# Concrete Architecture of Mozilla Firefox 6
# Gecko Subsystem

**Presented by Fully Optimized eXperience**

**Group Members:**
James Brereton – 06069736, 8jb66@queensu.ca
Gordon Krull – 06003108, 8gek@queensu.ca
Rob Staalduinen – 06009513, 8rjs1@queensu.ca
Katie Tanner – 06060472, k.tanner@queensu.ca

**Prepared for CISC 322,**
**Queen's University at Kingston, Ontario, Canada**

**November 14th, 2011**

# Table of Contents

# Abstract

This report is intended to present our findings on the concrete architecture of Mozilla Firefox 6.0. Specifically, it involves an in-depth examination of Gecko, which acts as the Browser and Rendering Engine for Firefox. We will compare the concrete architecture against the conceptual architecture derived in the last deliverable in this assignment. We will be examining any new functionality details we discovered as well as overall changes to the architecture, and justifying any divergences we uncovered from our original conceptual architecture.

We will begin by detailing the derivation process that we used to determine the dependencies in this project, including the processes and techniques used to sort and classify dependencies. We will then provide an in-depth analysis of each of the subsystems within Gecko, explaining their specific functionality and interactions with other components of Gecko. In doing so, we will detail the unexpected dependencies we found, and the reasons that we believe these dependencies exist within the Firefox system.

Beyond the main architectural differences, we will specify the various design patterns implemented within the system, outlining in which systems we found them and the reasons we feel the developers chose to use these design patterns. Additionally, we will provide an updated sequence diagram, detailing the specific flow within the components of the Gecko subsystem. Lastly, we will discuss the lessons we learned in completing our derivation, the limitations of our derivation process, and the techniques we used to overcome these limitations.

# Introduction

### Gecko Subsystem

The Gecko subsystem acts as the rendering and browser engine for the Mozilla Firefox system and is the focus of our concrete architecture. Gecko is responsible for receiving collected data from Necko, and parsing the various HTML, XML, CSS and other elements into a viewable form to be displayed by the user interface. Through our derivation, we found that Gecko was split in to five main subsystems. These subsystems, the Content Model, Frame Constructor, Document Parser, Style System and Platform-Specific Rendering and Widgets, interact in order to render a viewable page displaying the user's desired content.

### Concrete Architecture

The concrete architecture is a refinement of our original conceptual architecture using actual hard-coded file dependencies. We used the lsedit tool in order to edit our architectural landscape and root out any incorrect dependencies caused by misplaced files. We also examined unexpected dependencies using lsedit and determined whether they were caused by misplaced files, or were an inherent dependency in Gecko. We then used

our completed landscape in order to update our conceptual architecture into our full concrete architecture.

## Derivation Process

When deriving our concrete architecture for Firefox 6.0, we began with the conceptual architecture created previously and used this as a reference throughout the rest of our derivation process.

The next step was to begin using lsedit to examine and sort the source code files in order to derive dependencies. We created entities that corresponded to each of our expected subsystems from the conceptual architecture, and began to place files and folders into their proper subsystems within Gecko. Initially, we relied heavily on the naming conventions of folders in order to place them in the correct subsystems.

Once we were satisfied that we had placed all of the folders required into our created Gecko entity (not necessarily the correct subsystems) we then began to examine dependencies and determine which were expected and unexpected. In the case of unexpected dependencies, we often had to drill down to the file level and find individual files that cause the dependencies. In these cases, we used file naming conventions combined with documentation and source code commenting to determine the reason for the dependency and whether it was caused by a misplaced file, or it was an inherent dependency within Gecko.

If it was the case that the file was misplaced, we would move it to the correct subsystem and observe how this affected the dependencies. If we thought that the file was placed correctly but created an unexpected dependency, we took note of where and why the dependency existed. We repeated this process for each unexpected dependency until we were satisfied that we had placed all of our files correctly and with our rationale behind each dependency, finally arriving at our concrete architecture.

