Name & Matriculation: Ng Chi Hui (U1922243C)
Email: NGCH0118@e.ntu.edu.sg

1. Purpose

In this report, we present an analysis of the Firefox web browser. The goal of this paper is to present findings on Firefox's software architecture, its quality attributes and analysis models. This paper seeks to further understand the details and design choices used by the Mozilla team when creating Firefox. The methodology used to arrive at an accurate and correct architecture was to first read and investigate into 2 primary web browsers of Chrome and Safari to get a fundamental understanding of the general structure for a mature web browser. For further confirmation, Mozilla documentation about the functionality of each subsystem was used. This was in addition to corroboration and correlation of the inner components with the source code.

2. Introduction and Overview

Firefox is an open-sourced, cross-platform web browser developed by Mozilla Foundation, designed for the modern-day Internet users and applications that dominate the web. Firefox is written in C++/C and contains over 2400 kLOC (Grosskurth et al, 2003). It is available for all major operating systems in over 96 languages. Firefox was first released back on Nov 9, 2004, as part of the Mozilla project, with the goal of providing users with the best browsing experience whilst putting the user in control.

The system architecture of Firefox is similar to most modern web browsers which follow the layered architecture whereby each layer below provides services to the upper layer. The architecture of Firefox follows the layered architecture with the NSPR layer (Netscape Portable Runtime) at the bottom which interacts with Operating Systems.

3. Software Architecture

The architecture of Firefox can be described as having a hierarchical heterogeneous style as the whole system is designed in a layered fashion whereas within layers the subsystems are developed in another style. The main system of Firefox is developed in a semi-strict layering, with object-oriented functionality. It is semi-strict as some of the layers from a typical web browser architecture like the JavaScript Interpreter, Networking and XML Parser are combined into Gecko Subsystem. Presumably, this is to allow for better performance which may require closer coupling between logically higher-level functions and their lower-level implementations.

