CISC 322 ASSIGNMENT 3 REPORT

Inspect Element: *Roberto Ruiz de la Cruz, Matthew Pollock, Maxwell Keleher, Joseph Gravenor, Jack Guinane, Jonathon Gallucci*

Description

Over one and half billion people suffer from vision loss due to refractive errors [1], and over 300 million people suffer from some form of color blindness [2]. With over 60% of the global browser marketshare [3] [4], it is both surprising and concerning that Google Chrome does not service those who are visually impaired by default [5]. Our new feature seeks to uphold the Internet's goal of being open for everyone. Our plan is to implement an accessibility subsystem that provides Google Chrome with the ability to improve the web-browsing experience for those with farsightedness, cataracts, and various color blindness. Other accessibility features may be added in the future, and would be consolidated under this accessibility subsystem. For now, we will focus on the aforementioned disabilities in our proposed implementation.

FARSIGHTEDNESS:	**CATARACTS:**
Increased text size	Increased text contrast
Zoom image on hover	Image filter to increase image contrast
Large cursor	

READABILITY:	**COLOR BLINDNESS:**
Readable font (serif or sans)	Remove indistinguishable colours from images
Dyslexic fonts	Same process applied to site and text colours

Component Architecture

Our proposed new component is called Accessibility and is contained inside our Utilities component. Accessibility uses an Object Oriented architecture style and contains two components: one called Accessibility UI, and one called Accessibility Backend. Accessibility Backend depends on Render because, when our feature is turned on, it will need access to the DOM in order to change what is displayed on the webpage. The other component is Access UI. This component acts similarly to the backend component, however, instead handles adjustments to elements of Chrome's UI. The reason we decided on this architecture style is because we saw the many extra features that are kept in the Utilities component, so to keep with the architecture of Chrome, we saw this as the best option that also doesn't increase coupling.