# Conceptual Overview

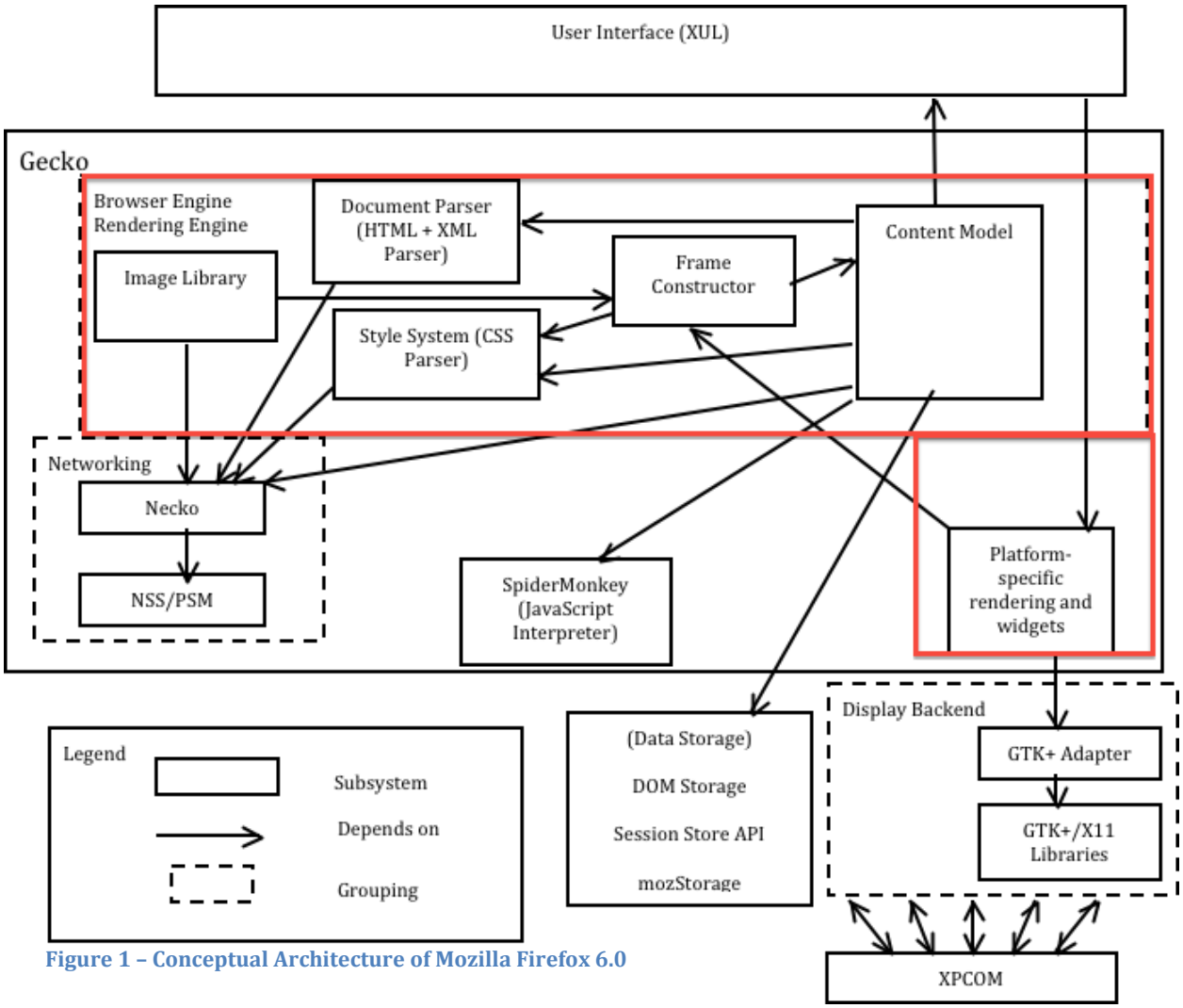


Figure 1 – Conceptual Architecture of Mozilla Firefox 6.0

The figure above depicts our original conceptual architecture of Gecko, with the Rendering and Browser Engine grouped in red. It consists of 6 subsystems, which are the Document Parser, Content Model, Frame Constructor, Image Library, Style System and Platform-Specific Rendering and Widgets. Although our conceptual architecture includes the Networking and JavaScript Interpreter subsystems as well, our concrete architecture focuses only on the components of the Rendering and Browser Engine.

This conceptual architecture is based on the flow of data outlined in Mozilla documentation about page rendering in Firefox. After receiving a page request from the user through the UI, the Content Model calls the networking subsystem, Necko, to get the various pieces of data needed for the page. The Content Model then sends this data to the

Document Parser and Style System for parsing of HTML, XML and CSS for the page. The parsed HTML, XML and image data from the Image Library is placed into the DOM Tree for manipulation by the Content Model, which then sends the DOM tree to the Frame Constructor along with Style Sheets from the Style System. Through reflow, the Frame Constructor builds the frame of the page before sending it to Platform-Specific Rendering and Widgets, which calls upon the Display backend for final rendering before sending the page to the UI Layer. The conceptual architecture depicts an object-oriented style.

