Concrete Architecture of Firefox

November 16, 2015

Drew Noël	drewmnoel@gmail.com	212513784
Einas Madi	emadi@my.yorku.ca	211558897
Siraj Rauff	srauff@my.yorku.ca	212592192
Skyler Layne	skylerclayne@gmail.com	212166906

Contents

1. Abstract

This project involves the analysis of the Concrete Architecture of Mozilla Firefox. Comparisons will be made to the Conceptual architecture derived in the previous portion of this course.

The architecture itself was extracted utilizing a number of tools used in various stages. A static code analysis tool, Understand, was used to identify the relations between the various files and directories in the Mozilla Source Code. This part was provided for us due to licensing issues, as well as a base containment file and layout that provided some structure. These were then further modified to expose the overall concrete architecture, which was visualized and modified via a Java tool known as Landscape Editor. The resulting concrete architecture was found to follow closely to the conceptual architecture derived previously, with a few notable exceptions.

Analysis of the source code led to the discovery of the three-tiered layered architecture specified in the Mozilla documentation. However, two major subsystems were found to interact with all three layers, but provided little functionality apart from data storage or third party libraries. These were split into two 'horizontal' layers labeled Libraries and Data Persistence.

For the sake of this project, we delved deeper into the architecture of a specific sub-system, in this case, the third or bottom layer known as Core App Services. The intention behind this layer is to provide an interface for the second layer, DOM, and to the rest of the world including the OS, the internet, as well as provide utilities such as JavaScript parsing and resource address resolution.

The nature of this layer results in an object-oriented conceptual architecture, with pluggable XPCOM objects each providing unique functionality. This further lead to an assumption that the majority of the dependencies that exist within Core App Services would be to and from the second layer with few – if any – dependencies within the layer itself.

Upon inspection and visualization of the code, the bottom layer was found to have an abundance of inter-dependencies. Much of these inter-dependencies defeated the original intent of a system designed to have pluggable modules since they lacked any sort of interface and were calling upon each other directly. This effectively created fully-fledged dependencies between the modules and would take a serious investment of effort to switch out a component.

2. Introduction and Overview

The purpose of this report is to expand upon our study of the Firefox architecture. We will be exploring the derivation of the concrete architecture of Mozilla Firefox and compare it with the conceptual architecture derived in the previous assignment. The concrete architecture for this project was derived starting with the conceptual architecture and utilizing various tools and methods on the source code provided by the Mozilla Foundation to explore the architecture in detail. Documentation provided by the

Mozilla foundation regarding the directory structure and explanation of components proved quite useful throughout this process.

The concrete architecture followed the conceptual architecture in an overall view, as it was mainly comprised of a layered architectural system. With closer inspection however, it was clear that this architectural view would require some refinement, and lead to the modification of the original conceptual architecture. When looking at the concrete architecture of the Core App Services layer, we found that it differed from what we originally believed in our conceptual architecture. We believed that all of the components within the Core App Services layer were pluggable modules written as XPCOM objects with little communications between one another. We also thought that the UI layer did not interact directly with Core App Services; this was disproved upon inspection of the source code.

Each of the components in the Core App Services layer are written using an object-oriented pattern. We will also look deeper at the objects and interactions within this layer. Specifically, we will be taking a further look at the following components:

- Necko
- SpiderMonkey
- NSPRpub
- Internationalization
- Inter-Process Communication (IPC)
- Hardware Abstraction Layer (HAL)
- Display Interfaces
- URILoader
- Security

3. Architecture

In the previous assignment, we looked at the conceptual architecture of Firefox. Utilizing the Mozilla Foundation's Documentation as well as some basic code inspection we derived a three-tiered layered architecture, with each layer containing its own unique architecture. The focus of our previous report was an explanation of the derived conceptual architecture.