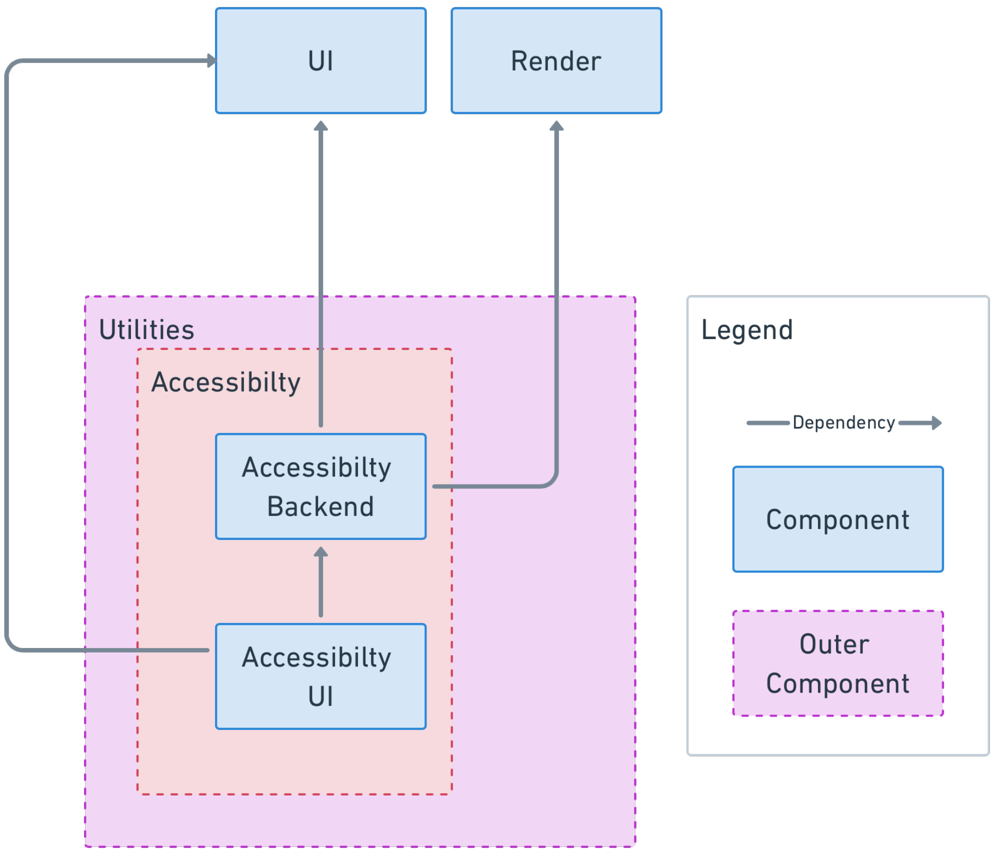
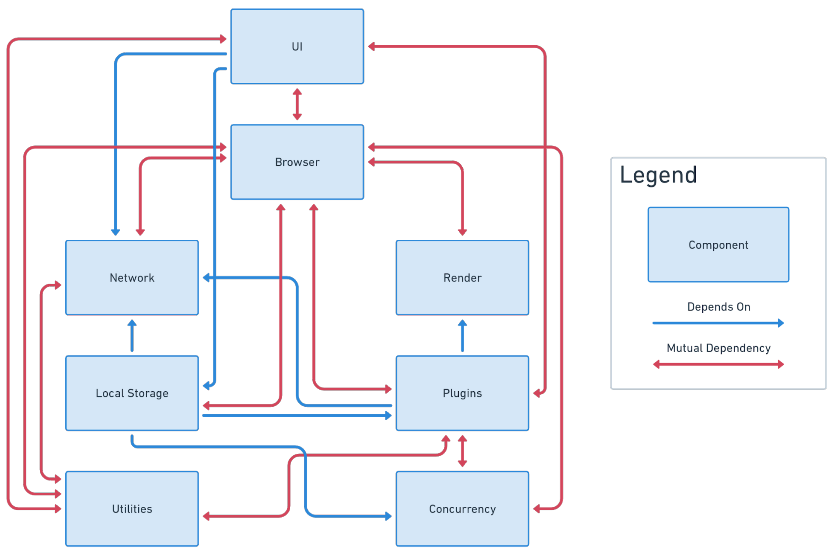


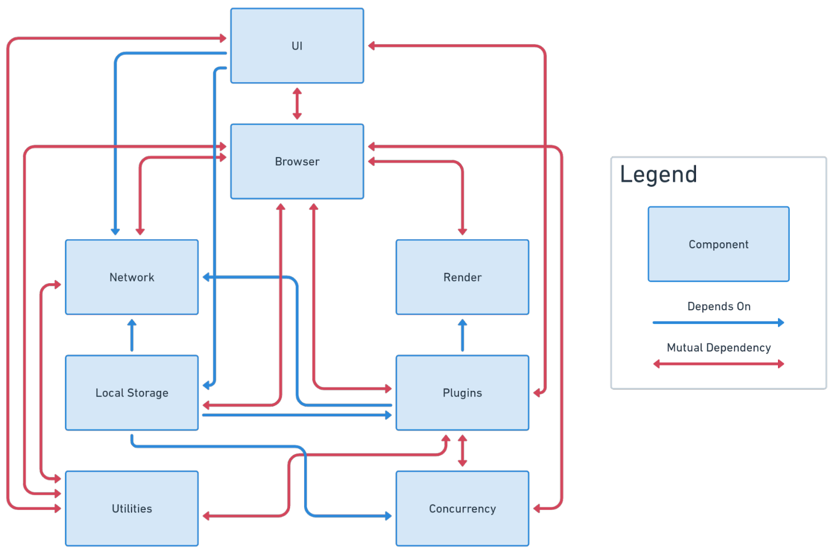
Fig 1: Conceptual architecture of our implementation

State of Chrome Before and After Our Enhancement

The current state of Chrome does not include any accessibility features, beyond changing font size. Our enhancement sought to relieve Chrome of that. The addition of built-in accessibility features would elevate Chrome to be a more inclusive piece of software that a wider part of the population could enjoy out of the box. The Chrome system has 8 highly coupled subsystems. We wanted to be mindful of the amount of dependencies that Chrome already has and we wanted to avoid aggravating that problem. There is one new dependency on the renderer subsystem from utilities. Our project acts as a simple overlay for Chrome that will only interact with the render and UI subsystems so access to render code is required. Our high level conceptual architecture includes this new required dependency. The low level conceptual architecture is discussed in-depth in the Component Architecture section of our report.



