# Concrete Architecture of Firefox

Andre Campos - 0230481 - drick@uvic.ca

Bryan Lane - 0434698 - blane@uvic.ca

Neal Clark - 0429078 - nclark@uvic.ca

Sunpreet Jassal - 0323709 - ssjassal@uvic.ca

Stephen Hitchner - 0430473 - hitchner@uvic.ca

## Abstract

This report details the actual concrete architecture of Firefox; specifically the JavaScript interpreter, the XML parser and display backend components. The architecture extracted from the code did not follow the rigid boundaries of the conceptual architecture. After some investigation code modules were grouped together to try and recreate the conceptual layers. This was a difficult task because the code modules were not clearly separable. In the end there was a large amount of cohesion between code modules divided into the conceptual layers. This cohesion causes the actual extracted architecture to no behave like the expected conceptual layered architecture.

## Glossary

AOM - Application Object Model
API - Application Programming Interface
CSS - Cascading Style Sheet
DOM - Document Object Model
Firefox - web browser created by the Mozilla Foundation
GTK+ - aka GIMP toolkit, one of the two most popular widget toolkits for XWindows
HTML - Hyper Text Markup Language
kLOC - thousand lines of code
MFC - Microsoft Foundation Class library
MIME - Multipurpose Internet Mail Extensions
Mozilla - original public name of Mozilla Application Suite (aka SeaMonkey), created by the Mozilla Foundation

Necko - Networking Library of Firefox

NGLayout - Browser Engine of Firefox

NSPR - Netscape Portable Runtime. NSPR is a platform abstraction library, presenting a uniform interface to Firefox, no matter what platform it's running on.
OS - Operating System
OSI - Open Systems Interconnection. A general network reference model
OS X - an Apple operating system
PKCS - Public Key Cryptography Standard
PKI - Public Key Infrastructure
SSL - Secure Socket Layer. A cryptography protocol

TLS - Transport Layer Security. A cryptography protocol that is the successor of SSL
UI - User Interface
URI - Uniform Resource Identifier
URL - Uniform Resource Locator
WINAPI - Windows Application Programming Interface
X.509 - Standard formats for public key certificates and certification path validation algorithm
X11 - aka XWindows. Networking and display protocol that provides a standard toolkit to build GUIs on *NIX.

XDR - External Data Representation

XML - Extensible Markup Language
XPCOM - Cross Platform COM (component object model) implementation
XP - Cross platform, as in XPCOM, XPFE, XPInstall, XPIDL.
XUL - XML User interface Language

# 1 Introduction

FireFox is a product that is released by the Mozilla Foundation. FireFox is written in C/C++ and contains over 2,400 kLOC. The conceptual architecture is similar to most modern web browsers. The application is comprised of several independent components layered together to form what is commonly called a layered architecture.

According to the documentation for the Mozilla project FireFox had a modular layered architecture. The purpose of this report is to examine the concrete architecture of FireFox and identify if it matches the conceptual architecture. This was accomplished by using a number of software tools to examine the Mozilla codebase The directory structure of the FireFox code base was used to differentiate the different FireFox components. After analysis it was apparent that FireFox does exhibit a layered modular architecture. There were a number of dependencies discovered in the concrete architecture which will be discussed in detail below. Spider monkey and Gecko exhibited the most undocumented dependencies.

# 2 Concrete Architecture

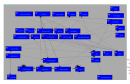

*Figure 1: Concrete Architecture of Firefox*

# 2.1 Comparison with Conceptual Architecture

As expected the actual architecture of Firefox differed from the conceptual architecture. The extracted concrete architecture did not exactly reflect the conceptual architecture. There was no one specific concrete code component

that contained each of the UI, Browser Engine, Rendering Engine, Data Persistence, Networking, JS Interpreter, XML parser or Display Backend. These conceptual components were comprised of several concrete code modules. In many cases, it was not apparent which concrete code modules were contained in which conceptual layer. We grouped together the code modules we felt resided in the each

of the conceptual architecture layers.

The conceptual layer architecture defined how each layer was to communicate: the UI layer only communicated with the Gecko layer, and Gecko in turn communicated with all the other layers. This is not the case in the concrete architecture extracted from the code. There are three components: MODULES, CONFIG, EXTENSIONS, NSPRPUB, XPCOM, XPINSTALL and INTL that connect to almost every other code components and it was unclear which of the conceptual layers they resided. These modules seemed to contain low-level functionality that is need by all other components. These groupings are shown in Figure 1

There are more connections between modules than expected. The concrete code modules did not just communicate we grouped into the UI layer communicate with modules in the Network, Data Persistence, XML Parser, Display Backend, and JS Interpreter modules. A clear layered architecture is not apparent from the extracted code, and it appears that several code modules are dependent on other code modules that break rigid communication rules of a layered architecture.

Detailed analysis of the JavaScript Interpreter, Display Backend and XML Parser are included below.