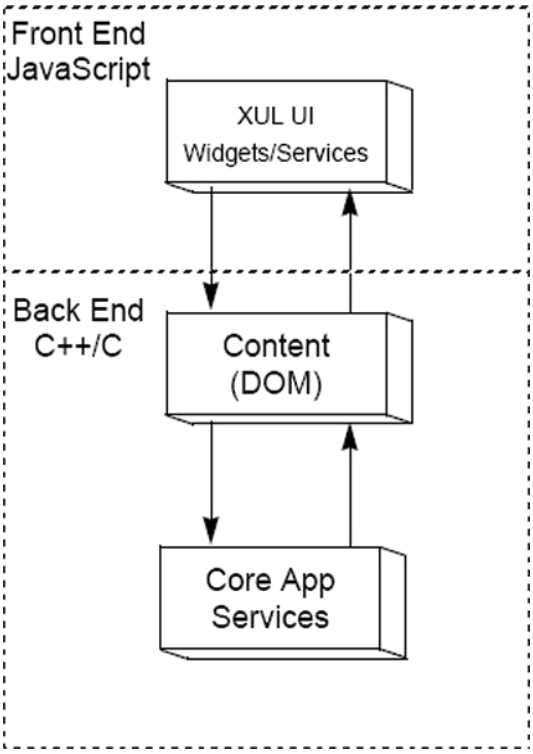

Figure 1: Original Conceptual Architecture

Upon further inspection, this architecture proved to be partly inaccurate. Overall, the concrete architecture tended towards the three-tiered layered architecture we had found previously, however all three layers were found to make use of multiple modules that were not distinctly part of any of the three main layers. These modules were found to have common uses or patterns, and were placed into two groups.

The first group, *Libraries*, consisted of modules that were used throughout the source code. One example of this was XPCOM as the majority of the objects, regardless of layer, interfaced with the XPCOM module. Some other libraries include modules within internationalization that provide unique characters and encodings depending on locale, which are utilized by a wide variety of modules throughout the overall system.

The second group, Data Persistence, contained modules that had little functionality other than storing data on the host system. This includes cases such as caching and

storing user profiles and local content. Separate layers made use of this for various reasons including accessing profile settings and options as well as simply caching content for increased performance.

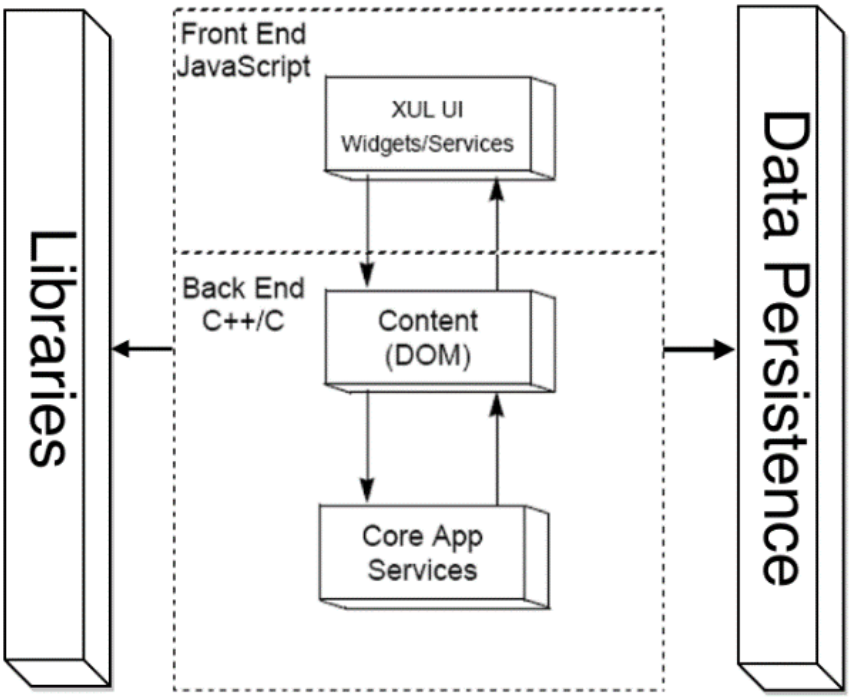

Figure 2: Modified Conceptual Architecture

Conceptual vs Concrete Dependencies

Although much of the concrete architecture was found to be similar to the conceptual architecture in terms of groupings, the dependency relations were found to deviate significantly between the two. Two major methods of deviance were discovered.

First, in the conceptual architecture, non-adjacent layers should not communicate directly, rather, they should make use of the layers in between to pass information back and forth. This, however, did not prove to be the case as one can see in our concrete architecture in Figure 3. There are direct dependencies between the UI layer and the Core App Services layer (the topmost and bottom most layer). In the layered architecture that was created in the conceptual assignment, these two layers would have had to make use of the DOM layer to communicate.