The above figure shows our previous concrete diagram from Assignment 2. Below shows the new dependency from Utilities to Render, previously mentioned in this report.



Sequence Diagram and use case diagram

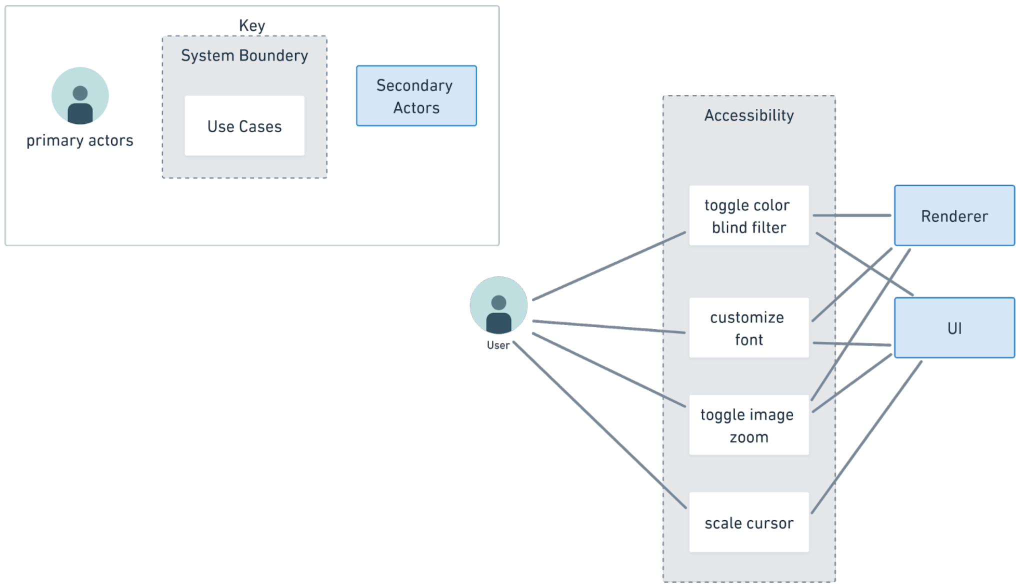


Fig 2: A use case diagram

The first 3 use cases, toggle color blind filter, customize font, and toggle image zoom, all act on both Render and UI. These use cases use the two subsystems to change their colors, replace their fonts, and zoom in on their images, respectively. The last use case, scale cursor, only acts on UI, as the cursor is a UI element.