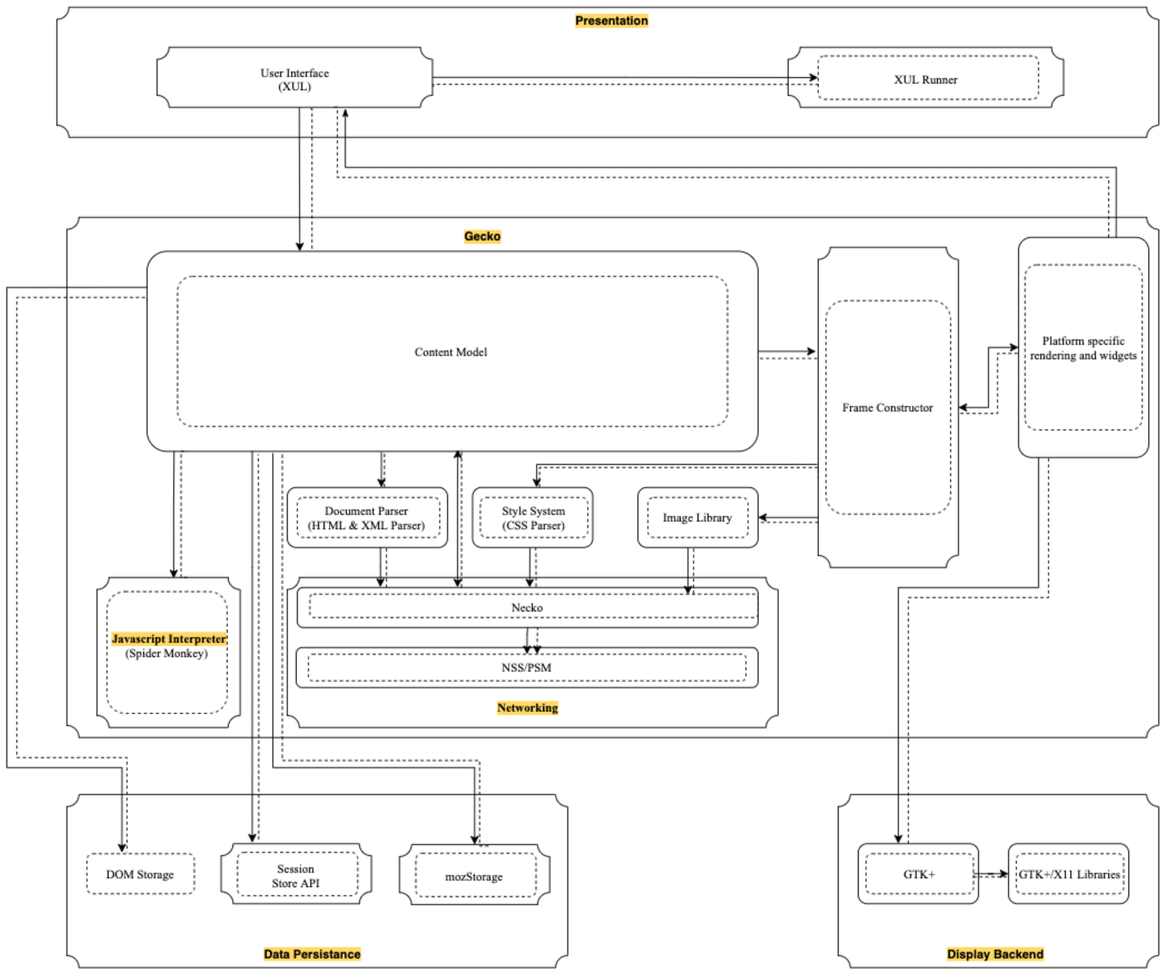

Figure 1 System Architecture of Firefox

The above image illustrates a layered architecture of the Firefox browser and its subsystems.

4. Subsystems of Firefox Architecture

4.1. User Interface (Ui)

The user interface is the first layer in the architecture that renders the browser and allows the user to interact with the browser. Firefox user interface components refer to every part of the browser window other than the requested page. For example, the navigation bar, the address bar, the forward and backward buttons etc.

4.2. Gecko

In modern web browsers, the browser engine and rendering engine is split into 2 different layers. The browser engine works as a bridge between the user interface and the rendering engine. The rendering engine is responsible for drawing a visual representation of the requested web page on the browser screen by interpreting files like HTML, XML and CSS and generating the layout that is displayed in the User Interface. In Mozilla Firefox, the browser and rendering engines are combined and handled by the Gecko subsystem. The Gecko subsystem also combines the networking layer, XML parser and the JavaScript Interpreter unlike the structure of typical web browsers. Gecko layer consists of multiple functionalities which the presentation layer depends on.

Below explains the components within the Gecko subsystems:

A. **Document Parser (HTML & XML Parser)**
   The document parser receives HTML and XML code via Networking subsystem. This happens after Necko receives the URL from the content model when the user passes it from the Ui. After parsing the HTML and/or XML code it sends it back to the content model for further use.
B. **Style System (CSS Parser)**
   The CSS parser receives CSS and style data from Necko before parsing it and sending it to the Frame Constructor for webpage construction.
C. **Image Library**
   Similarly, the Image Library loads image data from Necko before sending it to the Frame Constructor.
D. **Content Model**
   The content model has several functions:
   - Interacts with UI to receive URL data and sends user requests to Necko to retrieve components necessary to render the webpage.
   - Receives aspects of the fetched web page before sending it to the frame constructor to construct the webpage.
     - Receives parsed HTML from HTML parser
     - Interacts with Spider Monkey for interpreted JavaScript
     - Interacts with Data persistence to retrieve cached data
   - Manipulates data received by creating the DOM tree structure before gathering all data needed and sending it to frame constructor
E. **Frame Constructor**
   The frame constructor received parsed HTML, XML and DOM tree data from Content Model as well as image data from the Image Library and CSS data from the Style System. Using these data received, it will construct the webpage by using the platform-specific rendering widgets.
F. **Platform Specific Rendering Widgets**
   Provide platform-specific graphics to Frame Constructor to display the webpage according to the platform Firefox is running on. This is done by interacting with the specific libraries in the display backend before the webpage is returned to the user interface to be displayed.
G. **Networking**
   The networking layer is used to retrieve the URLs entered by the user using common internet protocols such as HTTP or FTP. It is also responsible for handling all aspects of Internet communication, character set translation and security. In Firefox, there are two main components, Necko and NSS/PSM. Necko which is responsible for the transport of data and NSS/PSM is used to support cross-platform development of security enabled client and server applications.