Second, the dependencies within the architectures of the subsystems were not as expected. Both the top and bottom layer were originally found to follow an Object-Oriented model; the intention being that objects should be pluggable. This meant that they should ideally depend solely on the adjacent layers through an interface and limit inter-dependencies, which did not prove to be the case as seen in Figure 6. Unexpected

dependencies were also found within the DOM layer where various ‘filters’ in the pipe-and-filter architecture were found to have forward, as well as backward, dependencies.

Core App Services Concrete vs. Conceptual

For the purpose of this report, we are limiting the in-depth analysis to a single-layer, the Core App Services, Figure 5. As mentioned prior, the conceptual model for the Core App Services involved each module being pluggable and ideally should not depend on each other; rather they should implement clean interfaces that are defined in the DOM layer.

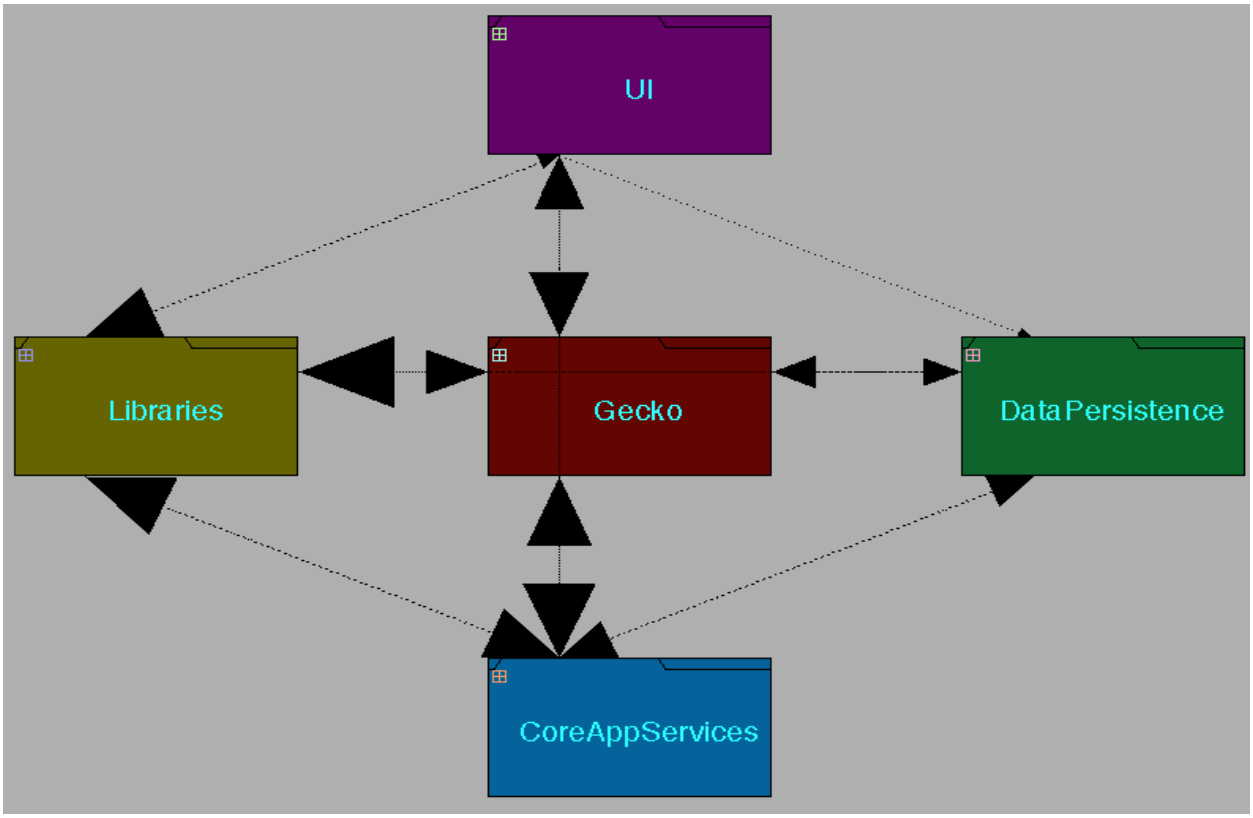

Figure 3: Overall Concrete Architecture

The main advantage of this is that it creates a scenario where the DOM layer is allowed to be the main driving force, leaving any implementation specific information (such as character encodings, locale, etc.) to be easily changed via objects in the Core App Services layer. This is important due to the structure of the Mozilla Application Framework that relies on the DOM for the majority of the heavy lifting and pluggable module to provide it with connections to the operating system and the internet as well as extendable functionality.