## Concrete Overview

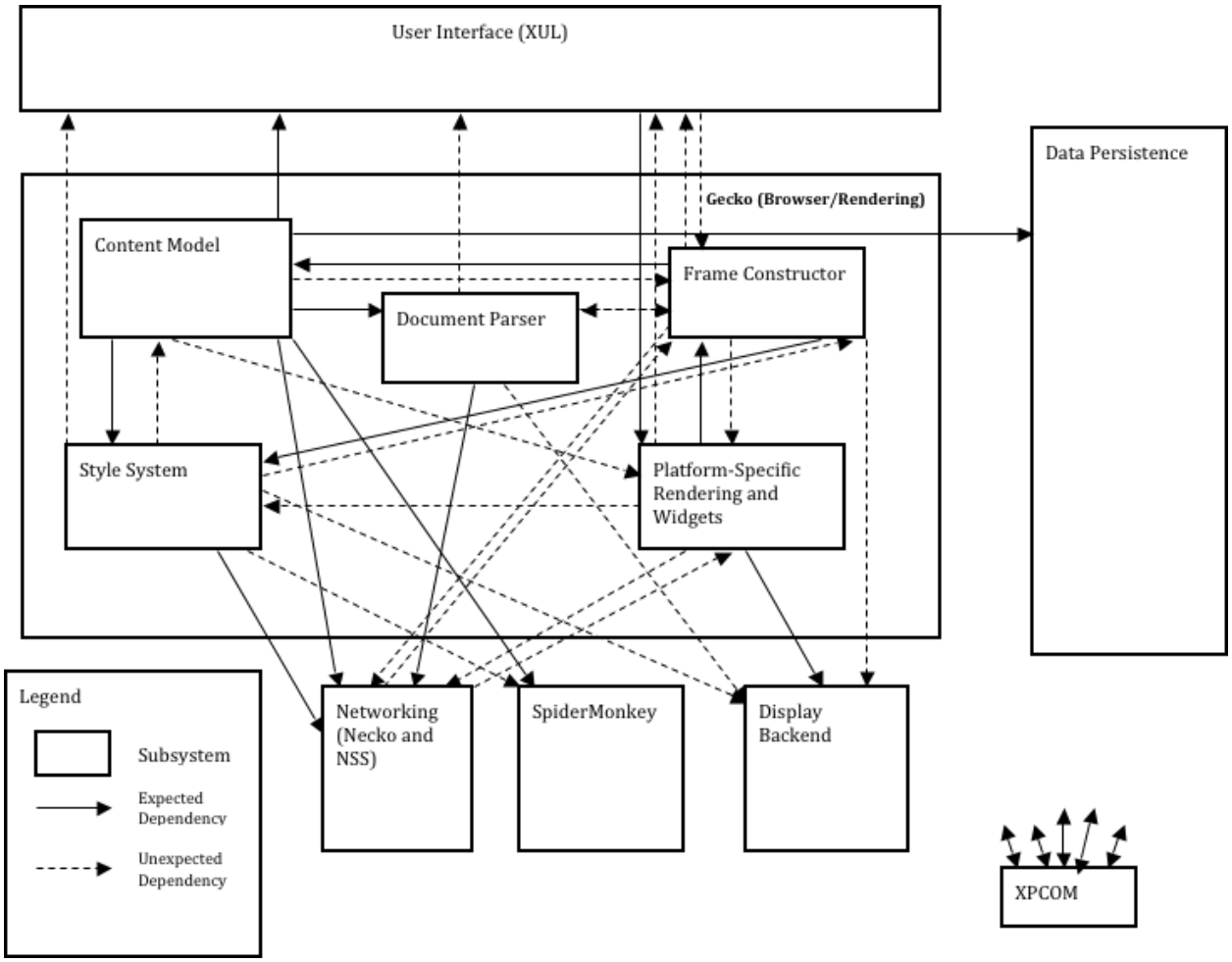


**Figure 2 - Concrete Architecture of Mozilla Firefox 6.0**

Figure 2 shows our representation of the concrete architecture for the Gecko subsystem of Firefox. While it still maintains the object-oriented style that we deduced when deriving the conceptual architecture, it is clear that there are far more dependencies both internal and external to Gecko that we were not expecting between subsystems. Most of these are in the form of new two-way dependencies where we originally expected a one-way dependency between two components of the Gecko subsystem, as well as system calls to external components that require use of files such as content sinks and parsers located

within Gecko. Out of a possible 30 internal dependencies, our derivation process revealed 16 visible dependencies.

## Conceptual vs. Concrete

The most significant difference between our conceptual architecture and our concrete architecture is the number of subsystems. Our conceptual architecture included the Image Library, a subsystem we believed to be responsible for the uploading and rendering of images. During our derivation of the concrete architecture, we found that while there are definitely libraries dedicated to images in the jpeg, png, libimg, and libpr0n folders within the source code of Firefox, there are multiple files outside of these directories within the other components of Gecko that are responsible for the rendering and handling of image data needed for a web page. These can be found in canvas subfolder of the content folder and a number of different files that don't show up in lsedit due to their file extension. As result of this and the discovery of standards for image and video libraries within the Content Model, we removed the Image Library from our concrete architecture and included libraries responsible for image, video and audio standards within the Content Model.

Apart from this major difference and the evident increase in the number of overall dependencies, the architectural style still remains object-oriented, since it is the only architectural style that corresponds to the concrete architecture's significant interdependence between components.

## Document Parser

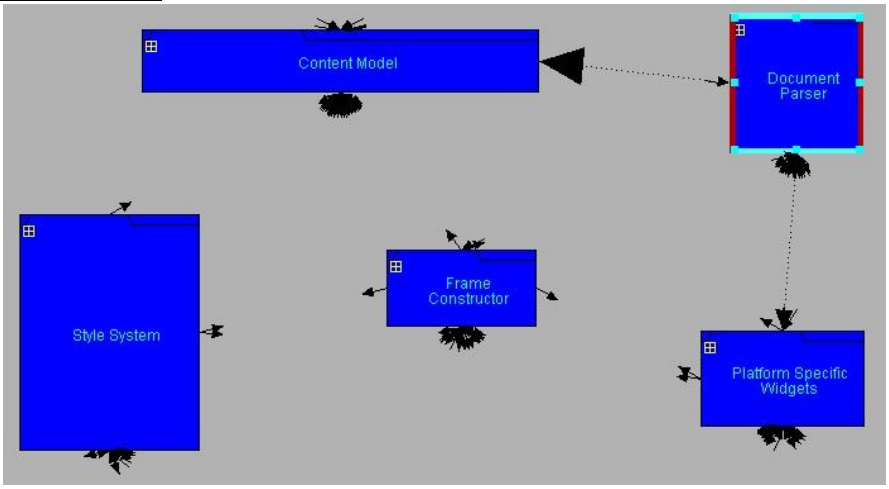


**Figure 3 - Document Parser Dependencies**

The Document Parser is the main parser within Gecko. It parses all HTML and XML data sent to Gecko before sending it back to the Content Model or the other components of Gecko and Firefox. The contents of the Document Parser are found within the parser folder, which in turn contains expat, htmlparser, html, and xml folders. In terms of expected dependencies, the Content Model depends on the Document Parser to parse the HTML and

XML data required for rendering a web page, and externally, the Document Parser depends on Necko for any additional data that needs to be accessed via the network.

