component depends on Browser for logging, memory, and threading.

Utilities

The Utilities subsystem, which was originally presented as ‘Components’, contains the extended features of Chrome built by the Chromium team. This includes the Password Manager, Bookmarks, favicon, and more. Our first report suggested the existence of these features and claimed that, like third-party extensions, they would assess Chrome through the Plugins component (see our first report’s ‘Plugins’ section for a diagram of this interaction). We neglected to include these feature into any subsystem, and have corrected this mistake. These features are now bundled together in Utilities, and (unsurprisingly) the Chromium team given their features far fewer limitations. Although still dependant on Plugins, the Utilities component is also dependent on Browser, Network, and UI. Browser is mainly depended on for posting tasks as well as accessing page information, and Network is depended on for accessing the internet. Utilities’ dependency on UI is bidirectional because of the integrated nature of these features. For example, the Bookmarks feature places a “bookmark this page” star in the omni-box of the window. This interaction requires direct access to the UI for creating and listening to this button, and UI depends on Utilities for information on whether to fill in the star (bookmarked page), or leave it blank (not bookmarked).

Other Dependencies

The following are explanations for dependencies not present in our first report and are not explained in this report.

UI dependent on Plugin

Views/Coca depends on plugins for its shared url loader factory

UI dependent on Local Storage

Views/Coca depends on local storage for file system context. An example of this use is when a local path is entered into the address bar. The UI must be able to tell whether this is a URL or a file path

UI dependent on Network

Views/coca depends on network for URL biased functions, such as to tell it the security of the site, so it can update the url text, etc
Extra depends on network to convert a file path into a url

UI dependent on Browser

Extra depends on browser for miscellaneous backend, such as logging, compiling, and threading. Aura depends on browser for backend, such as logging, compiling, and threading. Views/coca depends on browser for miscellaneous backend, such as logging, compiling, and threading. Graphics depends on browser for backend, such as logging, compiling, and threading

Browser dependent on UI

Browser depends on aura and to tell it which platform it is on, Views/coca for I/O, Graphics to draw to screen, base for random utilities such as pointers and material design controllers.

UI depends on Utilities

Views/Coca depends on Utilities so it can update Utility based information, such as whether or not a site is bookmarked, whether the download bar is updated etc. Aura depends on components for memory allocation(client_discardable_shared_memory_manager.h), gpu presentation (displaying the window). Graphics depends on components for viz, the client library and service implementations for compositing and gpu presentation.

Network depends on Browser

Network needs to post tasks.

Browser dependent on Local Storage

Retrieves personal info like bookmarks or themes from local storage.

Local Storage dependent on Browser

Local storage needs access to the browser backend and needs to post tasks

Browser depends on Plugins

Browser depends on plugins to perform split network calls. Relies on Plugins to manage service instances.

Browser dependent on Concurrency

Browser depends on Language support and Core: Language support for its compositor's implementation of LayerTreeFrameSink and Core for to run unit tests and perf tests.

Network dependant on Utilities

I think only one dependency, so check to confirm, but probably for firewall or information the networks wants out of Utilities

Local Storage dependent on Concurrency

Blob storage system depends on Concurrency because it needs to be accessible across multiple processes. Local Storage also depends on Concurrency to run unit tests.

Plugins dependent on Network

The network subsystem within plugins depends on components to make network requests. The purpose of the network subsystem is to split network calls into separate mojo requests. The splitting processes would reduce thread hops, which increases performance

Plugin dependent on UI

Plugins depends on extra to embed things into the UI. Plugin's dada decoder module relies on graphics' sika to provide data decoder's image decoder with Jpegs. Relies on Graphics to tell it where it can imbed plugins(more specifically to implement navigable_contents_delegate and navigable_contents_implementation). Plugins depend on base for random utilities such as pointers and material design controllers