Immediately noticeable from the extracted concrete architecture is the abundance of inter-dependencies between the Core App Services components, as seen in Figure 4.

The vast amount of inter-dependencies because the modules to be much less replaceable than was intended. What follows is an in-depth look at the causes for unexpected dependencies on a per-module basis.

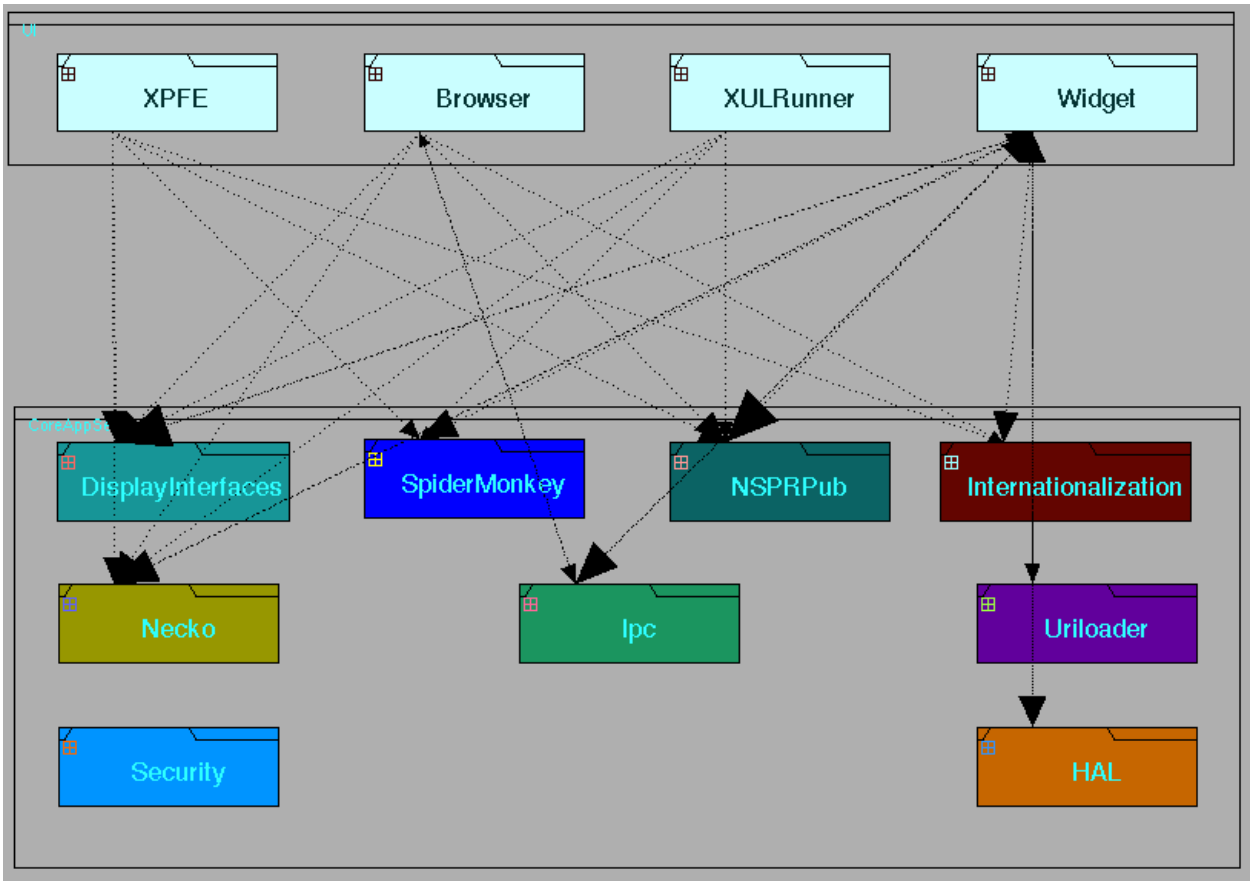

Figure 4: Unexpected Interdependencies between UI and Core App Services

Necko

Some of these inter-module dependencies were due to very simple causes, for example, Necko. Necko provides the application with a networking interface necessary to connect it to the internet. Ideally, Necko would implement an object interface that the DOM layer requires, and each of the other modules would then expect a network stream of the base type. However, in Firefox, as well as in other applications built using the Mozilla Application Framework, there is a single network interface object per

application. As such, the other components have no choice but to interface with Necko for network access.