H. JavaScript Interpreter (Spider Monkey)

The JavaScript interpreter of Firefox is known as Spider Monkey. It parses and executes the JavaScript code embedded in a website. Once interpreted, the results will be sent to the rendering engine for display.

4.3. Data Persistence

This is a small database created on the local drive of the computer to store the data of the browser locally. Examples of data stored are user data such as cache, cookies, bookmarks, and preferences. In Firefox, the implementation of the data persistence layer in Firefox is through mozStorage which is a database API built on top of SQLite. It handles the process of getting storage service and then connects to databases.

4.4. Display Backend

The lowest layer of Firefox architecture lies underneath the operating system user interface method. It provides cross-platform interfaces for Ui by drawing basic widgets like combo boxes and windows provided by the various operating systems that are supported by Firefox. The display backend of Firefox is divided into two sub-components of GTK+ adapter and X11 libraries for the UNIX operating system. This layer provides Gecko's Rendering Engine (Frame Constructor) with platform-specific data to allow proper rendering on different platforms. This subsystem allows Firefox to be highly portable by generating platform specific graphic data.

5. Software Architecture Styles and Design Decisions

This section will highlight some of the notable design patterns and architectural styles used in the Firefox software.

5.1. Gecko

Gecko, the heart of the Firefox application, adopts an object-oriented architectural style. One interesting design pattern that can be observed in the Gecko subsystem is the **Adapter pattern**. The intent of the adapter pattern is to convert the interface of a class into another interface that the clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces. The subsystem of Platform-specific Rendering and Widgets adopts this pattern and acts as the "adapter" between Frame Constructor and Display Backend (GTK+ Libraries). The Display Backend is an independent rendering engine that outputs and inputs the same objects regardless of the platform. Thus, the widgets subsystem will perform all rendering specific to the platform for input into the display backend and for generating the output to the user in an appropriate format for the platform. This allows the rendering process in Firefox to be streamlined, supporting cross platform usage for users.

5.2. Necko

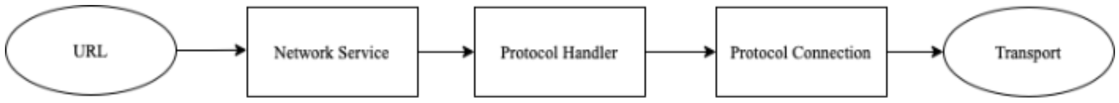

Figure 2 Execution style of Necko Subsystem

In the Necko subsystem, which is the main networking library for Firefox, there is a noticeable difference in architectural style. The **pipe and filter style execution** are employed during a network request and **implicit invocation architecture style** for its observer objects (Lai et al., 2007). The diagram above describes the pipe and filter style architecture of Necko when a network request is initiated. First, a URL will be passed to the network service where the URL protocol is determined, and the appropriate protocol handler is obtained. The protocol handler then instantiates a protocol connection that produces a transport object which is a physical connection to the remote data source. This process runs asynchronously on a separate thread which notifies the application when incoming data is received.