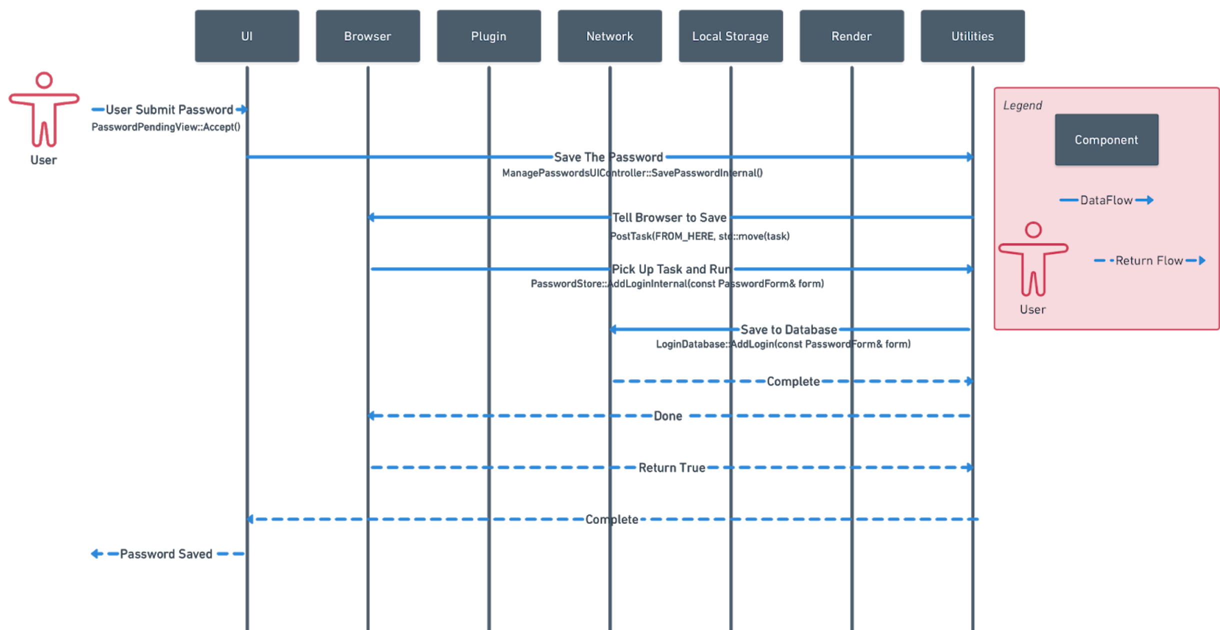
Plugins dependent on Utilities

The network subsystem within plugins depends on utilities to make network requests. The purpose of the network subsystem is to split network calls into separate mojo requests. The splitting processes would reduce thread hops, which increases performance. It also uses utilities to retrieve security and content settings.

Plugins dependant on Concurrency

Plugins depends on language support to implement the network subsystem within plugins

Sequence Diagram 1

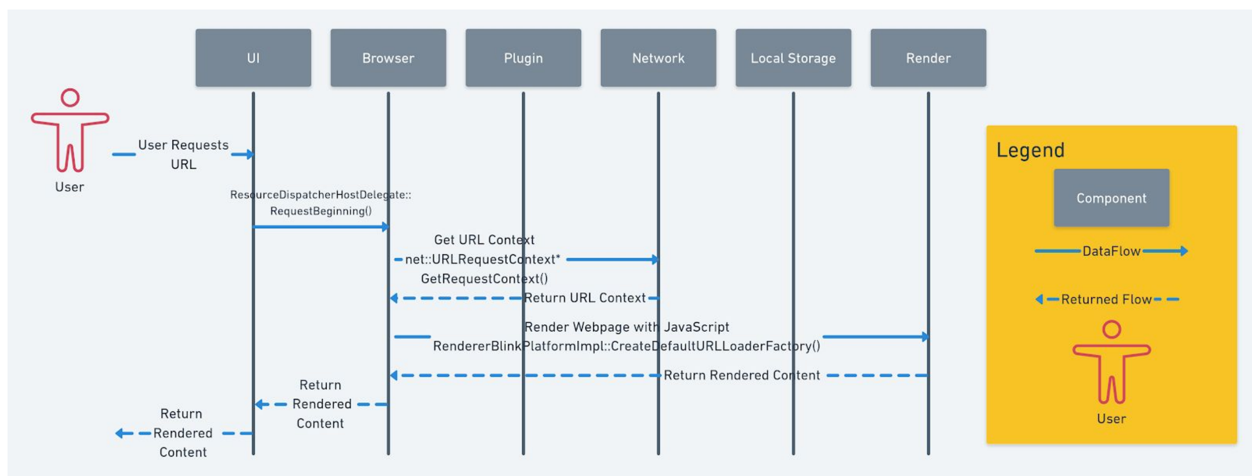


The first use case we investigated was a user submitting a password and Chrome saving it. It is important to note that Chrome has already done work to set-up the following sequence before the user even enters a password. The text box that the user will be entering their password into will already have been identified as a password field by the Password Manager Form Fuzzer. When a tab is loaded, the Autofill Form Parser runs, and the Form Fuzzer predicts which form fields are for passwords. Once a password form is identified, `CreatePasswordFormFromWebForm()` is run to create a new PasswordsForm Element. This process occurs on page load, so we can assume that the form element in our use case has already been properly identified before the sequence begins.