# 2.2 JavaScript Interpreter

The JavaScript Interpreter, called SpiderMonkey, resides in the JS folder of the firefox source code. Inside SpiderMonkey are a large amount of classes and three packages to support the processing of JavaScript. The most important of these classes and packages appear to be: liveconnect (package), xpconnect (package), jsinterp (class), jsxdrapi (class), jsdbgapi (class) and jsapi (class).

LiveConnect is the bridge between JavaScript and Java [2]. Mozilla's Modules package depends on LiveConnect, and LiveConnect depends on jsprf.o, jsapi.o, jscntxt.o, and jslog2.o (all classes in SpiderMonkey), and LiveConnect also depends on Mozilla's XPCOM package, which is external to SpiderMonkey (see Figure 2, below).

*Figure 2: LiveConnect's Dependencies*

XPConnect is a package that allows JavaScript objects access to XPCOM objects and vice-versa [3]. As shown by Figure 3, XPConnect depends on jsprf.o, jsxdrapi.o, jsapi.o, jsobj.o, jsinterp.o, jsdbgapi.o and jsdhash.o internally, and depends on XPCOM (for obvious reasons) externally.

*Figure 3: XPConnect's dependencies.*

JSInterp is the JavaScript interpreter called by the embedding components. JSInterp is also depended on by jsstr.o, jsscript.o, jsopcode.o, xpconnect, jsxml.o, jsparse.o, jsobj.o, jsapi.o, jsgc.o, jsiter.o, jsfun.o, jsdbgapi.o, jsapi.o, jsemit.o and jsarray.o; all of which are internal to SpiderMonkey. All of the classes that JSInterp depends on are internal to SpiderMonkey, including jsprf.o, jsstr.o, jsregexp.o, jsscript.o, jsscan.o, jsopcode.o, jsxml.o, jsobj.o, jsnum.o, jslock.o, jsapi.o, jsiter.o, jsfun.o, jsdbgapi.o, jscntxt.o, jsatom.o, jsarena.o, jsarray.o, and jsbool.o.

The other three of the mentioned important classes are all API classes, that is, they utilize the facade pattern to present a simpler interface to a variety of classes to make the system easier to to use. These APIs include JSAPI, the main API for SpiderMonkey; JSXDRAPI, the external data representation API for

SpiderMonkey; and JSDbgAPI, SpiderMonkey's debug API.

JSAPI is SpiderMonkey's main API, and as expected, depends on almost everything in the package, and is depended on by many classes in the package as well as all external packages that depend on SpiderMonkey (in other words, most SpiderMonkey calls from external sources are made through JSAPI). JSAPI is depended on by many external packages, including XPFE, XPInstall, Mozilla extensions, Docshell, Mozilla's toolkit, Storage, Security's manager package, Embedding's components package, DOM, Mozilla's modules, Caps and Content. The few classes that JSAPI isn't dependent on are likely either a) not intended to be accessed externally or b) dead code.

JSXDRAPI is SpiderMonkey's API interface for handling XDR (External Data Representation). According to the comment in the JSXDRAPI header file, the XDR API has three main parts: the state of serialization/deserialization APIs which allow users to serialize the runtime state for later restoration (most of which is implemented in various SpiderMonkey object classes), the callback APIs and the utility functions (most of which are implemented in jsxdrapi.c). JSXDRAPI is utilized by the Caps and Content packages externally, and jsregexp.o, jsscript.o, xpconnect, jsobj.o and jsfun.o internally.

JSDbgAPI is SpiderMonkey's debug API. It depends on JSAPI (not surprisingly), and a number of other internal files such as the interpreter, JSAPI and jsscript.o (among others) and is depended on by xpconnect, jsscript, jsopcode, jsobj, jsscope, jsgc, jsinterp, jsfun, jscntxt and jsexn internally, and Security's Manager package, DOM, Caps, Content externally, as well as SpiderMonkey's JSD package (the SpiderMonkey debugger).

The rest of the classes in the SpiderMonkey package are mostly classes to provide object types (i.e. jsbool.o) or libraries to provide popular functions to be used (ie. jsmath.o).

### Unexpected Dependencies

JSDbgAPI being depended on by so many external packages came as a bit of a surprise. Not surprisingly, SpiderMonkey's JSD package (SpiderMonkey's debugger) depends on it, but the security manager, DOM, Caps and Content packages all depend on it as well, as do several class objects internal to SpiderMonkey (such as XPConnect, jsopcode.o, jsinterp.o, jsscript.o, to name a few).

Three classes and one package in SpiderMonkey have no dependencies on anything, and nothing depends on them. These objects (jslong.o, jsutil.o, host_jskwgen.o and fdlibm) might be dead code.

## 2.3 XML Parser

XML Parsers are in parser module in the top-level mozilla directory, as shown by Figure 4. Two different parsing strategies are employed by the parser module: expat module performs DOM parsing, whereas xml performs SAX parsing.