The Necko subsystem also adopts some interesting design patterns such as the **Observer, Proxy, Abstract Factory, Singleton and Façade**. For instance, the **Façade** design pattern is used to abstract concrete implementation for the protocol subsystem within Necko where there are numerous handler objects for the various protocols (e.g., FTP, SMTP, HTTPS etc.). This is implemented through the ProtocolHandler interface. Others such as the stream converter and socket subsystems also use similar **Façade** objects.

6. Quality Attributes

6.1. Maintainability

Maintainability is one of the most important attributes that will affect the quality of the software. The attribute of maintainability refers to the capability of the software product to be modified. One of the ways to analyze the maintainability of the software is through the maintainability index. The maintainability index is calculated as a factored formula of Halstead volume, the cyclomatic complexity per module as well as the Lines of Code per module. Firefox has been shown to be more maintainable than other open-source projects. When compared with Apache, MySQL and Filezilla, Firefox ranks the highest in terms of its maintainability index, of 85 (Ganpati et al., 2012, 228).

In the design of the Firefox browser, many important software principles are used to ensure low coupling and high cohesion. This makes code more maintainable. For instance, each component within the Gecko Subsystem has only one responsibility, it needs to change only when there is a change to that responsibility. I.e., the Document parser is only responsible for parsing HTML and XML files. The Single Responsibility principle is adopted throughout the Gecko subsystem.

The separation of concerns principle is also used in its design at both the class levels as well as the layered architecture. This allows for better modularity as each section addresses a separate concern. For instance, each layer in the Firefox architecture has a well-defined functionality and that there's no to little functional overlap with each other, I,e the presentation layer is only responsible for user interaction through user interface.

6.2. Portability

Portability refers to the capability of the software to be transferred from one environment to another. The Firefox browser fares well in this aspect as it is a cross-platform application available on Windows, OS X, Linux, Android, and iOS and compatible with the up-to-date Web technologies. Individual parts of the design and the design are capable of being reused in different environments. Portability is supported by the Specific Platform Rendering Widgets within the Gecko subsystem together with the Display Backend subsystem as discussed in **Section 5.1** where the Adapter design pattern was used.

6.3. Efficiency

Efficiency refers to the capability of the software product to provide appropriate performance, relative to a number of resources used, under stated conditions. Relative to other popular browsers such as Safari and Chrome, the Firefox browser can utilize memory more efficiently. This is achieved by running the first 4 tabs the user created with the 4 separate processes. Additional tabs opened by the user will be run using threads within those processes. This means that multiple tabs need not create their own processes; they will be able to share a browser engine that already exists in memory. Whereas, in the Chrome browser each process is heavily multithreaded where each tab in the browser runs its own process independent of the others (Shan Gimhana, 2020). Firefox uses 1.77x less memory as compared to Chrome (Mozilla, n.a). In general, Firefox is superior in terms of efficiency in memory management as compared to many other popular web browsers.

6.4. Reusability

Reusability is the property of a software system that its components can be easily reused in the development of other software systems. Firefox/Mozilla applications are designed with good reusability in mind. Many of the core components used by the Firefox browser are also shared with other Mozilla products like Thunderbird and Portable Firefox. Examples of these components includes the handling of web content such as the Gecko subsystem, Necko, and Spider Monkey etc. These components are also termed core components by the Mozilla team and are used in other software systems built by Mozilla (Mozilla, 2013).

Reusability is also enhanced with the use of layered architecture which is designed to limit inter-component dependency and has several advantages. First, they support designs based on increasing levels of abstraction. Second, they support enhancement, where changes to a layer only affect those layers above it. Third, they support reuse of an already developed computing infrastructure layer such as operating systems and networking packages. For instance, the Necko networking subsystem can be used by all upper layer components for networking functions.