**Unexpected Dependencies (Internal)**

The Document Parser depends on the Content Model, which was not expected since upon face value the Content Model dependency should only be one-way rather than two-way. This was justified through the discovery of calls from the nsParser file and nsHtml5TreeOperation file. The Document Parser calls a number of HTML form constant files within content, which we believe, based on source code comments, are called upon so that the HTML parser can conform to particular forms when parsing HTML. The Html5TreeOperation file calls upon a file called nsEventStates.cpp, which is responsible for notifying components when changes occur to the layout. . The Html5TreeOperation file calls upon a file called nsEventStates.cpp, which is responsible for notifying components when changes occur to the layout. The Document Parser probably uses this file to notify the content model if there are any changes in the HTML5 data in the code.

**Unexpected Dependencies (External)**

External to Gecko, the Display Backend and User Interface subsystems have unexpected interactions with the Document Parser. The Display Backend calls upon the Html5Parser and Html5StreamParser files through the TextureWrap.cpp file. This is one example of a dependency whose cause is difficult to determine. We do know that TextureWrap is used for the creation of sample shapes, and since it is calling the files responsible for the parsing of HTML5, these shapes may require some HTML5 code which must be parsed. It should be noted that the TextureWrap file is within a sample code folder within the Display Backend, so it may merely be a dependency for testing purposes.

The User Interface interacts with the Document Parser through nsScriptableUnescapeHTML.cpp, which converts HTML to plain text. It also calls upon the Document Parser through the editor built into the UI, which makes sense since the editor is responsible for the editing of HTML through the user interface in the Firefox browser (through the ContentSink file).

# Content Model

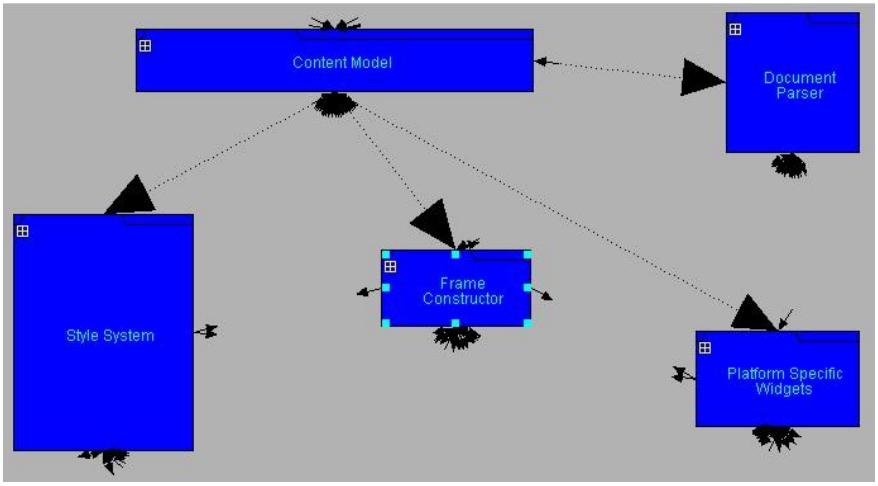


Figure 4 - Content Model Dependencies

The Gecko Content Model acts as the first point of contact for all information being received from Necko. It is responsible for providing the HTML and XML data to the Document Parser, using this parsed data to create the HTML DOM Tree. It also provides the CSS Data to the Style System for parsing. It then passes this data to the Frame Constructor in order to create viewable content. Based on our conceptual architecture, we were expecting many outgoing dependencies from the content model. We were expecting dependencies on the User Interface, the JavaScript interpreter, Necko, the Document Parser and the Style System. In creating our Concrete Architecture, we initially grouped six folders in to the Content Model, which included the folders labelled DocShell, Media, DOM, Content, RDF and Browser, but later determined that some files within these folders needed to be moved to different components.

### Unexpected Dependencies (Internal)

We had expected there to be a dependency from the Frame Constructor to the Content Model in order to access the DOM Tree, but we did not expect this dependency to be bi-directional. In examining the files within Frame Constructor, we found two files called nsTreeUtils and nsTreeContentView, which are used by the Frame Constructor to interact with the DOM Tree. We found that the Content Model uses these files as it is constructing the DOM Tree, rather than having its own implementation for these processes. This dependency was one purely made to avoid duplication of code.

We had not expected there to be a dependency on the Platform Specific Rendering and Widgets component, as we believed all the rendering interaction would take place with the Frame Constructor. However, upon examination, we discovered two solid links that justified this particular dependency. The first is that Content Model will call an object called puppetWidget, which, deceptively, is not a widget at all. It is in fact a placeholder for widgetless rendering, such as sandboxed processes. Often sandboxed processes don't follow many of the same rules as platform specific processes, forcing the need to jump past the frame constructor in order to get access to this particular class. We also discovered a

dependency that is specific to android devices. In order to construct the DOM Tree for android devices, the Content model requires GPS Data and other hardware functionality before passing it on to the Frame Constructor. As such, it will call nsAndroidBridge and nsAppShell in order to gain access to this functionality.