The sequence begins with the user pressing login on the PasswordForm Element, generating a `password_pending_view`. This creates a confirmation pop-up in UI, and sends `PasswordPendingView::Accept()` if the user selects 'yes'. Inside of the UI subsystem, `ManagePasswordsBubbleModel::OnSaveClicked()` is called, beginning the process of saving the password. Eventually, the UI calls `ManagePasswordsUIController::SavePasswordInternal()` from the Password Manager component within Utilities. This function which gets passed to the `new_password_form_manager`, running `FormSaverImpl::SaveImpl()`. Since it is a new password in this use case, it uses the `PasswordStore::AddLogin()`. This function schedules a Task to be posted to the Password_Store Thread, using `PasswordStore::ScheduleTask()` and `TaskRunner::PostTask()` to post a Task in the Browser subsystem.

Technically, Chrome completes Tasks asynchronously, allowing other processes to continue after posting a Task. However, to represent this on a sequence diagram, we will assume the Task is completed right away. In this case, the Task is picked up with `&PasswordStore::AddLoginInternal()`, notifying the PasswordStore that a login needs to be stored. Finally, `PasswordStoreDefault::AddLoginImpl()` tells the LoginDatabase to add the password using `LoginDatabase::AddLogin()` in Network. This method adds the password to the SQL database, saving it across all the user's Chrome applications.