7. Analysis Models

7.1. Context Diagram

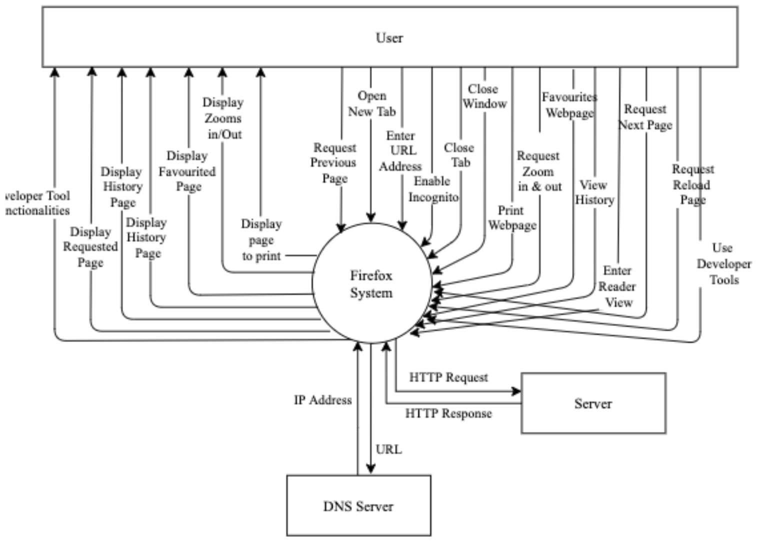

Figure 3 Context Diagram

The context diagram (Figure 3) shows the data flow between external entities and Firefox. There are three main external entities that interact with the Firefox System, the user, and the DNS server.

When the user interacts with the browser, he/she will be able to perform several actions that will request data from the Firefox system, for example bookmarking a page. The Firefox system will have to show to the user that the page has been added to the bookmarks. Firefox supports users with all the common features that a typical web browsers support. Users will interact Firefox Ui to make a request and Firefox handles the request by performing actions to what has been requested. This response by Firefox will then be displayed to the user.

DNS servers translate domain names (i.e., URL entered by user) into IP addresses, making it possible for DNS clients to reach the origin server. When the user enters a particulate website URL into the Ui, Firefox performs a DNS query by sending the website's URL to the DNS server. The DNS server replies with the IP address of the respective URL and returns this to the Firefox system.

The server is used to communicate with the browser to handle user request. When the IP address is obtained, Firefox will send a HTTP/HTTPS request (depending to the Internet Protocol) to the server. After the server handles the request, a HTTP response will then be sent back to Firefox

7.2. Sequence Diagram

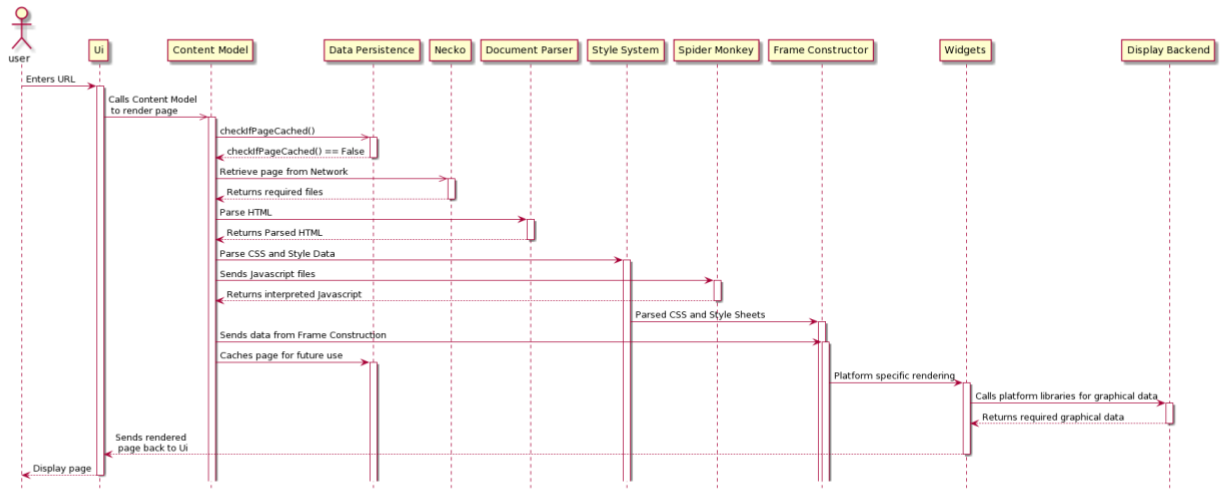
user
Ui
Content Model
Data Persistence
Necko
Document Parser
Style System
Spider Monkey
Frame Constructor
Widgets
Display Backend
Enters URL
Calls Content Model to render page
checkIfPageCached()
checkIfPageCached() == False
Retrieve page from Network
Returns required files
Parse HTML
Returns Parsed HTML
Parse CSS and Style Data
Sends Javascript files
Returns interpreted Javascript
Parsed CSS and Style Sheets
Sends data from Frame Construction
Caches page for future use
Platform specific rendering
Calls platform libraries for graphical data
Returns required graphical data
Sends rendered page back to Ui
Display page