## Style System

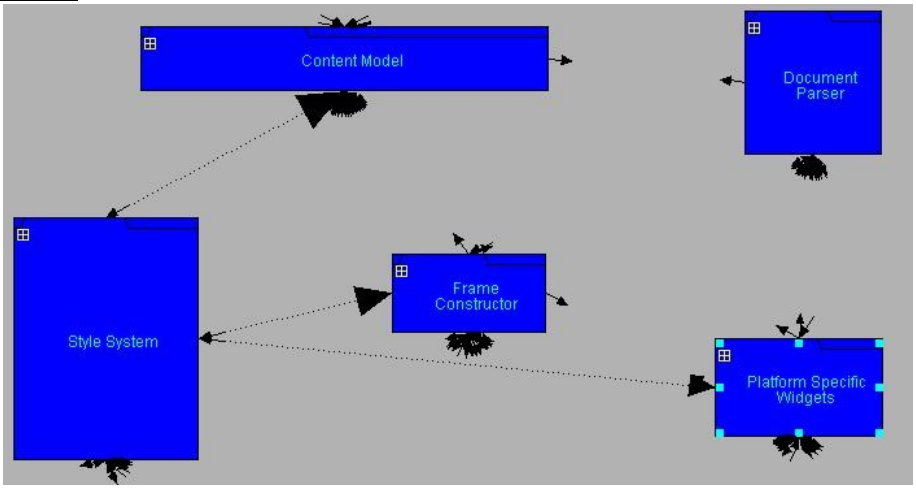


Figure 5 - Style System Dependencies

The purpose of the Style System to handle style sheets as well as CSS Parsing as needed. It receives style data from Necko, then processes it and sends it on to the Frame Constructor. In our concrete architecture we included the style folder, located within the layout folder. We also included various individual files relevant to CSS parsing from within the layout folder in order to eliminate dependencies caused by misplaced files. Initially we had only expected three forward dependencies from the Style System: one from Frame Constructor to the Style System, one from Content Model to the Style System, and one from Style System to Necko.

### Unexpected Dependencies (Internal)

We found that the dependency on the Content Model was caused by a large number of Style Rules and Interfaces within the Content Model which are called upon by nsCSSRuleProcessor.cpp. We believe these rules are called upon to govern the process of receiving and parsing CSS data that the Style System receives from Necko before sending it along.

The Frame Constructor contains a large number of CSS objects, which the Style system calls upon during the parsing of CSS. It also holds various files such as nsBlockFrame.cpp and nsIScrollableFrame.cpp which are also used by the Style System in the CSS Rendering Process. We believe this dependency exists so that all of the style elements of a frame are processed properly before being sent to the Frame Constructor to process the other elements that need to be painted onto the frame.

The Style System's StyleConsts.h file depends on various "Look and Feel" objects related to platform-specific rendering. This causes an unexpected dependency from the Style System on to the Platform-Specific Rendering and Widgets subsystem within Gecko.

**Unexpected Dependencies (External)**

Through our derivation, we found that the Style System has a dependency on the UI Layer which stems from a file called nsXULWindow.cpp. This file seems to deal with handling XUL Windows and their preservation. We believe this dependency exists in the event of interactions with a XUL window that requires some sort of style parsing.

Our concrete architecture derivation produced a two-way dependency with the Display Backend, both of which were unexpected dependencies that did not exist in our conceptual architecture. The Style System depends on files that deal with font metrics, and calls specifically upon nsFontMetrics.cpp. The Display Backend relies on the Style System for font constants in the file gfxFontConstants.h. We believe that this mutual dependency exists in order to avoid duplicated code, thus improving efficiency.

The file nsCSSPropertiesQS.h inside of SpiderMonkey, the JavaScript interpreter, causes a dependency from the Style System to SpiderMonkey. The Style System calls this file to fetch these CSS properties from SpiderMonkey.