Sequence Diagram 2

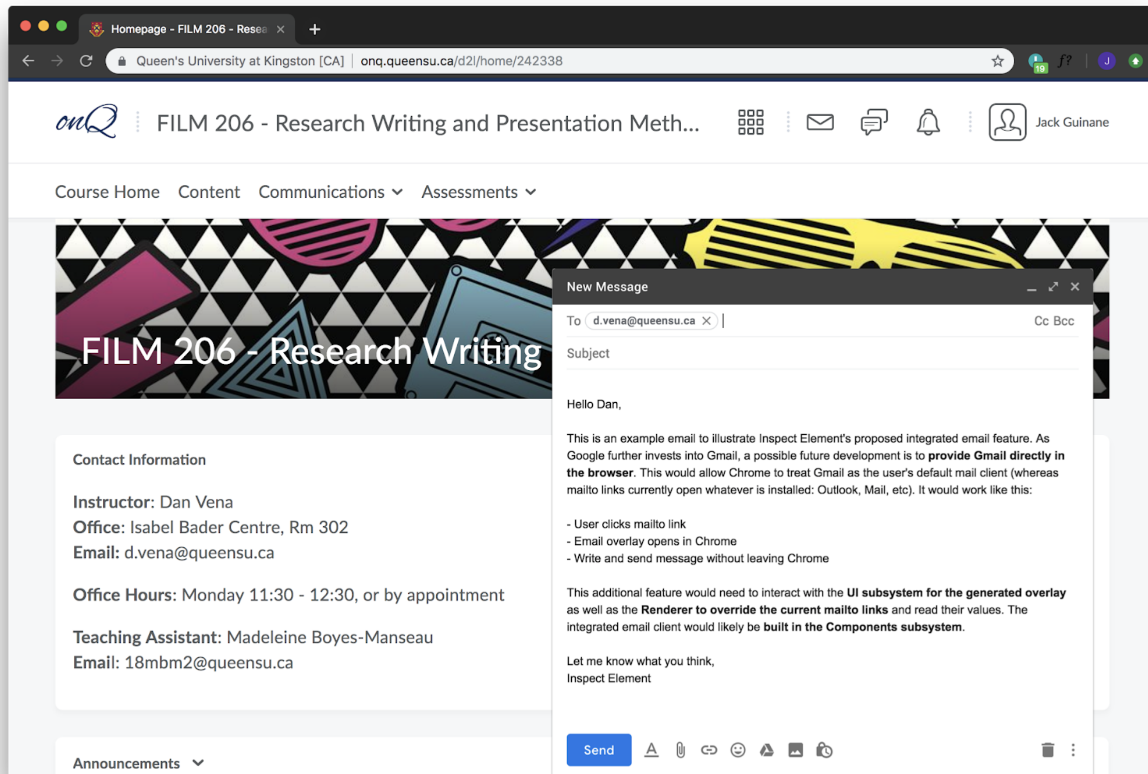


This sequence diagram shows a use case for when a webpage containing JavaScript is rendered. The main components that would render webpages are Blink and the V8 Engine, where most of the rendering would be completed with Blink and V8 would load the JavaScript. In the directories given to us, both Blink and V8 were removed for simplicity on our part of this report. The process of rendering a webpage would involve applying css layouts to the dom tree, exporting this back to the Browser component, and displayed to the user through the UI component.

Here in this sequence diagram, the user requests a URL through the UI. The UI using the ResourceDispatcherHostDelegate requests the beginning of the web content to be loaded and rendered. This request goes to the Browser which then in turn calls the Network get the URL Context with URLRequestContext. The context is returned back to the Browser, which is then passed over to the

Render component to be rendered for the user. The webpage (with JavaScript) is the rendered with the call RendererBlinkPlatformImpl::CreateDefaultURLLoaderFactory(). This content is successfully rendered and returned back to the user after being passed to the Browser and UI components.

Assignment 3 Idea



As Google further invests into Gmail (its free email service), a possible future development is to provide Gmail directly in the browser. Our proposed addition to Google Chrome is a built in email editor, allowing users to compose and send email without leaving their current page. This would allow Chrome to treat Gmail as the user's default mail client, instead of Outlook or Apple Mail.

The way it will function is when a user clicks a mailto link, a pop-up is generated that resembles the Gmail compose window. They can then write and send their email, without leaving the application or even the web page. This additional feature would be developed in the Utilities subsystem, and depend on the UI subsystem (to generate the pop-up) and Render subsystem (to identify and replace mailto links).

Lessons Learned

Lessons Learned: What did we learn since the conceptual architecture?

We learned that the concrete architecture is very different from the conceptual, so much so that the layered parts of the conceptual architecture were dropped. We found that with the thick coupling of the concrete architecture, we ended up with an object oriented style without any hint of a layered style. The thick coupling of Chrome is partly a result of files and classes which are not fully explained or connect parts we did not expect.

To maintain our cohesion and lower some of the coupling we added new subsystems and new dependencies to our conceptual architecture. We added the subsystems Utilities and Concurrency. Utilities are the extended features of Chrome that do not need to follow the same standards as Plugins partly because they were made by the Chrome team. We previously thought that the features in Utilities were built using the Plugin component but a closer look at the architecture shows otherwise. Concurrency is the second addition to our new subsystems and that handles inter-component communications and concurrency across components. Local storage was changed to handle persistence and session to session data as opposed to handling the file storage parts of Chrome.

Where the layered object oriented style of our conceptual architecture had great modifiability, the concrete suggests a decrease in modifiability. One of the drawbacks of the object oriented style is that you need to change all invocations if you plan to modify the structure. The size and interconnectedness of Chrome means that the work of re-connecting these invocations to a new component affects its modifiability.

Chrome Team Issues:

High coupling in the architecture probably led to difficulties with the Chrome team when they needed to do large data migrations or make significant changes to subsystems. These significant changes to structure would mean the Chrome team has to change invocations or design around the interface of the components affected. Another concern is how testable Chrome is due to the side-effects of objects. Objects experience side-effects when two or more objects are affecting the same object. The effects on an object by one of these two (or more) objects will look unexpected to other objects that affect that object too. Side effects mean that the Chrome team has to test for them whenever multiple objects interact, and in a codebase as big as Chrome that testing can be near impossible.

Chrome being open source means that it is likely there are issues with the quality of the code written. The Chrome team must test all new additions thoroughly to release the product that Chrome is today. Looking through the concrete architecture, we found difficult to understand code and bad documentation which shows that our concerns about open-source projects apply to Chrome and create frustrations for the Chrome developers.

The Chrome team is broken into smaller groups because of the size of the code base. These smaller teams need to coordinate and communicate with each other which is an extra challenge to create code that wasn't already made or that another sub-system covers.

Limits of Findings:

It was difficult to find much information about the programmers who wrote the code. In some cases, we were able to find the owners of a directory with emails posted which is helpful for finding other code written under that email. When searching some of the emails online it did not lead to any sort of profile on any site which limited the background available for these programmers.