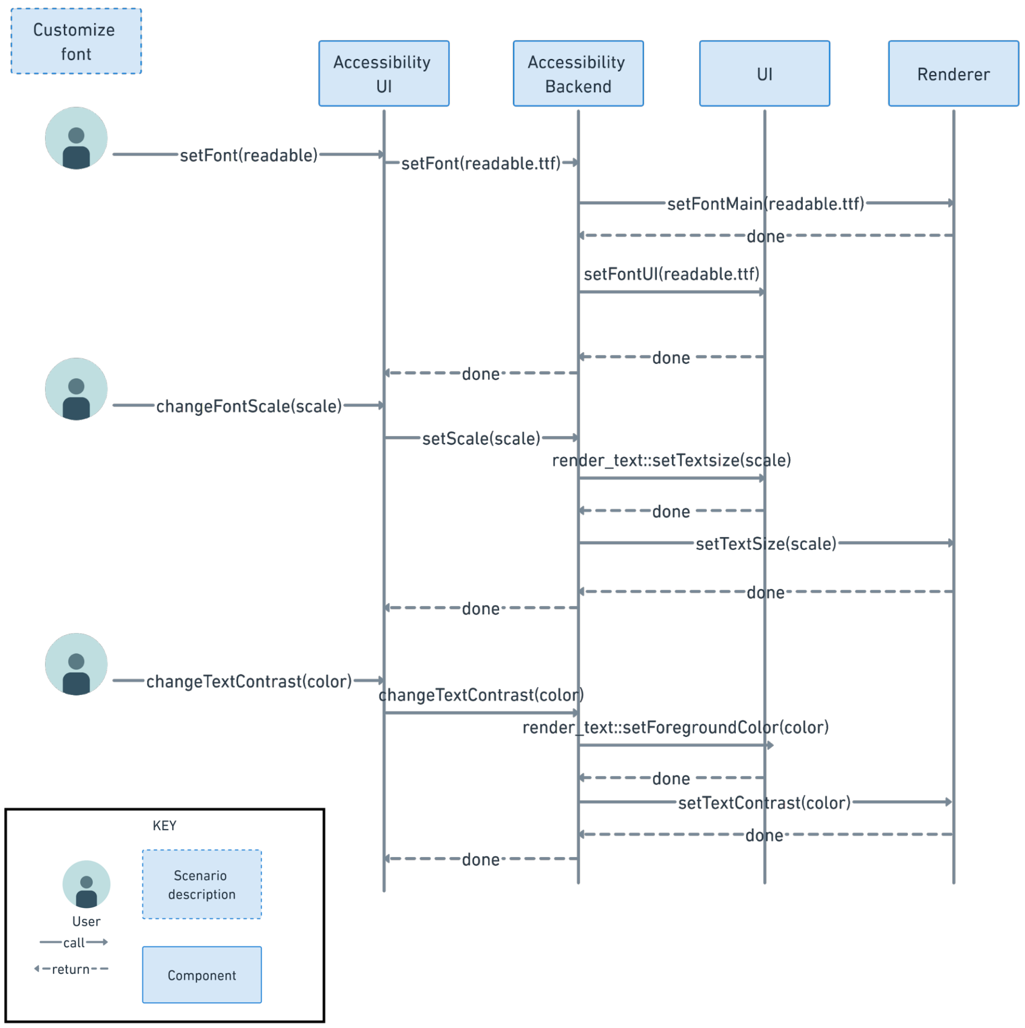


Fig 3: An example sequence diagram

This sequence diagram shows the use case of font customization. All user interaction takes place through Accessibility UI. In the scenario of the user setting the font to the readable font through the Accessibility UI, the Accessibility UI tells the accessibility backend to change the font, passing it the font file. Accessibility backend tells the renderer to change the font of the page text to the passed font. Renderer returns the rendered web page. Next, accessibility backend to change all the font in the UI to the passed font. After this is is finished the UI returns the rendered UI. After this, the UI returns finished. This pattern continues for the next two use cases in this scenario, with Accessibility UI telling accessibility backend how to alter UI and renderer, and the two returning the rendered UI and the rendered web page respectively.

The Effects on Maintainability, Evolvability, Testability, and Performance

Our addition works as an overlay over the U, with small changes to how Chrome should display the information. A dynamic canvas like a video or web application will feel a performance hit due to changing the browsers colours. When Chrome draws the UI components, each frame would need to be colour corrected.

The pressing concern of our application is the testability, maintainability, and evolvability of Chrome. Any component that has its own UI instance will be affected by our enhancement, which would mean developers need to test these instances with the accessibility features on, as well as off. For example, the password manager has its own UI instance which would be affected by the overlay we plan on adding. When testing the password manager, someone in the password manager team will test these windows. With our addition, to thoroughly test these windows, the team needs to see what the windows look like with and without our accessibility features which creates more work when testing.