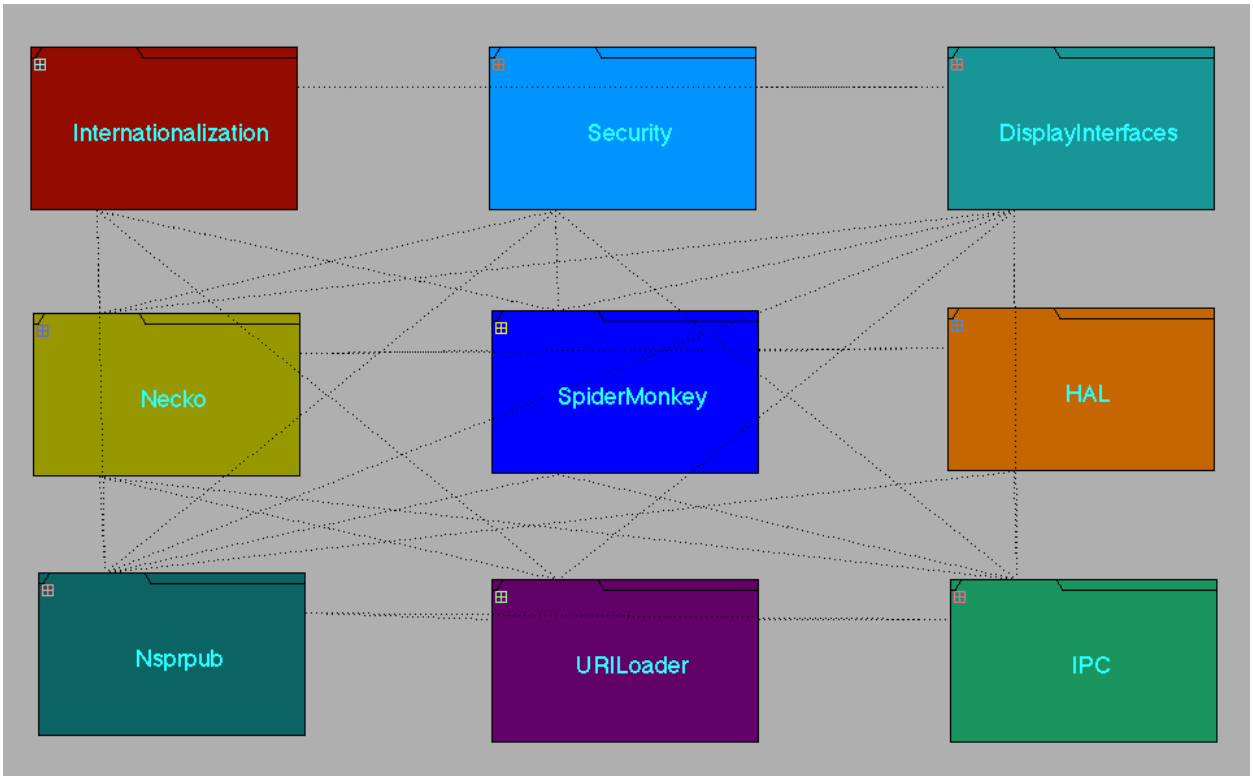

Figure 5: Core App Services Concrete Architecture

SpiderMonkey

The next of the simple dependencies is found in the SpiderMonkey module. As discussed in the previous assignment, Mozilla applications contain a piece of software to allow JavaScript code to interact with XPCOM objects, as well as allow XPCOM code to interact with JavaScript objects. This piece of software is located inside the SpiderMonkey module since it is actually written in JavaScript itself.

NSPRpub

The NSPRpub module is an abstraction module that allows the rest of the Mozilla application to be written in a platform-agnostic manner. This module contains compilation macros that allow Firefox to use various functionality such as threading, 64-bit integers, floating-point number to string conversions, and many others. Without this module, parts of the application would be more difficult to read; any call to these functions would need to be aware of which platform it has been compiled.

Internationalization

The internationalization module provides encoding support for the application, namely UTF-8 and UTF-16 support. If the network traffic sent from the server is in one encoding but Firefox expects it to be in another encoding, this module will decode the text

correctly. The advantage to using this module rather than surrounding all text access with encoding/decoding functions is that this allows for support of additional encodings (for example UTF-32 or CP1047) to be added in one location. Additionally, developers will likely forget to wrap their text access with the needed functions for proper decoding, and could lead to encoding errors in the application. The dependencies related to this module are therefore justified.