*Figure 4: Parser module exposing internal modules and their dependencies*

htmlparser, inside parser module, has quite a few dependencies. It uses string functionality from nsprpub and obtains localization and unichar support from intl. htmlparser also depends on expat and xml as it delegates parsing requests to them. htmlparser is serving as a facade. While expat does not have any dependency on any other module in the system, xml module depends on xpcom as it uses stream input handling and string handling operations provided by xpcom. Note that neither xml not

expat communicate back to htmlparser after finishing XML parsing requests; the flow of control goes back to htmlparser, along with the parseing result, after parsing is complete.

### Deviation from Conceptual Architecture

One puzzling deviation from the conceptual architecture is that the parser module is not being invoked by any other module. In the conceptual architecture, it was expected that the rendering engine exclusively invokes the xml parsers but the concrete architecture shows no dependency to parser. There could be a dynamic dependency instead of static dependency (at compile time), which is what LSEdit captures and presents.

## 2.4 Display Backend

-andre campos 7/3/07 3:21 PM

The display backend consists of gfx, which is a a set of interfaces for a platform-independent device (drawing) context, as well as the interface with the underlying graphics interface. In addition, it has auxiliary modules for dealing with PostScript files and a shared module with common functionality. The figure below depicts the internal organization of what this report refer to as "Display Backend". The Drawing primitives and the Native Graphics Adapter are the most important part of the Display Backend.

Drawing Primitives is the component exposed to the rest of the Mozilla suite that is in charge of actually rendering everything from lines and squares to text on the screen. Although the diagram below does not show how external components use the Drawing Primitives, this sub-component is widely depended upon.

The Native Graphics Adapter follows the Adapter design pattern. Although the diagram does not show a dependency between Drawing Primitives and this component, we believe that the dependency does exist and that these two components talk through XPCOM. Otherwise, Drawing Primitives wouldn't be able to actually display graphics on the screen.

The dependencies between the components in the Display Backend and XPCOM was expected, considering that XPCOM is the "glue" of the Mozilla suite. Moreover, the Netscape Portable Runtime was also expected to have heavy dependecy upon, as it's a shared set of components through the entire Mozilla suite. However, the dependency to INTL, particularly from Drawing Primitives was a surprise, given that INTL deals with localization. It turns out that Drawing Primitives uses a module for handling unicode strings in INTL. Perhaps that module, namely unicharutil, should be placed in another component.

Overall, the display backend matches the conceptual architecture previously proposed. The few discrepancies happened between the display backend and components that weren't initially considered.

## Discrepancies

#### intl

This dependency was unexpected because intl is a module for internationalization and should not be invoked by gfx, which is a module that uses the underlying native graphics libraries to render the user interface. The dependency is actually between gfx and unicharutil, which is a sub-module of intl. Unicharutil is a module that implements string manipulation routines.

#### nsprpub

Contains C code for the cross platform "C" Runtime Library. The "C" Runtime Library contains basic non-visual C functions to allocate and deallocate memory, get the time and date, read and write files, handle threads and handling and compare strings across all platforms. This code is also known by the name, "nspr" and "Netscape Portable Runtime". This code originated in Mozilla Classic.

Given that nsprpub is such a basic component, it makes sense that gfx depends on it. This dependency was not foreseen but it's completely natural.

#### xpcom

Contains the low-level C++ interfaces, C++ code, C code, a bit of assembly code and command line tools for implementing the basic machinery of XPCOM components (which stands for "Cross Platform Component Object Model"). XPCOM is the mechanism that allows Mozilla to export interfaces and have them automatically available to Javascript scripts, to Microsoft COM and to regular Mozilla C++ code. Some low-level XPCOM classes and interfaces are also defined here (e.g. the event loop for all platforms). XPCOM is compatible and very similar to Microsoft COM (although XPCOM is cross-platform).

This component is very central in the Mozilla architecture; it can be seen as the "glue" between components. Therefore, a dependency between gfx and xpcom is should have been foreseen.

# 3 Conclusion

The conceptual architecture was compared to the actual architecture extracted from the code. It was found that the conceptual and actual architectures differed in several different ways. It was not immediately apparent which concrete code modules made up the conceptual architecture layers: there was no defined naming scheme. Once concrete code modules were groups together, there was a high amount of cohesion between all the grouped code modules, this does not follow the design goals of a layered architecture. There were several code modules that were not grouped within a conceptual

layer, these modules (XPCOM, INTL, NSPRPUB, XPINSTALL) contain low-level functionality that was needed by all the code modules.

# References


[1] http://www.mozilla.org/docs/source-directories-overview.html

[2] http://developer.mozilla.org/en/docs/JavaScript_Language_Resources

[3] http://en.wikipedia.org/wiki/XPConnect