Alternate Implementations

We considered three possible alternative methods of implementation for our feature. The first is to modify the Utilities subsystem. We would create a new component in Utilities, containing the proposed accessibility module. The benefits of this would be that it would not have large effects on the overall architecture, since Utilities already depends on UI. The main difference is that Utilities would now depend on Render. Based on our Assignment 2 analysis of the Chrome concrete architecture, it would also be a cohesive addition to the Utilities subsystem.

The second method of implementation is to build it directly into the UI subsystem. Since the UI is the subsystem that displays the final window, the accessibility module could modify the window before it is finally presented to the user. This would unfortunately require large changes to the UI subsystem, which may affect dependencies. It would also be harder to test and evolve, since the feature would not be consolidated under one module.

The final method we considered, was creating an entirely new subsystem. This new subsystem would depend on Renderer and UI. The benefits of this method is that it would not require the modification of any other subsystems, and would be easier to evolve and maintain. The drawbacks are that this would create more dependencies throughout the system, causing the overall architecture to be less maintainable, testable, and evolvable.

SAAM Analysis

STAKEHOLDERS

- Alphabet inc
- Chromium Development Team
- Chrome Users
- Chrome Users with Vision Impairment
- Web Developers
- Team Inspect Element

SCENARIOS

- Developer deploys website
- User activates feature
- User accesses website
- Chrome is ported to another platform
- Add new accessibility feature

SCENARIO	UTILITIES IMPLEMENTATION	UI IMPLEMENTATION	IMPLEMENTATION AS NEW SUBSYSTEM
New subsystem added to UI: (Chromium Development Team)	Direct: Our architecture would not be affected by internal changes to the UI subsystem	Indirect would need to change the relations between our Accessibility UI and the other subsystems	Direct: Our architecture would not be affected by internal changes to the UI subsystem
Website is deployed: (Web Developers)	Direct: our feature manipulates the content of a website after it is rendered. The content would still be fetched from the network sub-	Direct: our feature manipulates the content of a website after it is rendered. The content would still be fetched from the network sub-	Direct: our feature manipulates the content of a website after it is rendered. The content would still be fetched from the network sub-

	system by the renderer	system by the renderer	system by the renderer so the actual content being fetched would not affect the architecture
User improves an inaccessible website using our feature: (Chrome Users with Vision Impairment)	Direct: our feature manipulates the blink output in Renderer and/or skia output in UI	Direct: our feature manipulates the blink output in Renderer and/or skia output in UI	Direct: our feature manipulates the blink output in Renderer and/or skia output in UI
User accesses a website without using our feature: (Chrome Users)	Direct: our architecture is only called when activated	Direct: our architecture is only called when activated	Direct: our architecture is only called when activated
Software is ported to another operating system: (Alphabet inc)	Direct: The UI determines how to interact with new platforms. Our architecture would be separated from that decision making	Direct: The aura component handles the platform specifics. Our architecture would be separated from that decision making	Direct: The aura component handles the platform specifics. Our architecture would be separated from that decision making and is outside of the UI subsystem
We add a new accessibility feature: (Team Inspect Element)	Indirect: we would need to implement new functionality. This would require adding a new option to the accessibility UI and adding the functionality to the Accessibility Backend	Indirect: we would need to implement new functionality. This would require adding a new option to the accessibility UI and adding the functionality to the Accessibility Backend	Indirect: we would need to implement new functionality. This would require adding a new option to the accessibility UI and adding the functionality to the Accessibility Backend. New dependencies may appear

Evaluation

We evaluated our implementation based on modifiability and maintainability. We found that all implementations would require a modification of the architecture to implement new functionality for the feature. The UI implementation require modification because it is directly embedded into the subsystem. Any changes made to the UI subsystem would require changes to the interaction of our feature with other internal components. This leaves us to decide between the Utility implementation and New Subsystem implementation. Though a new subsystem would provide more opportunity for evolution, we believe the ease of maintainability from the Utilities implementation makes it the best implementation option. Due to the difficulty of maintainability and poor evolvability of adding it to UI we ranked this as the worst option.