# Frame Constructor

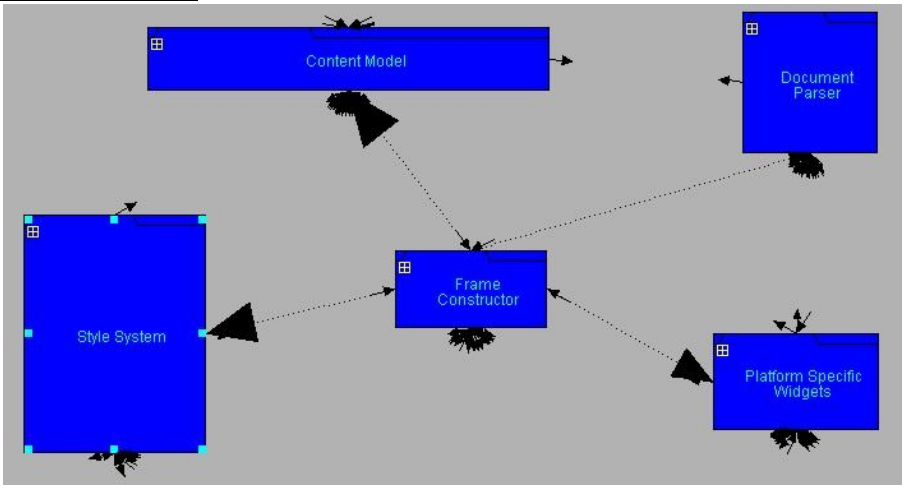


Figure 6 - Frame Constructor Dependencies

The Frame Constructor subsystem receives CSS data from the Style System as well as a DOM element from the Content Model and produces a frame in which to display the DOM element. We determined that the Frame Constructor consisted of a bulk of the components located in the Mozilla/layout folder. In deriving the conceptual architecture of Gecko, we came to expect a dependencies on both the Style System as well as the Content Model components of Gecko, since Mozilla's documentation reveals that the Frame Constructor requires a DOM tree object for which to build a frame, as well as CSS style rules specifying how each DOM element is to be displayed.

### Unexpected Dependencies (Internal)

Internally, unexpected forward dependencies of the Frame Constructor were on the Document Parser and Platform-Specific Rendering and Widgets. The dependency on the Document Parser was largely caused by calls to its HTML Content Sink (nsHTMLContentSink.cpp). In addition to translating the parser's method calls into a content model, the Content Sink file contains code to periodically trigger reflow operations, which are largely handled by the Frame Constructor in nsFrame.cpp. The Frame Constructor contains the file nsHTMLParts.h, which calls upon the Content Sink for additional frame-state bits used in constructing the Frame Tree.

The Frame Constructor must also call upon Platform-Specific Rendering and Widgets since aspects of both content and frames differ across various platforms. The Frame Constructor calls specifically upon header files in the widget folder such as nsIPluginWidget.h and nsILookandFeel.h to render frames which are consistent with that platform's appearance and metrics.

### Unexpected Dependencies (External)

External to Gecko, the Frame Constructor has dependencies both to and from Necko. The Frame Constructor depends on the networking file nsNetUtil.h, which deals with handling URIs and I/O streams. Additionally, the Frame Constructor's Document Viewer depends on Necko's nsURILoader.h when loading Documents. In turn, Necko depends on the Document Viewer when performing actions related to secure browsing (nsSecureBrowserUIIMPL.cpp) .

Another bi-directional dependency exists between the Frame Constructor and the UI Layer. In the toolkit folder, nsAppRunner.cpp depends on aspects of the Frame Constructor. The Frame Constructor depends on the Cross-Platform Front End when it encounters XUL.

## Platform-Specific Rendering and Widgets

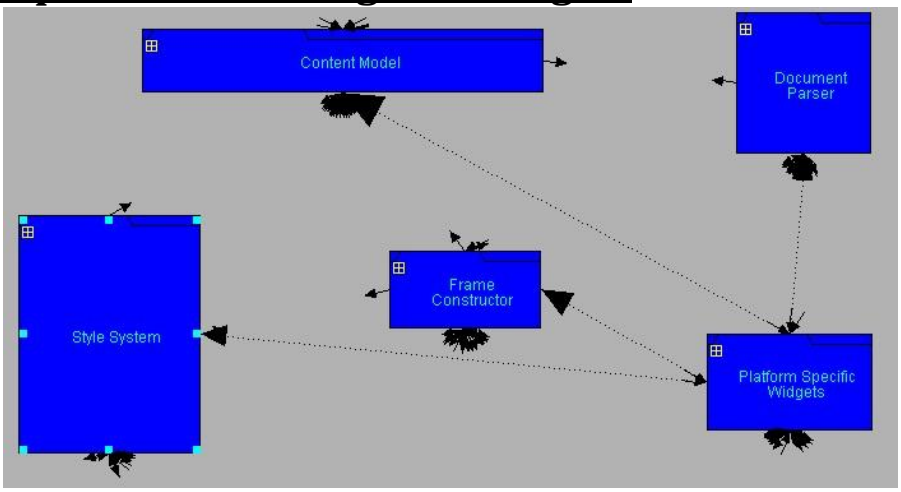


Figure 7 - Platform-Specific Rendering and Widgets Dependencies

Platform-Specific Rendering and Widgets provides platform-specific GUI data to the Frame Constructor. Our concrete interpretation of this component of Gecko consists of the Mozilla/widget folder, excluding the testing components which caused a number of false dependencies. A dependency on the Frame Constructor was expected since frame data is required for rendering the necessary widgets and platform-specific appearance of the browser. An external dependency on the Display Backend was expected because this subsystem requires interaction with the GTK+ adapter.

**Unexpected Dependencies (Internal)**

Forward dependencies on all components of Gecko are manifested in our derived concrete architecture, 3 of which were not present in the conceptual architecture. A dependency on the Style System is necessitated by widgets requiring CSS for their functionality, such as menus. Aspects of Platform-Specific Rendering responsible for supporting native themes, including nsNativeTheme.cpp, call upon nsStyleStruct.h in the Style System for internal style rules. Several "Look and Feel" files call upon StyleConsts.h in reference to style constants to ensure consistency and reduce duplicated code.