README.md files were often not present and sometimes when they were, they were not detailed enough. One in particular in base/UI contained the following: "the best description for the role of this component I can come up with is this: random utilities for building UI's". When something vague like this came around we were left to extract information ourselves from the variable names and functionality.

Understand was crucial to our ability to build our concrete architecture. Using Understand though was a challenge and presented its own limits to what we could learn. We experienced huge wait times when initializing, moving nodes from architecture to architecture, opening subsystems and saving. We would sometimes have to wait over 10 minutes to open a sub-system one layer down from the top of the architecture. This severely slowed down our analysis and at times made it nearly impossible to learn the more granular parts of Chrome. We will talk more about the good and the bad of Understand in the section: The Process of Using Understand.

The Process of Using Understand:

It was frustrating that due to the size of Chrome, Understand had issues functioning. Whenever we made copies from the given concrete architecture to our concrete architecture, the program would take at times over 20 minutes for a single node. This usually got worse as time went on, closing the program would fix it but we would experience the long initialization wait times. To properly use Understand we needed to use workarounds. Understand did not accept duplicates so after asking TAs, they suggested we workaround that problem by making new subdirectories. Understand never gave any alternative and would just output an error message that did not suggest these workarounds. When we wanted to share the Understand project, the import/export functionality was not there on some of our computers. When exporting and then importing somewhere else, the project would be unchanged which slowed our ability to work concurrently.

The greatest difficulty was having understand crash while saving. The process of saving is a delicate one that can ruin all the work made until your last save but saving took a minimum of 10 minutes. At times while waiting, we would experience crashes which interrupted the delicate process of saving and corrupted our project. We experience 3 of these crashes where all our data was ruined. Fortunately after the first one we made a back-up on Google Drive.

Our Team Issues:

Our team worked very well together on this project. We scheduled regular meetings and assigned accountable work. We also worked a lot more independently then brought our work to a discussion which added confidence and polish to our work. Our biggest flaw was that we did not always have enough concurrency. Moving forward, we plan to have everyone have some sort of task while any work is being done to make sure we can move efficiently through assignment 3.

Chrome-clusion:

Chrome is large. Chrome is so large that Understand, an enterprise-grade piece of software was unable to work consistently and efficiently. Chrome is also difficult for its developers to work in not only due to its size but the lack of good documentation. Chrome's subdirectories are largely undocumented or contain documentation which is difficult to follow and dive into.

All concrete architectures are very different from their conceptual architectures, Chrome is no different. Chrome has thick coupling which means Chrome has modifiability issues. To combat some of this coupling, the Chrome team made two subsystems, Utilities and Concurrency which we considered in our conceptual architecture but instead merged into other components.

Glossary:

Conceptual Architecture: the developers view of the software architecture
Google Chrome: web browser released by Google based off of the Chromium project
Chromium: an open source web browser which Google Chrome was based off of
HTML: Hypertext Markup Language is an encoding method used to format a web-page layout
CSS: Cascading Style Sheets: a means of formatting the style of elements on web pages
JavaScript: an object-oriented computer programming language commonly used to create interactive effects within web browsers
DOM: Document Object Model is the universal specification for laying out and providing access to HTML objects
URL: Uniform Resource Locator is a protocol for specifying addresses on the Internet
UI: User Interface is an interface allowing the user to communicate with the system. Provides the user with means of input and output
Blink: a fork of the WebCore component of WebKit, which is originally a fork of the KHTML and KJS libraries from KDE
Skia: the open source package used by Chrome to render everything besides text
Views: Chrome's widget toolkit for creating custom browser interface IPC: Interprocess communication is a set of programming interfaces that allow a programmer to coordinate activities among different program processes that can run concurrently in an operating system
Mojo: a collection of runtime libraries providing a platform-agnostic abstraction of common IPC primitives, a message IDL format, and a bindings library with code generation for multiple target languages to facilitate convenient message passing across arbitrary inter- and intra-process boundaries
GDI: Graphics Display Interface: a Windows graphics renderer, used by Chrome for text rendering

External Resources

1) Understand
2) cs.chromium.org
3) https://www.chromium.org/developers/design-documents/aura/aura-overview
4) https://www.chromium.org/developers/design-documents/aura-desktop-window-manager

5) https://www.chromium.org/developers/design-documents/focus-and-activation-in-views-and-aura
6) https://www.chromium.org/developers/design-documents/aura/views