Inter-Process Communication

The Inter-Process Communication module or IPC contains the IPC code from the Chromium Project. This code contains bindings on how the children processes of the application can communicate with each other and with the parent process. The Chromium IPC code contains various implementations of communication methods such as UNIX sockets (which allow for TCP/IP style datastreams) and DBus (a binary data bus that is shared with other applications). It is possible that this module could also be placed in the Libraries layer due to the nature of the module.

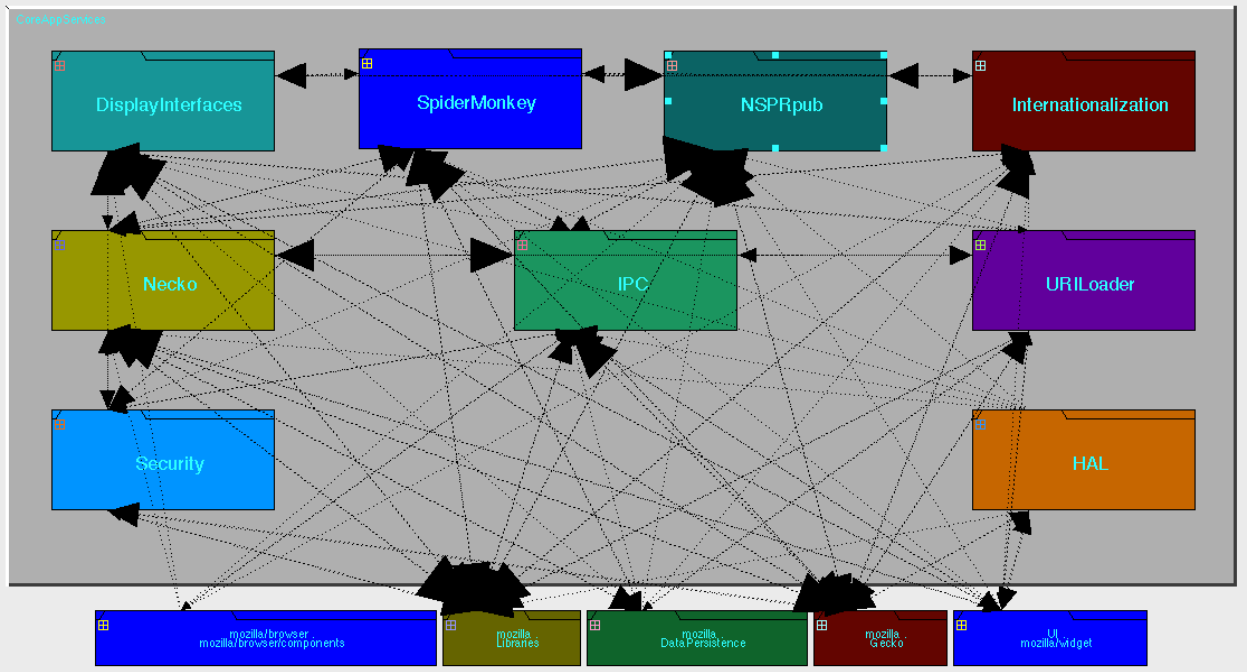

Figure 6: Core App Services Concrete Architecture with Directional Dependencies

Hardware Abstraction Layer

The Hardware Abstraction Layer, HAL, provides a way for the DOM layer to interact with hardware devices that are attached to the guest machine. In addition to battery sensor and vibration motor interaction, HAL also provides interfacing for some memory and disk space monitoring. This last duo of functionality would be better suited to be in Memory and NSPRpub respectively.

Display Interfaces

The Display Interfaces module covers all forms of accessibility apart from internationalization such as fonts, zooming, colors and keyboard shortcuts. The Display Interfaces module closely follows the XUL Accessibility Guidelines for applications to be used by everyone including people with physical, sensory, or communicative disabilities. Deciding the location of this module was a little tricky. At first glance, this module seems to belong in the UI subsystem. However, when looking deeper into the code, it belonged more within the Core App Services subsystem. There are more than two and half calls from within Core App Services subsystem to the Display Interfaces module versus the UI subsystem. As well, the Display Interfaces module interfaces with GTK, a multi-platform toolkit for creating graphical user interfaces, which makes this module operator system dependent. This makes the module more of a service versus a UI component.