The dependency on the Document Parser is caused largely by various nsWidgetFactory files, which construct and provide support for each platform's widgets. These files call upon the Document Parser's nsHTMLFormatConverter file, which contains methods for converting HTML into formats usable by Platform-Specific Rendering and Widgets.

The source code for PuppetWidget.cpp contains a comment which reveals the reason for the bi-directional dependency between Platform-Specific Rendering and Widgets and TabChild.cpp in the Content Model. The developer asserts that "TabChild normally holds a strong reference to this PuppetWidget [...], but each PuppetWidget also needs a reference back to TabChild (e.g. to delegate nsIWidget IME calls to chome)".

**Unexpected Dependencies (External)**

In addition to the anticipated dependency on the Display Backend, Platform-Specific Rendering and Widgets is bi-directionally dependent with Necko and dependent on the Cross-Platform Front End in the UI Layer. The interaction with Necko seems to be caused primarily by Android-specific rendering files such as AndroidBridge.cpp. Necko's URI loader provides support for mobile applications on Android devices via nsOSHelperAppService.cpp, and Android-specific security and networking files in Necko depend on the Android Bridge file. Widget support files within Platform-Specific Widgets depend on nsAppShellCID.h in the XPFE folder for further support in rendering widgets used in web applications.

# Design Patterns

Throughout the derivation of our concrete architecture, we noticed evidence of several different design patterns being implemented. Often these design patterns are not as strictly implemented as their formal definition, but those definitions provide a strong basing for the patterns we noted. In particular, we found evidence of Adapter, Façade and Observer patterns within the Gecko Subsystem.

### Adapter Pattern

Adapter patterns are a natural fit for something like the Firefox system, which prides itself on its ability to function on all platforms. One of the largest evidence of an adapter pattern in Gecko is the Platform Specific Rendering and Widgits. This subsystem essentially acts as a bridge between the Frame Constructor and the Display Backend(GTK+ Libraries). The display backend is designed as a platform independent rendering engine, so it expects input and output to be uniform, regardless of which platform it is running on. This is where the Widgits within Gecko come in to play, doing any rendering necessary for the specific platform, and providing the input in a recognized form to the display backend. It is this that allows the rendering process in Firefox to be streamlined, without losing its necessary multi-platform use.

### Façade Pattern

The façade patters we saw evidenced took the form of Content Sinks, of which we found many in components such as content model. A content sink essentially acts as a repository for parsed data, to be accessed by other parts of the system. In Content Model we found many content sinks referring to XML and HTML data, which is where the Document Parser stores this data after it is parsed. The Content Model is able to access this data through these content sinks, rather than getting data from the document parser directly. This façade pattern isn't necessarily strict, as other components within content model and document parser jump past this façade in order to access other systems, but in general content sinks provide a simplified interface for the transfer of parsed data.

### Observer Pattern

The evidence of an observer pattern was harder to find within Firefox, but there is sections that specifically implement this pattern. One specific instance is an EventStates file we discovered in the Content Model. This file is responsible for updating the DOM Tree, and indirectly the Frame Constructor, whenever changes are made to the layout, style, content or various other components of the website data. This allows for the easy updating of many different components whenever a change to a single component is made, creating the form of one-to-many dependency that you would expect from an observer pattern.