Figure 4 Sequence Diagram of Rendering a Non-Cached Webpage

Figure 4 shows the process of rendering a page that has not been stored in the cache memory. The process begins when the user enters the URL through the Firefox User Interface. The Ui will relay this message to the content model within the Gecko subsystem. The Gecko subsystem will check with the Data Persistence component to verify if the page information has been cached or not, which it has not been in this case. Following this, the URL will be forwarded to the Necko subsystem to fetch the requested page (i.e., HTML, CSS, and JavaScript files) from the web for rendering the page, which Necko returns. The content model will make synchronous calls to the Document Parser, Style System, JavaScript Interpreter, and the Frame Constructor to begin the rendering process concurrently. Parsed HTML and XML are returned to the content model for DOM manipulation. The interpreted JavaScript will also be returned to the content model. The DOM trees and CSS data is then sent to the Frame Constructor to complete the building of the frame of the page. The completed frame will then be sent to platform-specific widgets for final rendering. The widgets subsystem will then call the Display Backend subsystem to access different graphic libraries that will render the page properly. Finally, the page is sent back to the widget's component and then back to the User Interface to be displayed for the user.

7.3. Communication Diagram

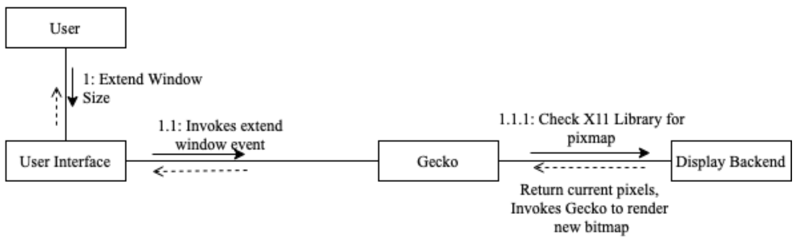
User
1: Extend Window Size
User Interface
1.1: Invokes extend window event
Gecko
1.1.1: Check X11 Library for pixmap
Display Backend
Return current pixels, Invokes Gecko to render new bitmap

Figure 5 Stretching a Browser Window

Resizing begins when the user grabs the corner of the browser window with their mouse and drag to extend it. This resize event is sent to the Gecko which checks the Display Backend for the most recent cached bitmap generated by Gecko. There will be less pixels stored in the display backend than needed to display the extended page view since this is an expansion of the window.

A request will be sent back to Gecko to generate the whole area of the page currently in view. Gecko will render the required graphics and text, respectively, return the bitmap to the browser and display the full page to the user.