URILoader

The URILoader interfaces with XPCOM and acts as a service that helps with content dispatch, handling foreign content and caching. While this module was originally thought to be a DOM module, a deeper look into its services proved to us that it is a Core App Services component.

The URILoader interprets content that comes from Necko and dispatches it to the DOM layer. It handles content that Mozilla cannot handle itself by finding applications that aid in the interpretation of the content. It speeds the process of document rendering by prefetching documents that were cached in the memory for faster loading.

Security

The security model in turn had unexpected dependencies due to its sandbox implementation taken from Chrome, as well as the storage of certificates inside the module. Due to the nature of these security elements, it is most likely that it is considered better practice to interface directly with the module that duplicate or forward the security elements to the environment outside the module.

4. Glossary

COM	Component Object Model
CSS	Cascading Style Sheets
DOM	Document Object Model
Gecko	Browser Engine
HTML	Hypertext Markup Language
Necko	Network Library
SpiderMonkey	JavaScript Engine
SSL	Secure Socket Layer
TLS	Transport Layer Security

UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XBL	XML Binding Language
XML	XML Markup Language
XPCOM	Cross-Platform COM
XUL	XML User Language

5. Conclusions

In our original conceptual architecture, we determined that Mozilla Firefox followed a layered architecture with each layer containing its own architecture. The bottom layer was the Core App Services layer that made use of Object-Oriented style for the numerous services it uses including networking and native graphics. Being Object-Oriented, it utilized pluggable XPCOM objects to allow for easy maintenance and evolution.

In studying the source code of Mozilla Firefox, we found many differences between the conceptual architecture and concrete architecture, though most of the original ideas were present. Firefox is still a layered architecture; however, there are multiple instances of communication between all the layers without any restriction to adjacent layers.

Analyzing the source code allowed us to realize that the majority of the communication between layers not adjacent to one another were between classes that provided third party libraries that we labeled as Libraries, as well as data storage that we labeled as Data Persistence. After the separation of these two makeshift layers, the dependencies between the UI layer and the Core App Services layer was greatly decreased.

In our research, we concentrated in particular on the Core App Services layer and found that it closely follows the same architecture envisioned in the conceptual architecture. However, we also discovered that though it was meant to be pluggable, there was a lot of interaction and dependencies between sub-modules making it difficult to replace and maintain modules in the intended manner. We were able to divide the Core App Services layer into 10 modules: Necko, SpiderMonkey, NSPRpub, Internationalization, Inter-Process Communication (IPC), Hardware Abstraction Layer (HAL), Display Interfaces, URILoader, and Security. Each of these modules were unique to one another and communicated mainly with the DOM, Data Persistence and Libraries layer.

6. Lessons Learned

The overall architecture being a layered style, it was not expected that there would be any dependencies between non-adjacent layers, i.e. the UI and Core App Services layers. Additionally, due to the Core App Services layer utilizing an Object-Oriented Architecture it was not expected that there would be inter-dependencies. While

investigating these, many of the unexpected connections were found to follow from reasonable constraints; on the other hand, some connections were due to a “best of the worst” situation, while others still were due to shortcuts for developer expediency and performance degradation (if properly implemented).

In one instance, the SpiderMonkey component, containing XPConnect, connections were not ideal, but at least make sense. The XPConnect software is written in JavaScript and for interfacing with JavaScript. Therefore, it being placed in the JavaScript component is acceptable. The alternative of this would be to actually move this piece to a library or make its own Core App Services module. The former is inappropriate since only Core App Services interface with it, and the latter just shifts the dependencies to another module.

In another instance, Necko, the dependencies are reasonable due to how the Mozilla Framework is implemented. Ideally, modules that require network access would request access from the DOM layer and the DOM layer would hand off the implementation that has been created, in this case, Necko. However, there is only one possible network implementation, and so the modules are aware of this and just interface with Necko directly. This saves code, due to the need to create an interface that only one component could even possibly fulfill.

7. References

"Mozilla Source Code Directory Structure." *Mozilla Developer Network*. N.p., n.d. Web. 15 Nov. 2015.

"Understand™ Static Code Analysis Tool | SciTools.com." *SciTools.com*. N.p., n.d. Web. 15 Nov. 2015.