# Sequence Diagram – Rendering a Webpage (Non-cached)

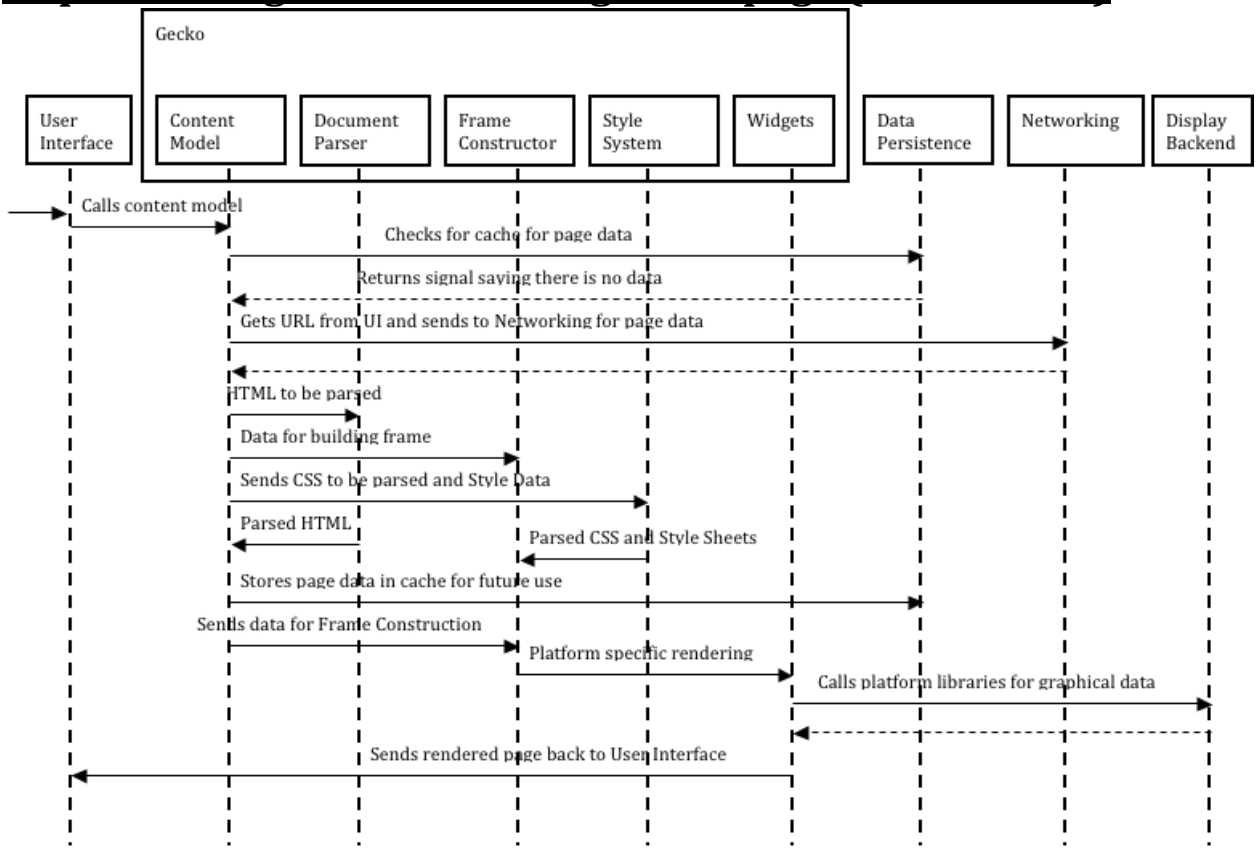


**Figure 8 - Sequence Diagram**

The above diagram is a sequence diagram for rendering a webpage that is not already in stored in cache memory and therefore must undergo the processes needed to render the page before returning it to the user.

The process begins with the user accessing Firefox's User Interface and requesting a particular page. The page request is sent to the Content Model. Before the URL is sent to Necko (Networking) to retrieve data needed for rendering the page, Gecko checks to see if the page information was previously stored in Firefox's Data Persistence subsystem. Since the page being rendered was not stored in the cache, a signal is returned indicating that the page does not exist in cache memory. After receiving the signal, Content Model sends the URL to Networking to retrieve the data needed for rendering the requested page, which Necko returns. Content Model then makes synchronous calls to the Document Parser, Style System and Frame Constructor so that they can begin their respective rendering processes concurrently. Parsed HTML and XML data are returned to the Content Model for further manipulation of the DOM tree. The tree is then sent to the Frame Constructor along with CSS data to complete the building of the frame for the page. The completed frame is then sent to Platform-Specific Rendering and Widgets for final rendering. The Widgets subsystem then calls the Display Backend for access to the different graphics libraries in order to render the page properly. It then sends properly rendered frame back to the User Interface so the user can view the page.

## Limitations and Lessons Learned

As a result of rapid evolution, little documentation for Firefox exists and comments in the source code are often not informative or non-existent. This made it difficult to pinpoint the root cause of dependencies in many cases, so it came down to our best interpretation of the source code as to why it existed. In addition to this, file and folder naming conventions were not always useful in determining which subsystem they belonged to.

We found it rather difficult to collaborate effectively because changes to the architecture in lsedit could only be reflected on one instance of the file. Even through using the Dropbox service it was hard to ensure that everyone had the same copy of the file, and re-uploading and downloading was a rather obtuse process to go through every time we made changes.

These limitations helped us to learn that meaningful comments are often crucial to the understanding of functionality of and relations between files and subsystems, especially in a large and complex software system. In relation to this, one cannot expect thorough or even adequate commenting or documentation and must rely on other means in order to understand and derive dependencies.

Initially we had thought that the data flow and purpose of each of the subsystems from the conceptual architecture would be clearly defined, but we learned that the folders contained files that were called beyond their initial scope, which is where unexpected dependencies arose from.

## Conclusion

A thorough analysis of subsystem relations in the Gecko browser and rendering engine indicates that Gecko implements an object-oriented architecture. Our concrete architecture derivation process uncovered a great deal of interdependency between the components of Gecko which was not present in our conceptual architecture. Our derivation also revealed more dependency between each component of Gecko and the components of Firefox that are external to Gecko. The most significant difference discovered in examining the concrete architecture of Gecko was the absence of the Image Library component, the elements of which were moved into the Content Model. Thus, the concrete architecture of the browser and rendering aspect of Gecko is comprised of the Content Model, Style System, Frame Constructor, Document Parser, and Platform-Specific Rendering and Widgets.

It is our belief that a majority of Gecko's unexpected dependencies exist due to developers' desire for expediency and efficiency, and may be related to Firefox's rapid release schedule. The next objective of this assignment will be the proposal of an architectural enhancement for Firefox 6.0.

# References


- http://dxr.mozilla.org/mozilla/index.html
- https://wiki.mozilla.org/Gecko:Home_Page
- https://wiki.mozilla.org/Gecko:Overview
- https://developer.mozilla.org/en/Gecko_Embedding_Basics
- https://developer.mozilla.org/en/Gecko_FAQ
- https://bugzilla.mozilla.org/show_bug.cgi?id=679615
- http://www-archive.mozilla.org/newlayout/doc/layout-2006-12-14/master.xhtml
- https://wiki.mozilla.org/Frame_inheritance_hierarchy
- http://dxr.mozilla.org/mozilla/mozilla-central/widget/src/xpwidgets/PuppetWidget.h.html