8. Conclusion

Firefox was once the top browser in the 2000s and showed promising growth, where it had a market share of 32% of the users in 2009. However, Firefox has shown a steady decline over the past 12 years where it lost a total of half a billion users and 75% of the market share it once held, reducing to a mere 5% of users (Dan, 2021). Whereas, Chrome has risen to the top as it offers a very intuitive synchronization system that allows the user to switch to any device and any google service (e.g., Gmail, Google Docs, Google Meets etc.) in one click. This ecosystem is not present in Firefox and is a major weakness of the software. Even with the various perks of the Firefox browser such as strong privacy and customizability through extensions and add-ons, it still fails to capture the hearts of many users. This would prove to be a stumbling block if not remedied.

In terms of architecture, at its highest layer, Firefox browser architecture best resembles a semi-strict layering where components are allowed to interact with a specific layer underneath it. For instance, the user interface subsystem is only allowed to talk to Gecko. This architecture allows for ease of maintainability and code reuse as the number of components affected by a change in the codebase is limited. While there remains much to be done for the Firefox software, the open-source nature of the browser, strong focus on modularity and code reuse remains the golden standards for designing web browsers.

9. Discussions and Reflections

9.1. Huge Source Code

The source code for Firefox was massive, which makes it impossible to review and understand every piece of detail as that will simply take too much time. Compromise was made by only reading specific parts of the code relevant to the discussions in this paper.

9.2. Poor Documentation

It was difficult to gather useful information about the browser Firefox as there was little documentation on MDN Web Docs. The comments in the source code were non-informative or non-existent. One of the main reasons could be attributed to the fact that Firefox is an open-source software thus, documentation is not of a high priority. Coupled with the rapid release cycle (new version every 4 weeks) of the software, this also means that the system has been changing rapidly, thus causing many inconsistencies across different versions of documentations making the information even more challenging to digest.

9.3. Difficulty in Deducing Higher Level Connections

Even with documentation available on the Mozilla website, most documentations are describing very detailed implementations of the software rather than providing a high-level overview. The various documentations are written for an audience consisting of developers that work closely with altering the low-level design. The specificity of the information was such that it spoke to the individual processes of the subsystems and their unique features that are modifiable. This made it challenging to deduce higher-level connections. Thus, the system architecture of Firefox was heavily referenced to the generic web browsers structure as well as referenced the workings of other similar web browsers like Chrome and Safari.

9.4. Learning Points

Researching and writing for this assignment reinforced the importance of clear documentation which is vital to the understanding of a system. This became especially clear after reading the Mozilla Documentation, which, at times, was not easy to comprehend at all. Many of the documentation might also have not been updated as when searched returned error 404. Meaningful comments are also crucial to the understanding of the functionality between files and subsystems especially when building large and complex software.

10. References

Dan. (2021, September 11). Here's Why Firefox is Seeing a Continuous Decline for Last 12 Years. https://news.itsfoss.com/firefox-continuous-decline/

Dean, B. (2021, July 21). Web Browser Market Share In 2021: 85+ Browser Usage Statistics. https://backlinko.com/browser-market-share

Ganpati et al., A. (2012, October). A Comparative Study of Maintainability Index of Open-Source Software. *International Journal of Emerging Technology and Advanced Engineering*, *2*(10), 228-230. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.414.1313&rep=rep1&type=pdf

Grosskurth et al, A. (2003, May 7). A reference architecture for web browsers. *JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE*, *v1.1*. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.60.1151&rep=rep1&type=pdf

Jazz. (2021, February 13). *Firefox vs Chrome: Which is the best for you?* https://www.debugbar.com/chrome-vs-firefox/

Lai et al., I. (2007, July 06). *Concrete Architecture of Mozilla Firefox (version 2.0.0.3)*. slideplayer. https://slideplayer.com/slide/8638995/

Mozilla. (2013, April 18). *Core*. https://wiki.mozilla.org/Core

Mozilla. (n.a). https://www.mozilla.org/en-US/firefox/features/memory/

Shan Gimhana, D. (2020, July 17). *How Firefox and Chrome use Process and Threads.* https://gimhana-ds.medium.com/how-firefox-and-chrome-use-process-and-threads-f58f478561f0