Limitations

The limitations imposed by our enhancement come from making changes to the UI and changing colors. By default, Chrome does not let you change the font of UI elements, so we will have to alter the gfx file within the UI's graphics subsystem to allow for the changing of the UI's fonts and colors. This directory is responsible for making calls to skia, which is responsible for rendering the UI elements. There will also be a slight performance hit from changing the browser's colors. In order to maintain consistency in the color correction we would have to correct css, images, videos, canvas, etc, all at the same time. Since drawing is not necessarily centralized, making sure that no piece of code is drawing before correcting the color being drawn is not trivial.

Software testing

When implementing new features into a product, testing the interactions between the components is crucial to understanding the impact this feature will have on the system as a whole. For testing core feature changes, Google employs a four-stage testing process, consisting of:

1. Testing by dedicated, internal testers (Google employees)
2. Further testing on a crowdtesting platform
3. "Dogfooding," which involves having Google employees use the product in their daily work
4. Beta testing, which involves releasing the product to a small group of Google product end users

These methods of testing are primarily for Google's core business (Google's Search Engine) but can give insight to what the Chrome team would base off their testing methodology. The team that would implement our accessibility features would not have access to Google's expansive register of employees and therefore would not be able to use methods 1 & 3 as much as the Search Engine team as they would not be as large or have as many resources. Teams responsible for Google products that are further away from the company's core business employ a much less strict QA process. In some cases, the only testing is done by the developer responsible for a specific product, with no dedicated testers providing a safety net. [6] While in the past, testing was mostly done manually, nowadays most companies employ a wide variety of automated testing solutions. The tools that are used range from PHPUnit for back-end unit testing to Watir for end-to-end testing efforts. [6]

Ideally, to test software made for people with vision impairments, we would test our own feature in a controlled area and record feedback from these testers with visual impairments when the feature is enabled versus disabled. Getting these suggestions further progresses development for these accessibility features. For our purposes, we will be testing our implementation of our feature as an extension, changing css elements to simulate higher contrast, larger more readable fonts, and larger cursor that a user will be able toggle on and off.

Using our research of vision impairment and the steps we have taken to implement the suggested best practices, we plan to test our feature but toggling on and off each feature independently. Afterwards, we plan on enabling features in combination with each other to test the interactions between them. Finally, we will be enabling all of our extension traits and using Chrome as an average user would to see and record any unwanted interactions.

Testing the impact of interactions between the proposed enhancement and other features can be done with automated scripts that Google already implements. As we were progressing on our Assignment 2 Concrete Architecture, we delved into the files of Chrome with the help of the Chromium documentation website. After searching through many file trees, we found that a lot of the files were used for testing implementations and making sure features do not poorly interact when enabled.

References

1. https://books.google.ca/books?id=jtqSBQAAQBAJ&pg=PA826#v=onepage&q&f=false
2. https://nei.nih.gov/health/color_blindness/facts_about
3. https://netmarketshare.com/browser-market-share.aspx?options=%7B%22filter%22%3A%7B%22%24and%22%3A%5B%7B%22deviceType%22%3A%7B%22%24in%22%3A%5B%22Desktop%2Flaptop%22%5D%7D%7D%5D%7D%2C%22dateLabel%22%3A%22Trend%22%2C%22attributes%22%3A%22share%22%2C%22group%22%3A%22browser%22%2C%22sort%22%3A%7B%22share%22%3A-1%7D%2C%22id%22%3A%22browsersDesktop%22%2C%22dateInterval%22%3A%22Monthly%22%2C%22dateStart%22%3A%222017-11%22%2C%22dateEnd%22%3A%222018-10%22%2C%22segments%22%3A%22-1000%22%7D
4. https://www.statista.com/statistics/544400/market-share-of-internet-browsers-desktop/
5. https://chrome.google.com/webstore/category/collection/accessibility
6. https://techbeacon.com/how-tech-giants-test-software-theres-no-one-way-qa